Analysis and Design of Algorithms – A Handbook

Course Conducted by

Sukanta Das

Associate Professor

Department of Information Technology

Department of Information Technology

Indian Institute of Engineering Science and Technology, Shibpur

West Bengal, India – 711103

Preface

I was taking the course on *Analysis and Design of Algorithms* in my department for more than 10 years. This handbook has been developed from the notes of my lectures. The sequence of topics is based on the flow of lectures in my classes. Number theoretic algorithms and few more topics were also covered in this course, but are not included here. If my students get benefited from this writing, my effort will be fruitful.

Sukanta Das

Associate Professor
Department of Information Technology
Indian Institute of Engineering Science and Technology, Shibpur
Howrah, West Bengal, India - 711103

Dated: July 30, 2021

Contents

Chapter 1

Getting into Algorithms

Algorithms are step-by-step procedure to solve computational problems. Steps are to be well-defined. An algorithm takes some input and produces output in finite amount of time. So, algorithms are functions.

Although there is no universally-accepted definition of Algorithm, we sometime define it as following:

Definition 1 *An algorithm is a step-by-step procedure that solves a computational problem and possesses following characteristics.*

1. **Definiteness:** *Each step must be precisely defined; the actions to be carried out must be rigorously and unambiguously specified for each case.*
2. **Finiteness:** *It must always terminate after a finite number of steps.*
3. **Output:** *An algorithm always produces an output.*
4. **Input:** *Algorithms take finite amount of inputs.*

Sometime the fourth characteristic may become optional to some algorithms. For example, if the problem is - "Find the summation of 100 natural numbers", then the algorithm that solves this problem does not need an input. However, these are trivial cases. In general, algorithms take some input, on which it works.

1.1 Programs and Algorithms

Programs have many similarities with algorithms. Like algorithms, programs solve computational problems, and they are too step-by-procedure. Programs have also input and output. However, there are a number of *technical* differences.

Programs are written in formal language, whereas algorithms have no such burden. We generally use natural language to write down an algorithm; one can use pseudo-code or any formal language. If above mentioned four properties are followed and they are communicable to the intended persons, then the step-by-step procedure is an algorithm. It may be noted here that *Programs* are not necessarily computer programs. Programs can be written in any mathematical language. And the point is, syntax of the language is to be strictly followed. In case of computer programs, syntax of programming language is strictly followed.

Historically, the idea of algorithms came much before than that of programs. Euclid's procedure of finding GCD is considered as the first algorithm of the world, and it dates back to 300 BC. Whereas history of programs can hardly be of 200 years.

In the modern study of algorithms, however, we consider that programs and algorithms are complementary of each other. Finally we have to write programs if we want to use computer to solve a problem. It is sometime considered that the algorithms are theoretical study of (computer) programs. The relation between these two mathematical entities is generally viewed as:

Algorithm+Formal Language=Program

1.2 Problems

Before designing algorithm as solution to a problem, we need to clearly and precisely write down the problem[I]. The statement of a problem has to specify the set of inputs and desired output. Mathematically, problem is a binary relation from a set of possible inputs to a set of possible outputs.

1.2.1 An Example Problem: Searching

Problem statement: Given a list of records $R_1, R_2, \cdots R_n$, identified by keys $K_1, K_2, \cdots, K_n$ respectively, decide if a record against a given key K exists (if exists, output the corresponding record).

[I]By "problem" we refer to *computation problem* only. Throughout the discussion, this meaning will persist.

Here inputs of the problem are a list of records and a key, also called *search key*, and output is *yes* if record against the search key exits and *no* otherwise. Hence, a problem is binary relation. In case of searching problem, the binary relation is defined

$$\text{from}\{R_1, R_2, \cdots, R_n\} \times \{K\} \text{ to } \{\text{yes, no}\}$$

As a variant of the problem, output can be the desired record if it exists and 'no' if it does not. Strictly speaking, this is a new problem as its output set is different from the above. However, we roughly call both the problems as Searching Problem.

A number of algorithms may be developed to solve a given problem. For the searching problem, a number of algorithms exist, such as *Sequential Search Algorithm*, *Binary Search Algorithm*, *Fibonacci Search Algorithm*, etc.

1.2.2 An Example Algorithm: Sequential Search Algorithm

Let us now show a standard way of writing algorithm. Here we clearly specify input and output of the algorithm. It may also be noted that the steps are well-defined: nothing is written in vague words. When computer program is written from the algorithm then what is to be done is clearly mentioned in the step;s.

Algorithm 1 ***SSearch***
Input: *A list of records* $R_1, R_2, \cdots, R_n$ *identified by keys* $K_1, K_2, \cdots, K_n$ *respectively; K (Search Key).*
Output: *'YES' if the record exists; 'NO' otherwise.*

```
Step 1: Set $i \leftarrow 1$
Step 2: If $K = K_i$ , output 'YES' and exit.
Step 3: $i \leftarrow i + 1$
Step 4: If $i > n$, then output 'NO'; otherwise go to Step 2.
```

1.3 Proof of correctness

Whether an algorithm can produce its promised output for any possible input is to be carefully checked. This is the primary task to check correctness of an algorithm. Additionally, we have to check whether or not definiteness, finiteness, input and output are present in an algorithm. It is clear that, in the above algorithm input and output are clearly specified, every step is well defined and the algorithm halts after some finite steps. For this algorithm, it is trivial to show that the algorithm outputs ‘Yes’ if the desired record exists, and ‘No’ otherwise.

Chapter 2

Analysis of Algorithm

For the same problem statement, we can have arbitrarily many algorithms. So, to find out which algorithm is better than others, *performance* of an algorithm is *analyzed*. To measure this performance, the *cost* of getting the solution is to be found out. This cost is – how much *resources* of a computing system that particular algorithm demands to solve the problem. *Resources* can be of different types. Which particular resource is to be emphasized in finding the performance of an algorithm depends on the computing system and use of the algorithm.

Whenever an algorithm demands more amount of resources than another, we call that the first algorithm is more *complex* than the second. So to measure the goodness of an algorithm, we introduce the notion of *complexity*. Finding out complexities of algorithms, in order to understand their goodness, is know as *Analysis of Algorithms*.

2.1 Complexity Measure

Since algorithm is an abstract object, to understand its performance, we need to implement the algorithm as a program and run it on some computing machine. In this course, we always consider single-processor machine to run the program.

Type of machine: Single-processor computer with single memory unit.
That is, parallel processing is not allowed. In such a machine, following are the most important *resources* that are needed to run a program.

- Time – Amount of time needed for execution of the program (*time*

complexity)
- Space – Amount of *extra* space needed for execution of the program (*space complexity*)
- Disk access – Number of disk access needed to retrieve the data from secondary storage

For different type of machine and computing, however, other resources such as *Messages* (Number of message passed in case of distributed systems), *Packets* (Number of packets transmitted for Internet), *Bandwidth*, etc. are to be taken care of.

Normally, complexity is dependent on the size of the input (n), so, it is represented as a function of the size of the input. For example, time complexity is $T(n)$ and space complexity is $S(n)$. In this course, we shall deal with only time and space complexities. We assume that we have enough space to accommodate data and program in main memory, which nullifies the need of secondary memory.

However, demand of resource by an algorithm depends on types of possible inputs. There are mainly three *cases* of possible inputs which are considered to study the complexity of an algorithm:

1. Best case: The situation when the algorithm requires minimum number of resources. Usually, we are not interested in this.

2. Worst case: The situation that has maximum resource demand for the algorithm. For example, if we consider time complexity, then the worst case is the situation when maximum amount of time is required to get the answer.

3. Average case: Average resource requirement of the algorithm considering all possible situations.

In general, average case complexity is as bad as the worst case. We normally are interested in the worst case complexity of an algorithm, which indicates that the algorithm cannot take more resource than it under any circumstances So, if not mentioned otherwise, by complexity we shall mean worst case time complexity only.

2.2 Role of Model of Computation

As mentioned, to measure the complexity of an algorithm, that is, its resource requirement, the algorithm has to be implemented on some machine. However, same algorithm may give different complexity values depending on the implementation. Even same program may require different execution time on different machines as the machines can have different speeds! That is, complexity becomes machine dependent. But, it should not be so. Therefore, we need a standard machine to measure complexity. To deal with this, an abstract machine is considered as the standard platform for measuring the complexity. This abstract machine is called a *model of computation*. So, to analyze an algorithm, that is, how good or bad it is, we always need a model of computation. Any model of computation is a formal system associated with a formal language for implementing the algorithm. Following are some popular models of computation:

- Turing Machine (TM) (see Figure 2.1),
- Random Access Machine (RAM) Model,
- Random Access Stored Program (RASP),
- Counter Machine or Register Machine, etc.

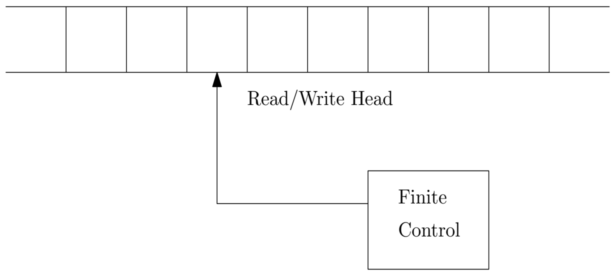


Figure 2.1: Schematic diagram of a Turing Machine

Among the above models, Turing Machine (TM) is the primitive model of computation and the most *expressive* one. One can choose any of these models, but the choice has impact on complexities of algorithms. For example, if $T_{TM}(n)$ and $T_{RAM}(n)$ are worst case time complexities of same algorithm on Turing machine and RAM model respectively, then generally $T_{TM}(n) > T_{RAM}(n)$. Since the algorithms are implemented on presently-available computers, we should choose a model which has close resemblance with standard single-processor computer architecture. Here we choose *RAM Model* as our model of computation as it has many similarities with standard *von Neumann architecture*. Note, however, that all of these models are computationally *universal*.

2.3 Random Access Machine (RAM) Model

This model is also known as *Unlimited Register Machine* (URM) model. The name comes from its feature of having unlimited number of registers that store integer values and are accessible randomly. The first register r_0 is the accumulator where all computations are performed. The following is the block diagram of RAM Model.

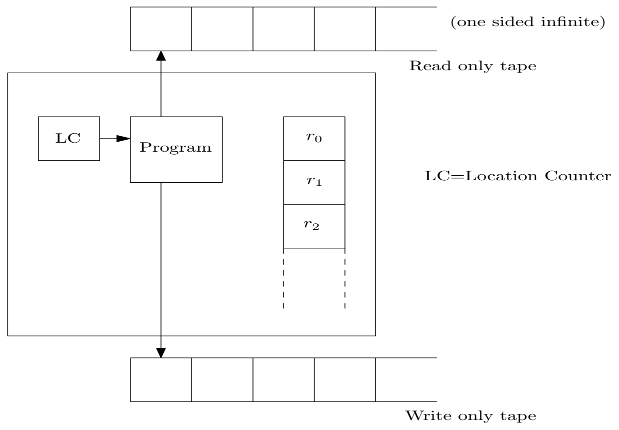


Figure 2.2: Block diagram of RAM Model

The function of *Location Counter* (LC) is similar to the program counter in a computer. A program in RAM model is a sequence of instructions, which interacts with registers, input and output buffers. Input buffer is read only whereas output buffer is write only. Likewise registers, input and output buffers are also one-way bounded infinite and contain infinite number of boxes/cells. From the cell of input buffer under the read pointer, an integer can be read and to the cell under the write pointer an integer can be written. After each read or write operation, the corresponding pointer head moves to the next cell.

RAM Model is a single accumulator machine with an assumption that each register and input/output cell can contain an integer of arbitrary length.

So, size of an integer is one word in RAM model. The model uses only 12 instructions, which are sufficient to write any program. It may be noted here that any real-life microprocessor offers many more instructions to write a program. For example, 8085 microprocessor, an old microprocessor from Intel offer around 100 instructions. However, here target is use a very small set of instructions which can mimic the essential instructions of any single-processor machine. Following are the instructions that we shall use to develop programs.

1. $READ$: Read from input buffer
2. $WRITE$: Write to output buffer
3. $LOAD$: Load the register some value
4. $STORE$: Store content of accumulator to some register
5. ADD: Addition of some value with the value of accumulator
6. SUB: Subtraction of some value from the value of accumulator
7. $MULT$: Multiplication of some value with the value of accumulator
8. DIV: Division of the value of accumulator by some value
9. $JUMP$: Unconditional jump to some label
10. $JZERO$: Jump if content of the accumulator becomes 0
11. $JGTZ$: Jump if content of the accumulator is greater than 0
12. $HALT$: Terminate or exit the program

RAM allows direct as well as indirect addressing. If the operand is '$= i$', it indicates integer i; if the operand is 'i', it means the content of register number i (direct addressing); however, if the operand is '$*i$', it shows indirect addressing. SWe use a function c to get content of a register. That is, $c(i)$ gives the content of register i. Let us take "ADD" as example to understand its meaning under different operands.

$$\begin{array}{lll} ADD & = i & \Longrightarrow\ c(0) \leftarrow (0) + i \\ ADD & i & \Longrightarrow\ c(0) \leftarrow c(0) + c(i)\ [\textit{direct addressing}] \\ ADD & * i & \Longrightarrow\ c(0) \leftarrow c(c(i))\ [\textit{indirect addressing}] \end{array}$$

$READ\ \ i$ means read the input cell under $\uparrow$ and store the value into r_i. The pointer $\uparrow$ will now shift to the next slot. Similarly, $WRITE\ \ i$ means write the content of register r_i into the output buffer cell under $\downarrow$ and move pointer $\downarrow$ to the next slot.

Note that, more instructions are added to make a computer faster and also sometimes specifically for operating system designers. For example, *TSL* of 8085 is included to solve *mutual exclusion* problem. But, these 12 instructions of RAM are sufficient to develop any program. Here, the idea is, once we have our algorithm, we can develop the code for it and see how much resources (time/space) it takes.

However, if we choose RAM model to analyse our algorithms, our next task will be to develop program for the algorithms on RAM model using its 12 instructions.

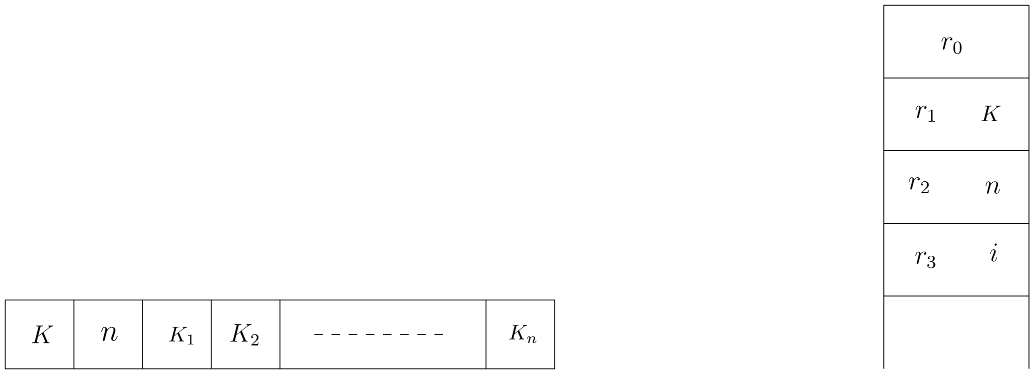
2.3.1 RAM implementation of Algorithm: SSearch

To find the complexity of the sequential searching algorithm (Algorithm 1), we develop a program for the RAM Model. We assume the following for ease of implementation:

- Input records are keys only.
- The input tape contains the search key (K), followed by the number of keys n, followed by the individual keys $K_1, \cdots, K_n$ in this sequence (see Figure 2.3a).
- The initial position of the reading head is at the cell containing K.
- We use r_1 to store K, r_2 to store n and r_3 to store a variable i (see Figure 2.3b).
- If the search key is present (successful search), we write 1 in the output tape; otherwise, 0 is written.

Now, following is the RAM implementation of Algorithm 1 (SSearch):

```
1.  $READ$ 1                      // Read $K$ into $r_1$
2.  $READ$ 2                      // Read $n$ into $r_2$
3.  $LOAD$ $=1$                   // Load 1 into $r_0$
4.  $STORE$ 3                     // Store value of $r_0 = 1$ into $r_3$, this indicates $i \leftarrow 1$
5.  $l_1 : READ$ 0                //Read $K_i$ into $r_0$
6.  $SUB$ 1                       //$r_0 \leftarrow K_i - K$
7.  $JZERO$ $l_2$                 //If $r_0 = 0$ go to label $l_2$
8.  $LOAD$ 3                      //Load $r_3 = i$ into $r_0$
9.  $ADD$ $=1$                    //Increment content of $r_0$ by 1
10. $STORE$ 3                     // Store $r_0$ into $r_3$, this implies $i \leftarrow i + 1$
```



(a) Input buffer (b) Registers

Figure 2.3: Content of Input buffer and Registers for RAM implementation of Algorithm 1

11. $SUB\ \ 2$	$//r_0 \leftarrow r_0 - r_2 = (i+1-n)$
12. $JGTZ\ \ l_3$	// If $i+1 > n$, go to label l_3
13. $JUMP\ \ l_1$	// Otherwise go to label l_1
14. $l_2 : WRITE\ \ =1$	// Write 1 in output buffer for 'YES'
15. $HALT$	// Exit
16. $l_3 : WRITE\ \ =0$	// Write 0 in output buffer for 'NO'
17. $HALT$	// Exit

Here lines 1 and 2 are for the initialization, lines 3 and 4 correspond to Step 1, lines 5, 6 and 7 are for Step 2, lines 8, 9 and 10 are for Step 3 and lines 11, 12 and 13 indicate Step 4 whereas lines 11 and 12 implement the *if* condition for output 'NO'.

2.3.2 Complexity Analysis of Algorithm: SSearch

As mentioned, complexity of the algorithm can be derived from the program. For the searching problem, the worst case is when the searching key is not present. There are two cost criteria – *uniform cost criterion* and *logarithmic cost criterion*, to get the cost of a program. Here we assume ***uniform cost criteria***, that is, each instruction has same cost – to execute each instruction unit time is needed.

Time Complexity: To check worst case time complexity, we consider the maximum number of times each instruction of the program gets executed. In this algorithm, the worst case is that the record against the search key (K) does not exist in the list.

We can observe that, the instructions (1- 4) are executed once taking 4 units of time. The instructions (5 - 13) take 9 time units, but that is actually a loop. So these 9 steps will be performed for every input until there is a jump on l_2 or l_3. As we are considering the worst case, for the last input after performing instruction 12, it will jump to l_3. This time it will not perform instruction 13. So for n number of keys, instructions (5 - 13) will take $9n - 1$ units of time. Again, for worst case l_3 will be performed and 2 units time is required for that. So the total time requirement is

$$T(n) = 4 + 9n - 1 + 2 = 9n + 5.$$

This is the worst case time complexity of the algorithm assuming the uniform cost criterion and considering the above implementation.

Space Complexity: For calculating space complexity, we need to know the numbers of extra registers used except the accumulator. The space needed to store the input is not considered as part of space complexity. As three registers (K at r_1, n at r_2, i at r_3) are used in the program, $S(n) = 3$. It is constant here, but generally space complexity also depends on n, the input size.

Obviously, the uniform cost criteria is for hypothetical machine and is not realistic. So, sometimes we assign different time units for different instructions depending on how much bits that process. This is known as *logarithmic cost criteria*. Here, cost is assigned with respect to the number of bits used by the operands. We take $\log_2$ of that value. It is more realistic than the uniform cost criterion.

Note: Complexities found out above depends on the program written against the algorithm. There are a number of ways of implementing the algorithm. So based on the implementation, complexities may change.

2.4 More on Mathematical Models of Computation

It is noteworthy to point that, if we change the model of computation, $T(n)$ and $S(n)$ will also change. Let us consider that, for an algorithm, the program implementation for RAM needs T_1 time units, for Register machine needs T_2 time units, whereas, the program for RASP model needs T_3 time units. In general, $T_1 \neq T_2 \neq T_3$ (if $T_1 = T_2 = T_3$, it is a special case).

In case of RAM model, a program can not change itself during its execution. This contradicts with the modern architecture of a computer, where a program can also be changed. During execution RAM model is like TM, except here location counter (LC) and register set are included. For TM, part of the I/O tape can work as the register set. Whereas, RASP is very much similar to von Neumann architecture where we store program and data in the same place and it can change the code of the program. Models reflect the essential properties of the actual computer. We always try to make models as simple as possible.

Computationally all these models are equivalent – complexity measured in any of these models of computations are *polynomially related*[I] with each other. For example, if we get $T_1(n), T_2(n), T_3(n)$ as the time complexities of same algorithm on different models, then they can be shown as *polynomially related*.

Turing machine is called the *primitive* model of computation. However, normally we do not use TM in algorithm analysis as it is more difficult to use. On the other hand RAM model has direct similarities with standard computer architecture.But, if we want to get more perfection and we want to know the lower bound of an algorithm, TM should be used. By default, we shall implement our algorithms on RAM model and find time complexity based on that.

Remark: All these models are based on stored program architecture derived from TM and the concept of computability theory is built on that. So, if a new model is developed which does not follow this architecture, then this analysis of algorithm will not work. Then a new kin d analysis may be required to understand the performance of algorithms.

[I]For any two functions $f(n)$ and $g(n)$, $f(n)$ is *polynomially related* to $g(n)$, if there exists two polynomials p_1 and p_2 such that the following relation holds:

$$f(n) \leq p_1(g(n)) \text{ and } g(n) \leq p_2(f(n))$$

Chapter 3

Growth of Functions

We have already observed that the time (resp. space) complexity is a function of the input size n, which is a natural number ($n \in \{0, 1, 2, \cdots\}$). This $T(n)$ (and $S(n)$) is a non-negative function.

3.1 Observation on Time Complexity

Consider the time complexity of the Algorithm: SSearch (see Page 11) which is $T(n) = 9n+5$ as per our implementation. Let us consider another function $T'(n) = 9n + 10$. Here, $T(n) < T'(n)$. But when n grows to a large number, do these two functions show significant difference? Consider the following table which shows growth of these function when n grows:

n	$T(n) = 9n + 5$	$T'(n) = 9n + 10$
0	5	10
1	14	19
2	23	28
3	32	37
5	50	55
10	95	100
100	905	910
1000	9005	9010
100000	900005	900010

Observe that, for small values of n the difference of these two functions is noticeable, but for large value of n, say 100000, the difference is insignificant.

Now, take another two functions $T(n) = n^2$ and $T'(n) = n^2 + 5n + 10$. Growth of these two function against the increase of n is noted in the following table:

n	$T(n) = n^2$	$T'(n) = n^2 + 5n + 10$
0	0	10
1	1	16
2	4	24
3	9	34
5	25	60
10	100	160
100	10000	10510
1000	1000000	100510
100000	10000000000	10000500010

Here observe that for small values of n, difference between these two functions is significant. But the significance of difference reduces when n grows. For $n = 100000$, $T(n) : T'(n) = 0.99995$. That is, they are very close to each other, and their difference has become insignificant. Further observe that for large n, $T(n)$ and $T'(n)$ are dominated by n^2. In other words, growths of these two functions are dominated by n^2 and $5n + 10$ of $T'(n)$ becomes insignificant

As another example, suppose $T(n) = n^3 + 2n^2 + 3n \log_2 n + 2$ is the time complexity of an algorithm and $T'(n) = n^3$ is of another. Obviously, $T(n) \neq T'(n)$ and $T(n) > T'(n)$. But like before, we observe the same thing:

n	$T(n)$	$T'(n)$
0	2	0
1	10	1
2	34	8
4	142	64
8	754	512
16	4802	4096
32	35298	32768
100	1021995.157	1000000
1000	1002029899	1000000000

From these tables, we can find that, when n is small, the percentage of difference between $T(n)$ and $T'(n)$ is noticeable. But, if n is increased, the difference is very less. For large values of n, $T(n) : T'(n)$ is almost 1:1.

The growth of the functions depends on n. Above example shows that the growth is practically determined by the highest term of the functions whereas lower terms become insignificant. In case of analysis of algorithms, we need not to know always exact complexity because of many reasons. Firstly, exact calculation of complexity depends on implementation of the algorithm on some standard machine. If implementation changes or architecture of machine changes, the calculation will be affected. However, in this case growth of function remains unaffected. Secondly, size of input can be arbitrarily large, for which final value of function depends on the highest term and the lower terms are insignificant. So it is good to *approximate* the complexities by simpler functions, growth of which are similar to that of complexities. This is done through *asymptotic notations*.

3.2 Asymptotic Notation

The first asymptotic notation was introduced by Paul Batchman in 1892, which is known as the Big-oh(O) notation. It primarily had no relation with analysis of algorithms.

3.2.1 Big-Oh Notation

$f(n) = O(g(n))$ if there exists c, n_0 such that $0 \leq f(n) \leq c \cdot g(n)$ for all $n \geq n_0$ where c and n_0 are two positive constants. Formally,

Definition 2 $O(g(n)) = \{f(n)$: there exists two positive constants c, n_0 such that $0 \leq f(n) \leq c \cdot g(n),\ \forall n \geq n_0\}$

Therefore, $O(g(n))$ is a set of functions and $f(n)$ is an element of this set, that is, $f(n) \in O(g(n))$. However, by abusing '=' symbol, we write $f(n) = O(g(n))$ to mean $f(n) \in O(g(n))$.

For Big-oh notation, $f(n) \leq c \cdot g(n)$ with any value of c. So, for $f(n) = n^3 + 2n^2 + 3n \log_2 n + 2$, if we consider $n_0 = 8$, then this condition is satisfied for $c = 2$, that is, $f(n) \leq 2 \cdot n^3$ for all $n \geq n_0$ where $g(n) = n^3$. Growth of functions $f(n), g(n)$ and $2n^3$ are of same order. So we write it as $f(n) = O(n^3)$.

Now, recall the time complexity of Algorithm: SSearch (Algorithm 1):

$$T(n) = 9n + 5 \leq 10n\ \forall n \geq 5.$$

Hence, $T(n) = O(n)$ (see Figure 3.1). We can observe that the constants (values 9 and 5) are not so significant if we consider the order of growth.

Similarly, take $f(n) = n^2 + 5n + 10$ and $g(n) = n^2$, so,

$$0 \leq f(n) \leq c \cdot g(n),\ \forall n \geq n_0 \Rightarrow 0 \leq n^2 + 5n + 10 \leq c \cdot n^2, \forall n \geq n_0$$

Let's take $c = 2$. Then

$$0 \leq n^2 + 5n + 10 \leq 2n^2,\ \forall n \geq 7;$$

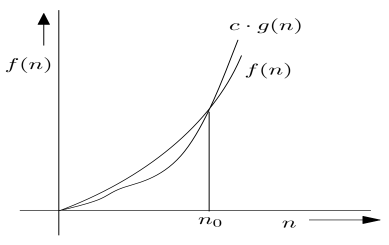


Figure 3.1: Growth of Function for Big-oh (O) Notation

that is, here $n_0 = 7$. So $f(n)$ is an element of the set $O(n^2)$. Now, consider

$$T''(n) = n^2 + 9n \log n + n \log\log n$$

then also,

$$T''(n) = O(n^2).$$

If we are satisfied with asymptotic notation and approximate analysis, we can avoid writing program on RAM model. Then we assume that a step of an algorithm take a constant amount of time when it runs of some computer. When machine changes, only the values of constant changes, but it remains as constant.

Complexity Analysis using Big-oh Notation

Let us now analyse the time complexity of Algorithm 1 (Page 3) in worst case without referring to model of computation.

Let Step 1 takes c_1 amount of time.
 Step 2 takes c_2 amount of time.
 Step 3 takes c_3 amount of time.
 Step 4 takes c_4 amount of time.

Observe that, only once out of the n times, Step 4 gives an output. However, ignoring this minute difference we assume that Step 4 always takes c_4 amount of time. Hence, the total time required in the worst case is

$$\begin{aligned} T(n) &= c_1 + (c_2 + c_3 + c_4)n. \\ &= c \cdot n + c_1 \text{ where } c = c_2 + c_3 + c_4 \\ &= n(c + \tfrac{c_1}{n}) \\ &\le n \cdot (c + c_1),\ \forall n \ge 1 \\ &= c_0 \cdot n,\ \forall n \ge n_0 \text{ where } c_0 = c + c_1,\ n_0 = 1 \end{aligned}$$

i.e. $$T(n) = O(n).$$

More on Big-oh Notation

In analysis of algorithm, we are interested in higher values of input size n, so, Big-oh notation is important. Big-oh notation represents *upper bound* of a function. That is, if $f(n) = O(g(n))$ then $g(n)$ is *an* upper bound of $f(n)$.

Example 1 $f(n) = a_k n^k + a_{k-1}n^{k-1} + \cdots + a_1 n + a_0,\ a_k > 0$ and other coefficients of the polynomial are nonnegative. Prove that $f(n) = O(n^k)$.
Ans. $f(n) = n^k(a_k + \frac{a_{k_1}}{n} + \cdots + \frac{a_1}{n^{k-1}} + \frac{a_0}{n^k})$
$\leq n^k \cdot (a_k + a_{k-1} + \cdots + a_1 + a_0),\ \forall n \geq 1$
$= A \cdot n^k$ where $A = a_k + a_{k-1} + \cdots + a_1 + a_0$
Clearly, $f(n) \geq 0$. So, $0 \leq f(n) \leq A \cdot n^k,\ \forall n \geq 1 \Rightarrow f(n) = O(n^k)$.

Example 2 Are $O(n^2+n) \equiv O(n^2)$ and $O(N+M) \equiv O(Max(N,M))$ true?
Ans. Both are true. In O-notation, we interpret '+' as 'Max'.

Example 3 Let $f(n)$ and $g(n)$ be two non-negative functions. Prove that,

1. $\max(f(n), g(n)) = O(f(n) + g(n))$;
2. if $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$;
3. $O(f(n) + g(n)) = O(f(n))$, if $f(n)$ is the maximum of both.

Ans. 1 If $f(n)$ and $g(n)$ are two non-negative functions, then
$\max(f(n), g(n)) = f(n)$ if $f(n) > g(n)$
or, $= g(n)$ if $g(n) > f(n)$.

Therefore, $\max(f(n), g(n)) \leq c \cdot (f(n) + g(n))$, if $f(n)$ and $g(n)$ are non-negative functions and $c = 1$. That is,

$$\max(f(n), g(n)) = O(f(n) + g(n)).$$

Ans. 2

$$f(n) = O(g(n)) \Rightarrow f(n) \leq c_1 \cdot g(n),\ \forall n \geq n_1$$

$$g(n) = O(h(n)) \Rightarrow h(n) \leq c_2 \cdot h(n),\ \forall n \geq n_2$$

where c_1, c_2, n_1, n_2 are positive constants. Therefore,

$$f(n) \leq c_1 \cdot g(n) \leq c_1 \cdot c_2 \cdot h(n) = c \cdot h(n),\ \forall n \geq \max(n_1, n_2),$$

where $c = c_1 \cdot c_2$. That is,

$$f(n) = O(h(n)).$$

Ans. 3 $(f(n) + g(n)) = f(n) \cdot (1 + \frac{g(n)}{f(n)})$
$\leq f(n)(1+1) = 2f(n)$ as $f(n) > g(n)$.
Therefore, $O(f(n) + g(n)) = O(f(n))$ if $f(n)$ is the maximum.

Here c, n_0 can be any positive integer. So, can we say

- $n = O(n^2)$? – Yes, as $0 \leq n \leq c \cdot n^2$ for $c = 1$ and $n_0 = 0$.
- $n^2 = O(n)$? – No, because, here the relation $0 \leq n^2 \leq c \cdot n$ has to be satisfied for all $n \geq n_0$, but, for whatever be the value of c, n_0, after some time n^2 will cross $c \cdot n$.

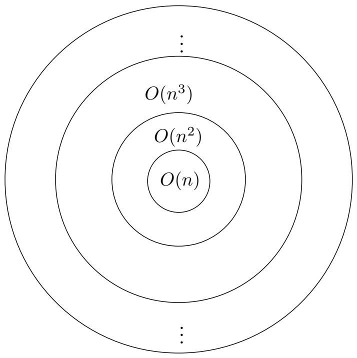


Figure 3.2: Hierarchy of Sets in Big-oh Notation

Therefore, behaviour of growth for any function can not be changed by choosing c & n_0 and these two values are related. When we increase c then n_0 decreases and vice versa. Also observe that, $n = O(n)$, $n = O(n^2)$, $n = O(n^3)$ and so on. Hence,

$$O(n) \subset O(n^2) \subset O(n^3) \subset \cdots$$

Similarly,

$$O(n) \subset O(n^2) \subset O(n^2 \log n) \subset \cdots$$

Therefore, although Big-oh notation gives upper bound of a function, this upper bound is not asymptotically *tight* (see Figure 3.2).

3.2.2 Big-Omega Notation

To understand the lower bound of a function in the analysis of algorithm, Donald Knuth proposed Big-omega (Ω) notation.

Definition 3 $\Omega(g(n)) = \{f(n)$: there exist two positive constants c and n_0 such that $0 \leq c \cdot g(n) \leq f(n),\ \forall n \geq n_0\}$

This represents that, the value of the function $f(n)$ can never be lower than the value of $c \cdot g(n)$ whenever $n \geq n_0$ (see Figure 3.3).

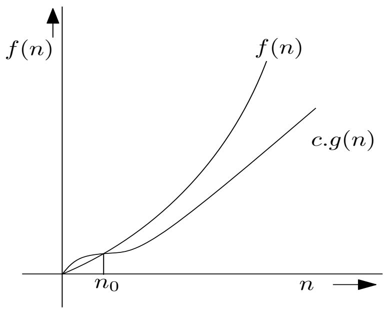


Figure 3.3: Growth of Function for Big-omega (Ω) Notation

Let $f(n) = 2n + 1$ and $g(n) = n$. Here $g(n) < f(n)$ for all $n \geq 0$. Hence we can say $f(n) = \Omega(n)$. However, is $f(n) = 2n + 1 = \Omega(n^2)$, that is, $0 \leq c \cdot n^2 \leq 2n + 1$ true for all $n \geq n_0$? No, because, for any c, n^2 will exceed $2n + 1$ after some time, that is, n_0 does not exist. Now take, $f(n) = 2n^2 + n + 1$. Is $f(n) = \Omega(n)$ true? Here, obviously it is true. So,

$$\Omega(n) \supset \Omega(n^2) \supset \Omega(n^2 \log n) \supset \Omega(n^3) \cdots$$

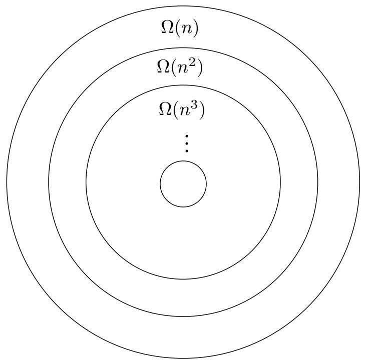


Figure 3.4: Hierarchy of Sets in Big-omega Notation

Although Big-omega (Ω) notation shows the lower bound of a function, like Big-oh notation,it is also not an asymptotically *tight* lower bound (see Figure 3.4).

Example 4 Consider $f(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$, where $a_k > 0$ and other coefficients are non-negative. Prove that $f(n) = \Omega(n^k)$.

Ans. We have to show that there exists c, n_0 such that $0 \leq c.n^k \leq f(n)$, $\forall n > n_0$. Now,

$$f(n) = a_k n^k + a_{k-1}n^{k-1} + \cdots + a_1 n + a_0$$
$$\geq a_k \cdot n^k \ \forall n \geq 0$$ [as the coefficients are non-negative and $a_k > 0$]

Hence, $f(n) = \Omega(n^k)$.

3.2.3 Big-Theta Notation

We have mentioned that O-notation and Ω-notation are not asymptotically *tight.* $f(n) = O(g(n)) \nRightarrow g(n) = O(g(n))$, for example, $n = O(n^2)$, but $n^2 \neq O(n)$. *A bound is called 'tight' for a function if upper bound of it is equal to the lower bound in terms of asymptotic notation.*

In complexity analysis, it is always desirable to find the tight bound of the function. To represent asymptotically tight bound, Big-theta (Θ) notation was introduced by Donald Knuth.

Definition 4 $\Theta(g(n)) = \{f(n)$: there exist three positive constants c_1,c_2 and n_0 such that $0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\ \forall n \geq n_0\}$

$f(n) = \Theta(g(n))$ if it is bounded both sides by $c_1.g(n)$ and $c_2.g(n)$ for all $n \geq n_0$ (see Figure 3.5).

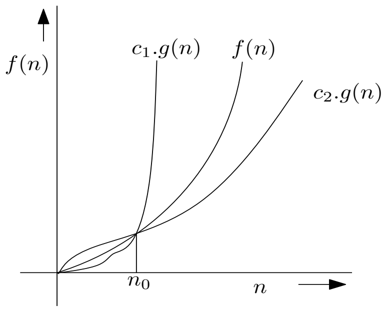


Figure 3.5: Growth of Function for Big-theta (Θ) Notation

Theorem 1 : *$f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.*

Proof : Let us first assume that $f(n) = \Theta(g(n))$. Then, by definition, there exists three positive constants c_1, c_2 and n_0 such that

$$0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n),\ \forall n \geq n_0 \tag{3.1}$$

So, from Equation 3.1, we can say that

$$0 \le c_1 \cdot g(n) \le f(n), \ \forall n \ge n_0 \tag{3.2}$$

and

$$0 \le f(n) \le c_2 \cdot g(n), \ \forall n \ge n_0 \tag{3.3}$$

Therefore, from Equation 3.3, $f(n) = O(g(n))$ and from Equation 3.2, $f(n) = \Omega(g(n))$.

Conversely, let $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$. Now, $f(n) = O(g(n)) \Rightarrow$ there exists two positive constants c_2, n_2 such that $0 \le f(n) \le c_2 \cdot g(n), \ \forall n \ge n_2$. Whereas, $f(n) = \Omega(g(n)) \Rightarrow$ there exists two positive constants c_1, n_1 such that $0 \le c_1 \cdot g(n) \le f(n), \ \forall n \ge n_1$.
Therefore, $0 \le c_1 \cdot g(n) \le f(n) \le c_2 \cdot g(n), \ \forall n \ge n_0$, where $n_0 = \max(n_1, n_2)$.
That is, $f(n) = \Theta(g(n))$. $\square$

For example, $2n^3 + 5n^2 + 12n = \Theta(n^3)$. In algorithm analysis, we mainly use Θ-notation which depicts both the upper and lower bound of a function.

Example 5 Let $f(n)$ and $g(n)$ be two functions. Prove that, $\max(f(n), g(n)) = \Theta(f(n) + g(n))$.
Ans.

$$\max(f(n), g(n)) \le (f(n) + g(n)), \ \forall n \ge 1 \tag{3.4}$$

Now, if $f(n) \ge g(n) \ \forall n \ge n_0$, that implies, $2f(n) \ge (f(n) + g(n)) \ \forall n \ge n_0$, that is, $f(n) \ge \frac{1}{2}(f(n) + g(n)) \ \forall n \ge n_0$. Therefore,

$$\max(f(n), g(n)) \ge \frac{1}{2}(f(n) + g(n)), \ \forall n \ge n_0 \tag{3.5}$$

Similarly, if $f(n) \le g(n) \ \forall n \ge n_0$ then also,

$$\max(f(n), g(n)) \ge \frac{1}{2}(f(n) + g(n)), \ \forall n \ge n_0 \tag{3.6}$$

Combining Equations 3.4, 3.5 and 3.6, we get that

$$\max(f(n), g(n)) = \Theta(f(n) + g(n))$$

3.2.4 Other Asymptotic Notations

There are two more notations – o (small-oh) and ω (small-omega). These are used to represent not-tight upper and lower bounds of functions respectively.

Definition 5 $o(g(n)) = \{f(n) :$ for any constant $c > 0$, there exists another constant n_0 such that $0 \leq f(n) < c \cdot g(n),\ \forall n \geq n_0\}$

So, $2n^2 \neq o(n^2)$ but $2n = o(n^2)$. Intuitively we understand that if $f(n) = o(g(n))$, then $f(n)$ is insignificant compared to $g(n)$ when n approaches to larger values. That is,

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

.

Definition 6 $\omega(g(n)) = \{f(n) :$ for any constant $c > 0$, there exists another constant n_0 such that $0 \leq c \cdot g(n) < f(n),\ \forall n \geq n_0\}$

Hence, $n^2 = \omega(n)$ but $n \neq \omega(n^2)$. Whenever $f(n) = \omega(g(n))$, then we get

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

if limit exists.

Here the term **for any constant** is important. These notations are also not tight bound. They are used rarely in analysis of algorithms.

3.3 Benefits of Asymptotic Notation

Using asymptotic notation to find the complexity of an algorithm offers the following benefits:

1. Simplifies complexity expression; so, time and space complexity can be represented in a simplified manner.
2. Though it is approximation, but it can reflect the proper rate of growth of the functions.
3. We need not to develop code on a machine for finding the time and space complexity anymore; rather we shall be able to derive them directly in terms of asymptotic notation.

However, model of computation is not nullified by the use of asymptotic notation. Actually we always use the model of computation to find the time complexity. But, we do so in our mind to calculate approximate time.

Remark: Base of log does not matter under asymptotic notations. Suppose that $f(n) = O(g(n))$ and $g(n) = n \log_2 n$. Then $f(n) = O(n \log_2 n)$. However,

$$n \log_2 n = n. \log_2 b. \log_b n$$

for any $b > 0$. Here $\log_2 b$ is a constant. So we can write $f(n) = O(n \log_b n)$. Since b is any base, we avoid to mention base of log within asymptotic notations.

Chapter 4

Recurrence relation

Complexity can sometime be expressed as a recurrence relation. To express complexities in asymptotic notations, these recurrence relations are to be solved. For solving them, there are some standard methods:

1. Unrolling the recurrence,
2. Substitution method,
3. Recursion-tree based method, and
4. Master method

4.1 Unrolling a recurrence

Many times, the easiest way to solve a recurrence is to unroll it to get a summation. This method works fine for comparatively simple recurrence relation. Let us take following recurrence relation:

$$T(n) = \begin{cases} c & \text{if } \ n = 1 \\ cn + 2T(\frac{n}{2}) & \text{if } \ n \geq 2 \end{cases} \tag{4.1}$$

We can solve this relation by unrolling the recurrence step by step:

$$\begin{aligned} T(n) &= cn + 2T(\tfrac{n}{2}) \\ &= cn + 2[c\tfrac{n}{2} + 2T(\tfrac{n}{2^2})] \\ &= 2cn + 2^2T(\tfrac{n}{2^2}) \\ &= 3cn + 2^3T(\tfrac{n}{2^3}) \end{aligned}$$

$\quad\vdots$

$\quad = kcn + 2^k T(\frac{n}{2^k})$

Let us first assume that $n = 2^k$. This implies, $k = \log_2 n$.

$\therefore T(n) = cn\log_2 n + nT(1)$

$\quad = cn\log_2 n + cn$

$\quad = cn\log_2 n(1 + \frac{1}{\log_2 n})$

$\quad \leq 2cn\log_2 n \;\forall\; n \geq 2$

$\therefore T(n) = O(n\log n)$

Now if $2^{k-1} < n < 2^k$, then $T(n) < T(2^k)$, and so $T(n) = O(n\log n)$. Hence for any n, upper bound of $T(n)$ is $O(n\log n)$. Following similar argument, one can show that $T(n) = \Omega(n\log n)$.

4.2 Substitution method

In this method, a solution to the given recurrence relation is guessed. Then, it is shown by induction that the assumption is correct. However, there is no fixed way to guess the solution. If we fail to prove that a guess is correct, then we go for next (better) guess. For example, we might assume the solution of Recurrence 4.1 as $T(n) \leq cn$. We would then assume it holds true inductively for $n < n$ (the base case is obviously true) and plug in to our recurrence (using $n' = \frac{n}{2}$) to get:

$$\begin{aligned} T(n) &= cn + 2T(\tfrac{n}{2}) \\ &\leq cn + 2c.\tfrac{n}{2} \\ &= cn + cn \\ &= 2cn \end{aligned}$$

Obviously, this is not what we wanted: $T(n)$ is less or equal to $2cn$ but we guessed it as less or equal to cn. Hence, our guess is wrong.

Let's make now new guess: $T(n) \leq 2cn\log_2 n$. If we assume that our new guess holds inductively for $\frac{n}{2}$, then substituting into Relation 4.1 we get:

$$\begin{aligned} T(n) &= cn + 2T(\tfrac{n}{2}) \\ &\leq cn + 2.2c.\tfrac{n}{2}.\log_2\tfrac{n}{2} \\ &= cn + 2cn(\log_2 n - 1) \\ &= 2cn\log_2 n - cn \\ &\leq 2cn\log_2 n \quad [\because c > 0, n \geq 0] \end{aligned}$$

Hence, our guess is verified and we get that $T(n) = O(n\log n)$.

It is important in this type of proof to be very careful. For instance, one might think that our first guess is correct as c and $2c$ both are constants. But this is wrong!

Although this method is very powerful and mathematically sound, but there is no way to guess the correct solution.This is the challenge of using this method.

4.3 Recursion-tree based method

This is a standard method to solve a recurrence relation. Sometime this method is used to guess the solution, and then substitution method is used to confirm it.

In this method, a tree is formed considering non-recursive part as node of the tree. The root of the tree is the non-recursive elements of the original expression. If the expression has k number of recursive elements, then the root has k children, each of which represents one recursive element. Now, each child is further extended with same recurrence relation considering lesser n. In this way, tree is formed. Finally leaves contains the constant part.

To find out solution, we add the content of nodes of each level and then sum up the cost of all levels which give us solution to the recurrence relation. Consider Recurrence 4.1 to illustrate this method:

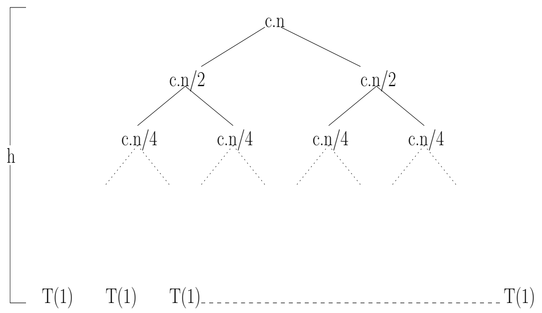


Figure 4.1: Recursion-Tree based method

Here we get a recurrence tree. Observe that summation of costs at each level is cn. Our next target is to find the hight (h) of the tree. If $n = 2^h$, then $h = \log_2 n$. But if $2^{h-1} < n < 2^h$, then $h = O(\log n)$. Hence, total cost of the tree is $h.c.n$, which implies that $T(n) = h.c.n. = O(n \log n)$.

As another example, consider that

$$T(n) \leq T(\frac{9n}{10}) + T(\frac{n}{10}) + c.n$$

. Let us now construct the recurrence tree against this relation.

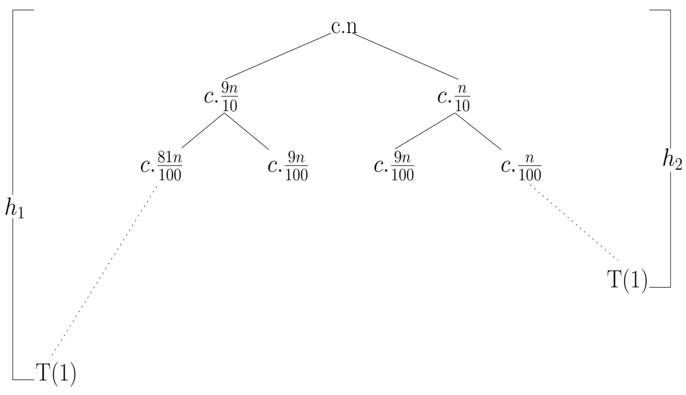


Figure 4.2: Recursion-Tree based method

This recurrence tree always remains unbalanced. One part of the tree reaches to the leaves faster than other part. In the above tree, h_1 is the highest number of levels from the root, whereas h_2 is smallest number of levels from root to leaves. Up to h_2 levels, summation of cost of each level is cn. Here

$h_1 = \log_{\frac{10}{9}} n$ and $h_2 = \log_{10} n$.

$\therefore T(n) \leq c.n.h_1 = c.n. \log_{\frac{10}{9}} n$. This implies that $T(n) = O(n \log n)$

Similarly, we can write $T(n) \geq c.n.h_2 = c.n. \log_{10} n$, which gives us $T(n) = \Omega(n \log n)$.

4.4 Master theorem

The Master Theorem provides us ready-made solutions to the following type of recurrence relations:

$$T(n) = aT(\frac{n}{b}) + f(b)$$

where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function.

Theorem 2 : *For the recurrence $T(n) = aT(\frac{n}{b}) + f(n)$ with $a \geq 1, b > 1$, $T(n)$ is asymptotically bounded as follows.*

1. *If $f(n) = O(n^{\log_b a-\epsilon})$ for some constants $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.*
2. *If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$.*
3. *If $f(n) = \Omega(n^{\log_b a+\epsilon})$ for some constants $\epsilon > 0$, and if $af(\frac{n}{b}) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.*

For Recurrence 4.1, $a = b = 2$ and $f(n) = cn$. We can use the master theorem to solve it, and here $f(n) = \Theta(n^{\log_b a})$. So $T(n) = \Theta(n \log n)$. Following is a corollary which we directly get from the above theorem.

Corollary 1 *The recurrence $T(n) = aT(\frac{n}{b}) + cnk$ where $a \geq 1, b > 1, c > 0, k \geq 0$ are constants solves to*

1. *$T(n) = \Theta(n^k)$ if $a < b^k$*
2. *$T(n) = \Theta(n^k \log n)$ if $a = b^k$*
3. *$T(n) = \Theta(n^{\log_b a})$ if $a > b^k$*

It is always advised to use master theorem that if the given recurrence relation is like $T(n) = aT(\frac{n}{b}) + f(n)$ and follows the stated conditions.

Chapter 5

Searching Problem - continued

Let us re-look at the Searching Problem. An algorithm, named Sequential Search Algorithm, for the problem has been developed. We have shown that worst case time complexity of the algorithm is $\Theta(n)$ (see page 16 and page 17). Although other algorithms, such as *Binary Search* have been developed which run much faster than Sequential Search, we cannot avoid need of this algorithm in several applications due to several reasons. Here question is, can we make the sequential search faster? Answer is Yes!

5.1 Quick Sequential Search

Let us re-look at Sequential Search Algorithm (page 3). There are some types of operations such as assignment (Step 1), increment (Step 3), comparison (Step 2 and Step 4), jump (Step 4). If we consider any standard computer architecture, we can see that assignment, increment, jump are comparatively less time consuming operation than the comparison. Let us recall the expression for time complexity in the worst case (see page 17).

$$T(n) = c_1 + (c_2 + c_3 + c_4)n$$

As c_1 is small and executed only once in the algorithm, we can write

$$T(n) \approx (c_2 + c_3 + c_4)n$$

Further, $c_2 \approx c_4$ as both are primarily comparison. So,

$$T(n) \approx (2c_2 + c_3)n$$

That is, major contribution to the time complexity is from comparison operation. We can see that in each iteration of the algorithm, there are two comparisons. In general, if we can reduce the number of instructions to be executed, an algorithm becomes faster. Here, if we can reduce the number of comparisons in worst case, we can improve the time complexity. Let us target to do that.

Let us insert the search key at the end of the list. Then, the comparison of Step 2 will be successful for at least once, and consequently we can bring the Step 4 outside of the loop. Following is the algorithm.

Algorithm 2 ***QuickSSearch***
Input: *A list of records* $R_1, R_2, \cdots, R_n$ *identified by keys* $K_1, K_2, \cdots, K_n$ *respectively;* K *(Search Key).*
Output: *'YES' if the record exists; 'NO' otherwise.*

```
Step 1: $i \leftarrow 1$, $K_{n+1} \leftarrow K$
Step 2: If $K = K_i$, go to Step 4.
Step 3: $i \leftarrow i + 1$; go to Step 2.
Step 4: If $i \leq n$, output 'Yes'; Otherwise output 'No'.
```

In this algorithm, observe that Steps 2 and 3 are repeatedly executed and there is only one comparison. In worst case, Step 2 is executed $n + 1$ times and Step 3 n times.Whereas in the original algorithm (page 3), Step 2 and Step 4 are executed n times. If we consider only comparison operation then $2n$ comparisons are needed in original algorithm, but its quick version demands only $n + 1$ comparisons. So, from number of comparisons point of view, this new algorithm is two times faster than the original!

5.2 Failure of Asymptotic Notations

Although asymptotic notations are very useful to express complexity of an algorithm, complexity with asymptotic notations does not always reflect the accurate performance of the algorithm. If we find out time complexity of Algorithm QuickSSearch (Algorithm 2), we can easily see that it is $\Theta(n)$. Time complexity of Algorithm SSearch (Algorithm 1) is also $\Theta(n)$. Hence, performance of both the algorithms is asymptotically same. But are they same?

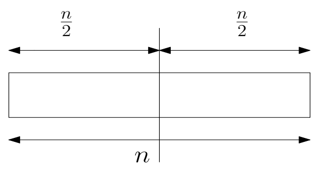
Previous discussion reveals that Algorithm QuickSSearch is much better than Algorithm SSearch. Therefore, when performance of two algorithms are asymptotically same, then which one is really better cannot be understood from their asymptotic complexity. In that case, performance of an algorithms on some particular machine are to be analyzed. That is, programs against those algorithms are to be carefully developed and with more care, time (and space) requirement of those programs are to be analyzed. This analysis is particularly needed when we develop real-life software.

It is also to be noted here that architecture of computing machine is very important while we analyze the performance of algorithms. Instruction set, performance of individual instructions, configuration of the machine are to be considered for more accurate analysis of algorithms.

5.3 Binary Search

Let us now consider that the input list is sorted. That is, if $K_1, K_2, \cdots, K_n$ are keys of n consecutive records, then $K_1 \leq K_2 \leq \cdots \leq K_n$. Under this constraint, we can develop a new algorithm, named *Binary Search Algorithm* which is asymptotically better than Sequential Search Algorithm.

For binary search, we divide the problem of size n in two sub-problems of size about $\frac{n}{2}$ each. We check if the middle record is the target record. If not and if the search key is less than middle key, we search for the key in left sub-problem. Otherwise, we search it in the right sub-problem. And then we repeat the same process until the record is found or the sub-problem size becomes zero.



The approach taken here is called *Divide-and-Conquer*, which is a classical approach of developing an algorithm.

Divide-and-Conquer Method: This method says that to solve a problem, divide it to get sub-problems. Then divide the sub-problems to get subsub-problems. This process is repeated until we get sufficiently small problems which can easily be solved. Then combine the smaller solutions to get the final solution of the problem.

In science, this approach is known as *Cartesian* approach. René Descartes, a French philosopher of sixteenth century and mathematician, advocated this approach for scientific discovery. This approach contributed a lot in the development of modern science. A good example of scientific discovery using this approach is Newton's Principia. Although this approach has been challenged in recent times after the advent of Chaos theory, it is still dominating way of scientific investigation.

Let us now write down the binary search algorithm.

Algorithm 3 ***BSearch***

```
Input: A list of records $R_1, R_2, \cdots R_n$ identified by keys $K_1, K_2, \cdots, K_n$
respectively where $K_1 \leq K_2 \leq \cdots \leq K_n$; $K$ (Search Key).
Output: 'YES' if the record against exists and 'NO' otherwise.
Step 1: $l \leftarrow 1, u \leftarrow n$.
Step 2: If $l > u$ then output 'NO' and exit; otherwise $mid \leftarrow \lfloor \frac{l+u}{2} \rfloor$.
Step 3: If $K = K_{mid}$ then output 'YES' and exit.
        If $K < K_{mid}$ then goto Step 4.
        Otherwise go to Step 5.
Step 4: $u \leftarrow mid - 1$; go to Step 2.
Step 5: $l \leftarrow mid + 1$; go to Step 2.
```

We next analyze the time complexity of the algorithm. Let us assume the following.

Step 1 takes c_1 time.
Step 2 takes c_2 time.
Step 3 takes c_3 time.
Step 4 takes c_4 time.
Step 5 takes c_5 time.

Here $c_4 = c_5$ since Step 4 and Step 5 contain similar operation. Step 2, Step 3 and Step 4/Step 5 are executed repeatedly (recursively). Like previous algorithm, the worst case for this algorithm is that the search key does not exist in the list. Assume that Step 2, Step 3 and Step 4/Step 5 are executed k times in worst case. Hence, time complexity of this algorithm is

$$T(n) = c_1 + (c_2 + c_3 + c_4)k$$

This k depends on n. We can also express $T(n)$ by recurrence relation.

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ c + T(\frac{n}{2}) & \text{if } n \geq 2 \end{cases}$$

Here, c is a constant that represents the time required for execution of single pair of (l, u).

This recurrence relation can be solved simply by unrolling it.

$$\begin{aligned} T(n) &= c + T(\tfrac{n}{2}) \\ &= c + [c + T\tfrac{n}{2^2}] \\ &= 2c + T(\tfrac{n}{2^2}) \\ &= 3c + T(\tfrac{n}{2^3}) \\ &\vdots \\ &= kc + T(\tfrac{n}{2^k}) \end{aligned}$$

Let $n = 2^k$, which implies, $k = \log_2 n$. So, we get

$$\begin{aligned} T(n) &= c.\log_2 n + T(1) \\ &= c.log_2 n + c \qquad [\because T(1) = c] \\ &= c.\log_2 n.(1 + \tfrac{1}{\log_2 n}) \leq 2c \log_2 n \quad \forall\ n \geq 2 \end{aligned}$$

$\therefore T(n) = O(\log n)$

Now if $2^{k-1} < n < 2^k$, then also one can show that $T(n) = O(\log n)$.

Note: The base of log does not matter under asymptotic notation. Change of base actually affects an expression by constant term. Since constant terms are avoided in asymptotic notation, we can avoid mentioning of base of log. So, we can write for the above $T(n) = O(\log n)$

We can easily find out number of comparisons needed in this algorithm for successful and unsuccessful search.

Theorem 3 : *If $2^{k-1} \leq n < 2^k$, a successful search of Algorithm 3 requires $(\min 1, \max k)$ comparisons. If $n = 2^k - 1$, an unsuccessful search requires k comparisons; and if $2^{k-1} \leq n < 2^k - 1$, an unsuccessful search requires either $kl - 1$ or k comparisons.*

5.4 Successful search: comparison

Performance of the above two algorithms are measured for the worst case, when the search key is absent in the list. If the search is successful, that is, if the search key is present in the list, then which of the above two performs better?

To answer this question, we have to know that after how many key comparisons, the target key is reached. And in that case, it cannot be blindly said that binary search is better than sequential search! Although we have $T_{Binary} = \Theta(\log n)$ and $T_{Seq} = \Theta(n)$ in worst case, sometime sequential search can perform better in successful search case.

Let us consider that p_i is the probability of accessing the i^{th} record. Obviously, in successful case,

$$p_1 + p_2 + \cdots + p_n = 1$$

Then the expected number of comparisons required is

$$\overline{C_n} = p_1 + 2p_2 + \cdots + np_n \text{ (for a successful search)}$$

Now if we assume that $p_1 = p_2 = \cdots = p_n = \frac{1}{n}$, then $\overline{C_n} = \frac{n+1}{2}$. This is not very realistic assumption. Probability of access of records generally differs. Let us now organize the records in a different way. Suppose that the most frequently searched record is placed in first location, the second most frequently searched record is placed in second location, and so on. Hence here $p_1 > p_2 > \cdots > p_n$. Under this assumption sequential search can perform very well.
Let us hypothetically assume that $p_1 = \frac{1}{2}$, $p_2 = \frac{1}{4}$, $\cdots p_{n-1} = \frac{1}{2^{n-1}}, p_n = \frac{1}{2^{n-1}}$, which satisfy the following: $p_1 + p_2 + \cdots + p_n = 1$. Then,

$$\begin{aligned}
\overline{C_n} &= \tfrac{1}{2} + 2.\tfrac{1}{2^2} + 3.\tfrac{1}{2^3} + \cdots + (n-1)\tfrac{1}{2^{n-1}} + n.\tfrac{1}{2^{n-1}} \\
&= (\tfrac{1}{2} + \tfrac{1}{2^2} + \tfrac{1}{2^3} + \cdots + \tfrac{1}{2^{n-1}} + \tfrac{1}{2^{n-1}}) + (\tfrac{1}{2^2} + 2.\tfrac{1}{2^3} + \cdots + \tfrac{n-2}{2^{n-1}} + \tfrac{n-1}{2^{n-1}}) \\
&= 1 + \tfrac{1}{2}(\tfrac{1}{2} + 2.\tfrac{1}{2^2} + \cdots + (n-2).\tfrac{1}{2^{n-2}} + \tfrac{n-1}{2^{n-2}}) \\
&= 1 + \tfrac{1}{2}(\overline{C_n} - \tfrac{1}{2^{n-1}}) \\
2\overline{C_n} &= 2 + \overline{C_n} - \tfrac{1}{2^{n-1}} \Rightarrow \overline{C_n} = 2 - \tfrac{1}{2^{n-1}}
\end{aligned}$$

That is, expected number of comparison is constant! Hence its performance on average is very good.This discussion indicates that to understand the performance of algorithms on an average, we need to know the distribution of the input data. For the development of real-life software this is very important.

5.5 Fibonacci Search

Fibonacci search is an alternative to binary search. Here also we assume that records are sorted in increasing order. The algorithm implicitly form *Fibonacci tree* for searching records. The algorithm uses only addition and subtraction (but no division), which may be advantageous to some systems.

A Fibonacci tree of order k has $F_{k+1} - 1$ internal nodes (circle) and F_{k+1} external nodes (box). Figure 5.1 is a Fibonacci tree of order 6. Following is the method of constructing the tree.

> If $k = 0$ or $k = 1$, the tree is simply $\boxed{0}$.
>
> If $k \geq 2$, the root is F_k; the left subtree is the Fibonacci tree of order $k - 1$; and the right subtree is the Fibonacci tree of order $k - 2$ with all numbers increased by F_k.

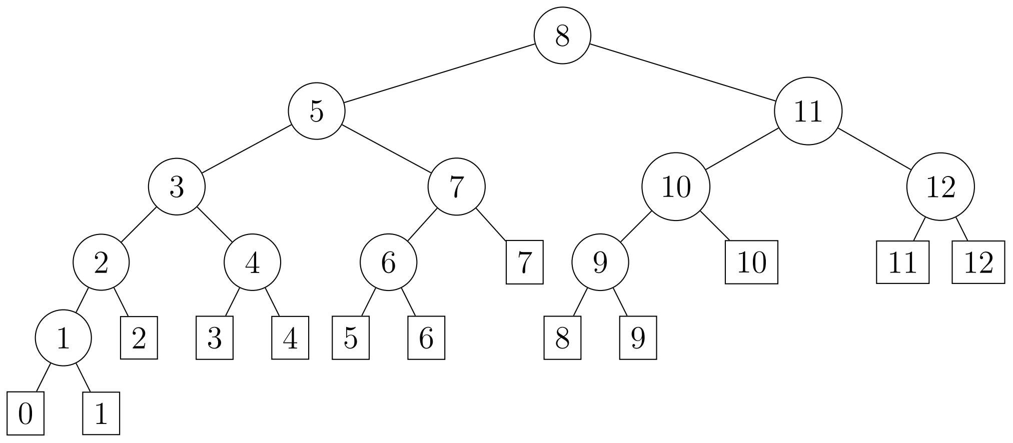


Figure 5.1: Fibonacci tree of order 6

Except the external nodes, the numbers of two children of each internal node differ from their parent's number by the same amount, and this amount is a Fibonacci number. For example, $5 = 8 - F_4$ and $11 = 8 + F_4$ in Figure 5.1. While the difference is F_j, the corresponding Fibonacci difference for the next brach on the left is F_{j-1}, while on the right it skips down to F_{j-2}. For example, $3 = 5 - F_3$ while $10 = 11 - F_2$.

Let us now combine these observations to reach to the following algorithm for recognising the external nodes. Here we have assumed that $n + 1$ is a Fibonacci number.

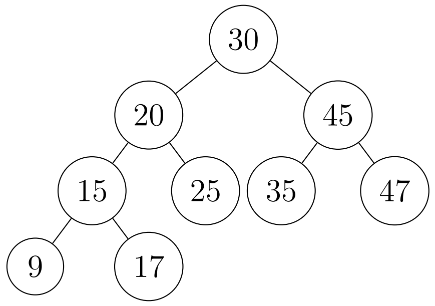
```
Algorithm 4 FibonacciSearch
Input: A list of records $R_1, R_2, \cdots R_n$ identified by keys $K_1, K_2, \cdots, K_n$ respectively where $K_1 \leq K_2 \leq \cdots \leq K_n$; $K$ (Search Key).
Output: 'YES' if the record against exists and 'NO' otherwise.
Step 1: $i \leftarrow F_k, p \leftarrow F_{k-1}, q \leftarrow F_{k-2}$ (Here p and q are consecutive Fibonacci numbers)
Step 2: If $K = K_i$ then output 'YES' and exit.
        If $K < K_i$, go to Step 3; otherwise go to Step 4.
Step 3: If $q = 0$, then output 'NO' and exit; otherwise set $(i, p, q) \leftarrow (i - q, q, p - q)$; then return to Step 2.
Step 4: If $p = 1$, then output 'NO' and exit; otherwise set $i \leftarrow i + q, p \leftarrow p - q$ and then $q \leftarrow q - p$; and return to Step 2.
```

Like Binary Search, this algorithm follows divide-and-conquer strategy and its worst case time complexity is also $O(\log n)$. However, unlike Binary Search, the input list is divided into unequal parts.

5.6 Binary Search Tree

Binary Search and Fibonacci Search use an implicit binary tree structure to efficiently search an element from the given list of records. Both the algorithms, however, demand a sorted list as input. These algorithms are appropriate mainly for fixed-size lists. If the records change dynamically, then we need to maintain the list sorted every time, which is generally expensive.

The *Binary Search Trees* (BSTs) are *explicitly* binary trees that allow to search elements efficiently like before, and additionally makes it possible to easily insert and delete records from the tree. BSTs are formed dynamically.If an element is not present in the tree, the element is inserted into it. Let us consider following sequence of keys as input: 30, 20, 45, 47, 15, 9, 25, 17, 35. As 30 is the first key, it forms the root of the tree. Next 20 is searched in left side of the tree. But it is not in tree, so inserted as left child of the root. Similarly 45 is unsuccessfully searched, and inserted as right child. And so on. Following is the structure. Observe that the tree is growing dynamically.



We exclude the details of BST algorithms, but can derive its complexity from the above discussion. Searching complexity depends here on the structure of BST and the search key. If the BST is almost complete, then there are $O(\log n)$ levels. In that case, searching complexity is $O(\log n)$. In fact, here the average case complexity is $O(\log n)$. However, worst case scenario is different. Suppose the input list is sorted. Then the BST becomes a linked list. So searching in the linked list is $O(n)$ when the search key is absent. In fact, the searching then becomes sequential search.

To avoid this worst case scenario, *balanced trees* have been introduced. In case of balanced trees, the leaves are almost at same level. If this kind of search trees can be developed, then the searching complexity is $O(\log n)$, even in worst case. *AVL* trees are such balanced trees. We also omit here details of AVL tree formation. However, we shall come back to the balanced trees when we shall discuss about *external searching* (see page 65).

Chapter 6

Sorting Problem

Sorting Problem is a class of *Permutation Problem*. In a permutation problem, one has to find a permutation of input data so that some given condition is satisfied. In case of sorting problem, a list of records $R_1, R_2, \cdots, R_n$, identified by keys $K_1, K_2, \cdots, K_n$ respectively is given. Then, one has to find a permutation Π such that $K_{\Pi(1)} \leq K_{\Pi(2)} \leq \cdots K_{\Pi(n)}$. Following is an example permutation of 4 elements.

$$\Pi = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 1 & 3 & 4 \end{pmatrix}$$

Here $Pi(1) = 2, \Pi(2) = 1, \Pi(3) = 3, \Pi(4) = 4$. Permutations are bijective functions. If there are n elements then the total number of permutations is $n!$. Permutations problems are very difficult to solve, because if we are asked to find a particular permutation from a huge pool of possible permutations, it will take exponential time in general. However, the good news regarding this is that we have faster algorithms for solving Sorting Problem.

Sorting problem does not find only the correct permutation, but also reorganizes the records according to the desired permutation. Following is the problem statement:

Problem Statement: Given a list of records $R_1, R_2, \cdots, R_n$, identified by keys $K_1, K_2, \cdots, K_n$ respectively, rearrange the records as $R'_1, R'_2, \cdots, R'_n$ such that $K'_1 \leq K'_2 \leq \cdots \leq K_n$.

That is, the records are to be sorted in *ascending order*. Records can also be sorted in *descending order*. By sorting, however, we shall mean here sorting in ascending order.

We classify the sorting techniques based on their way of functioning as following:

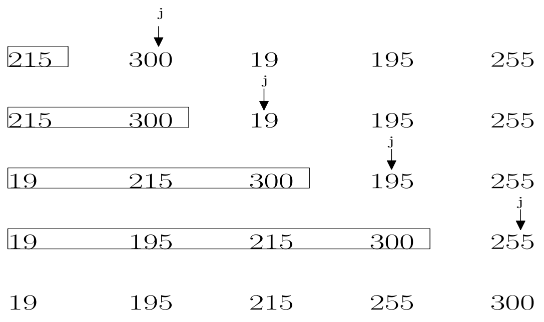
1. Sort by Insertion
2. Sort by Exchange
3. Sort by Selection
4. Sort by Merging
5. Sort by Special Purpose Technique.

6.1 Sort by Insertion

Insertion of a record in its relatively right position is the primary operation in this class of technique.

6.1.1 Straight Insertion Sort

Straight insertion sort (or, simply "Insertion sort") follows the rule of playing cards. Initially the first record is considered as sorted sublist. The next record is placed in proper position of the sublist so that it remains sorted. This procedure is repeated until all records are covered. Let j be used to point the element that will be inserted into the sorted sublist. Another variable i is used to find the proper location where K_j is to be inserted. Following is an example.



Algorithm 5 StraightInsertionSort

Algorithm: SISort

Input: *A list of records $R_1, R_2, \cdots, R_n$ identified by keys $K_1, K_2, \cdots, K_n$*

respectively.
Output: *Records are in n on-decreasing order.*

```
Step 1: Repeat Step 2 to Step 5 for $j = 2, 3, \cdots, n$.
Step 2: $i \leftarrow j-1$,$K \leftarrow K_j$,$R \leftarrow R_j$.
Step 3: If $K \geq K_j$ go to Step 5.
Step 4: $K_i \leftarrow K_{i-1}$,$R_i \leftarrow R_{i-1}$,$i \leftarrow i-1$. If $i > 0$, then go to Step 3.
Step 5: $K_i \leftarrow K$,$R_i \leftarrow R$ .
```

Time Complexity: Step 1 has increment and assignment. Let Step 1 costs c_1. Likewise, let Step 2 costs c_2, Step 3 costs c_3, Step 4 costs c_4 and Step 5 costs c_5.
Best case: Best case of the algorithm is, the list is sorted in ascending order. Then, Step 4 is not executed. So the time complexity is

$$T_{Best}(n) = c_1 + (c_2 + c_3 + c_5).(n-1)$$

which implies $T_{Best}(n) = \Theta(n)$.
Worst case: In this algorithm the worst case occurs when the output comes in the reverse order of the given input i.e. if the input is in descending order but we want the output in ascending order. In the worst case,
for $j = 2$, the inner loop will be executed once. So the cost will be $(c_3 + c_4)$.
for $j = 3$, the inner loop will be executed twice. So the cost will be $2(c_3+c_4)$.
for $j = 4$, the inner loop will be executed thrice. So the cost will be $3(c_3+c_4)$ and so on.
So, the total cost for the execution of the inner loop will be $(c_3+c_4)+2(c_3+c_4)+\cdots+(n-1)(c_3+c_4)$
For each j, Step 2 and Step 5 will be executed. And at very first Step 1 will be executed once. Hence the total cost is $T_{Worst}(n) = (c_3+c_4)+(c_2+c_5)+2(c_3+c_4)+(c_2+c_5)+\cdots+(n-1)(c_3+c_4)+(c_2+c_5)+c_1$
$= (c_3+c_4)\frac{n.(n-1)}{2} + (n-1)(c_2+c_5) + c_1$
Clearly, it is $T_{Worst}(n) = \Theta(n^2)$.
Space complexity: Only few variables are required here, and the requirement is fixed. So, the space complexity is $O(1)$.

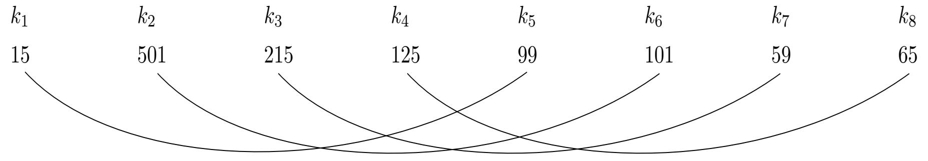
Analysis of the sorting algorithms becomes simpler if we can count only the number of key comparisons in sorting algorithm. In a general purpose sorting algorithm, comparisons of keys are the basic operations of ordering the input records.

Theorem 4 : *If a comparison-based sorting algorithm takes k comparisons to sort the input list, then the time complexity of the algorithm is $\Theta(k)$.*

Using this theorem, we can find out time complexity of the above algorithm. Here number of key comparisons in worst case is $1+2+\cdots+n-1 = \frac{n(n-1)}{2}$. Hence, $T(n) = \Theta(n^2)$.

6.1.2 Shell Sort

This sorting technique was proposed by Donald L. Shell in 1959. In this technique, sublists of 2 elements are first formed. Consider the following figure with 8 elements. We first form 4 groups.



If any group is out of order, the next element of the group is inserted in the first position. As next step, we form sublist/group of 4 elements, by inserting previously sorted one sublist into another. The list is inserted in such a way that the final list remains sorted. This process is repeated until we get a single sorted list.

We group the elements to proceed in the following way.
(K_1K_5) (K_2K_6) (K_3K_7) (K_4K_8) – in first pass
$(K_1K_5K_3K_7)$ $(K_2K_4K_6K_8)$ – in second pass
$(K_1K_2K_3K_4K_5K_6K_7K_8)$ – in third pass

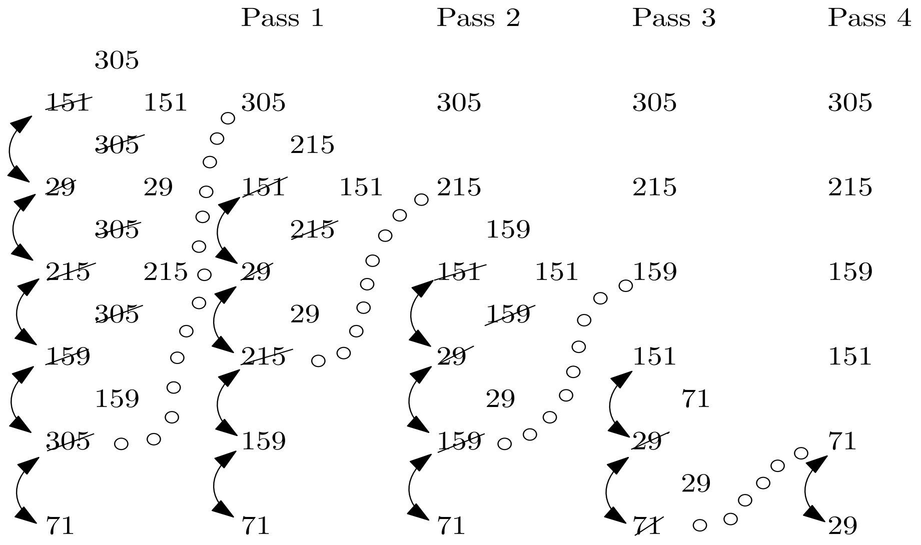
Although its analysis is very difficult and its performance is yet to be completely understood, it runs faster than Straight Insertion Sort.

6.2 Sort by Exchange

In this family of algorithms, if it is found that two records, when compared, are out of order, they are swapped. Here “exchange” of location of records is the primary operation to sort the list.

6.2.1 Bubble Sort

The concept of this method is that the larger element will “bubble up” to their proper position and when there is no bubble, the process will be terminated. During this process, a larger element *exchange* its position with its immediate smaller element. Following example illustrates the process. We start comparison from lowest two elements. If they are out of order, we interchange their position.Then we compare next two elements and repeat the process. After one *pass*, the largest element occupies ("bubble up" to) the topmost position, which is last position of the given list.



Algorithm 6 ***BubbleSort***

```
Input: Records $R_1, R_2, \cdots R_n$ are given with keys $K_1, K_2, \cdots, K_n$.
Output: Records are sorted in ascending order based on their keys.
Step 1: $BOUND \leftarrow n$.
Step 2: $t \leftarrow 0$. Perform Step 3 for $i = 1, 2, \cdots, BOUND - 1$.
Step 3: If $K_i > K_{i+1}$ then $R_i \leftrightarrow R_{i+1}$ and $t \leftarrow i$.
Step 4: If $t = 0$, terminate the algorithm.
        Otherwise $BOUND \leftarrow t$ and go to Step 2.
```

Time Complexity: We shall count number of key comparisons only to find time complexity of the algorithm (Theorem 4).

Best case: The best case is that the input list is sorted in ascending order. Then, number of comparison performed by the above algorithm is $n-1$ only. So, $T_{Best}(n) = \Theta(n)$.

Worst case: The worst case occurs when the input is sorted n descending order but we want the output in ascending order. In this case, the number of comparisons is $(n-1)+(n-2)+\cdots+1 = \frac{n(n-1)}{2}$. Hence, $T_{Worst}(n) = \Theta(n^2)$

Chapter 7

Quick Sort

We put the Quick Sort algorithm under the family of "Sort by Exchange". The idea behind Quick Sort is the following:

- Pick up an element K_p as Pivot Point/Element from the set of keys $K_1, K_2, \cdots, K_n$.
- Partition the given list such that all the elements to the left sublist is less than (or equal to) K_p and all the elements of right sublist is greater than K_p. Hence, K_p gets its final position after partitioning.
- Repeat the procedure for both the sublist. This process is continued until the whole list gets sorted.

One may find that the functioning of Quick Sort is very similar to the Divide-and-Conquer method. But in Quick Sort we make some intelligent division, whereas in Divide-and-Conquer method the divisions are made blindly to get smaller sub-problems. We shall see that the Merge Sort follow Divide-and-Conquer approach. So we love to put this sorting algorithm under Sort by Exchange family. One point can be said here that this division of family is not very clear-cut. For example, in Straight Insertion Sort we do the exchange operation to insert an element in its proper place.

Although any element can be chosen as pivot point, we classically choose the first element as pivot in our algorithm.

7.1 Steps of algorithm

Let us now take an example to show the working principle of Quick Sort. Following are the elements (keys) which are to be sorted.

$$512, 139, 612, 52, 739, 239, 550, 600, 700$$

Here, 512 (the first element) is chosen as the pivot element. Now, we use two pointers i and j, which point to the elements of left and right sublists respectively. Initially, i assumes the value 2 whereas j assumes the value n, and then i increases but j decreases. If $K_i \leq K_p$, i increases; otherwise, it stops. If $K_j > K_p$, j decreases; otherwise it stops. Then interchange K_i and K_j (R_i and R_j) if $i < j$, and then repeat the process until $i < j$. In this example, when
$i = 2$, $139 < 512$ i.e. 139 is to be in left sublist, but when
$i = 3$, $612 \not< 512$ i.e. 612 is to be in right sublist. So, i does not increase.
For the other pointer j, when
$j = 9$, $512 < 700$ i.e. it is to be in right sublist.
$j = 8$, $512 < 600$ i.e. it is to be in right sublist.
$j = 7$, $512 < 550$ i.e. it is to be in right sublist, but when
$j = 6$, $512 \not< 239$ i.e. the loop will stop here.

The next step is to interchange K_3 and K_6 (R_3 and R_6). In this case, we interchange 612 and 239. So the above list becomes

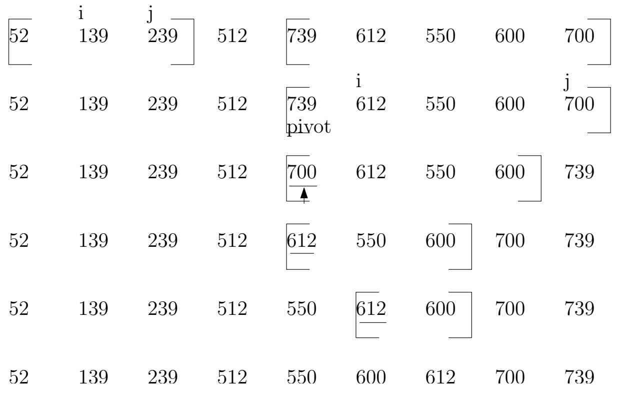
$$512, 139, \underline{239}, 52, 739, \underline{612}, 550, 600, 700$$

Now, the pointer i resumes its functionality. So, for
$i = 4$, $52 < 512$, so 52 is to be in left sublist, but for
$i = 5$, $739 \not< 512$. The the loop stops here. Now pointer j resumes its function. For $j = 5$, $512 < 739$ i.e. 739 is to be in right sublist.
Now, when j decreases, i becomes greater than j. This is the condition of coming out of loop for partitioning. As a next step, K_i and K_p are interchanged. That is, 52 and 512 are interchanged. And, the first pass gets completed. The result is

$$[52, 139, 239]\ 512\ [739, 612, 550, 600, 700]$$

This is the final position of 512 and here two sublists are formed. We next proceed with a sublist, and the other is pushed in a stack.

Here the question is that which sublist should we choose first? To reduce Space Complexity we proceed with smaller sublist and put the larger in the stake. In this case, the left sublist is processed first and the right sublist is pushed in stack. Hence we get the following during execution



Algorithm 7 ***QuickSort***

```
Input: Records $R_1, R_2, \cdots R_n$ are given with keys $K_1, K_2, \cdots, K_n$.
Output: Records are sorted in ascending order based on their keys.
Step 1: $l \leftarrow 1, r \leftarrow n$, and set the stack empty.
Step 2: $i \leftarrow l, j \leftarrow r+1, K \leftarrow K_l$.
Step 3: $i \leftarrow i+1$. If $K_i < K$ then repeat this step.
Step 4: $j \leftarrow j-1$. If $K < K_j$ then repeat this step.
Step 5: If $j \leq i$, interchange $R_l \leftrightarrow R_j$ and then go to Step 7.
Step 6: Interchange $R_i \leftrightarrow R_j$ and go back to Step 3.
Step 7: If $r-j \geq j-l > 1$, push $(j+1, r)$ on the top of the stack and go to Step 2.
        If $j-l > r-j > 1$, push $(l, j-1)$ on the top of the stack and go to Step 2.
Step 8: If the stack is non-empty, pop its top entry $(l', r')$ and then $(l, r) \leftarrow (l', r')$; return to Step 2.
```

7.2 Worst and best case

In general, the worst case with respect to time complexity arises if the pivot is chosen in such a way that all elements in one sublist and the other sublist is empty. For this algorithm the worst case for time complexity is obtained when we are given a sorted list. Then in each step there is a completely unbalanced sublist. The pivot element will remain at one end of the sublist. For the first element the loop will run for $n-1$ times, for the second element the loop will run for $n-2$ times and so on. Hence the number of comparisons in worst case is $(n-1)+(n-2)+...+1 = \frac{n(n-1)}{2}$

Hence, $T_{worst} = \Theta(n^2)$

This worst case time complexity can also be expressed as: $T_{worst}(n) \leq T_{worst}(n-1) + c.n$

The best case for time complexity in Quick Sort is when we get completely balanced partition for each step. Then best case time complexity is obtained by $T_{best} \leq T_{best}(\frac{n}{2}) + c.n$

$$\text{i.e. } T_{best} = \Theta(n \log n)$$

However, worst case for space complexity does not match with worst case for time complexity. In fact, the worst case for space complexity is the best case for time complexity, and the best case for space complexity is worst case for time complexity. For space complexity the worst case occurs when we get completely balanced partition for each step and the best case occurs when there is sorted list in each step. Then the pivot element will be at one end of the sublist and we have nothing to put in the stack. Hence we get $S_{worst}(n) \leq S_{worst}(\frac{n}{2}) + c \Rightarrow S_{worst}(n) = \Theta(\log n)$

7.3 Average case analysis

Quick Sort is really quicker! It performs better than other algorithms in many cases. But this is not reflected in its worst case analysis. The worst case time complexity is $T_{worst}(n) = \Theta(n^2)$ and the best case time complexity is $T_{best} = \Theta(n \log n)$. If the partitioning of list is unbalanced, the performance of Quick Sort may get worse. Let us consider a bad partitioning where about 90% of elements are always in one side and the rest 10% is in other side. Then the time complexity can be expressed as

$$T(n) \leq T(\frac{9n}{10}) + T(\frac{n}{10}) + c.n$$

This recurrence relation also results in the following solution: $T(n) = O(n \log n)$. Therefore, a very bad partitioning like above also demands only $O(n \log n)$ time. Although for many algorithms, average case is asymptotically as bad as the worst case, for Quick Sort the scenario can be different. So we go for average case analysis. We discuss here two (actually three) methods for the average case analysis.

Method 1:

Let us consider all the inputs are equally probable. Though it is not very realistic assumption, we do it to simplify the analysis. Let us now develop Randomized Quick Sort algorithm where pivot point is chosen uniformly at random. Expected time required by this algorithm is the time requirement in average case as expectation gives the mean value.

Observe that the algorithm spends most of its time in partitioning the list (Step 2 to Step 6). The goodness of the algorithm depends on the partition scheme. So our claim is the number of key comparisons during partitioning determines the time complexity of the algorithm. In fact, the keys are compared only during partitioning of list.

Let X be a random variable which notes the total number of keys comparisons. Then the time complexity is $\Theta(X)$. Let us denote $K_{ij} = \{K_i, K_{i+1}, \cdots, K_j\}$ as a sublist of the given list. How many times two keys can be compared? – At most once if both of them are in the same sublist and one of them is a pivot element.

Let X_{ij} be the indicator random variable which indicates whether two arbitrary keys K_i and K_j $(i \neq j)$ are compared or not. Clearly,

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$$

Now, $E[X] = E[\sum\limits_{i=1}^{n-1} \sum\limits_{j=i+1}^{n} X_{ij}]$
$\quad = \sum\limits_{i=1}^{n-1} \sum\limits_{j=i+1}^{n} E[X_{ij}]$ [by the linearity of expectation]
Let A be an event and I be its indicator random variable. Then

$$I(A) = \begin{cases} 1 & \text{if A occurs} \\ 0 & \text{otherwise} \end{cases}$$

Then $E[I] = 1.Pr\{A\} + 0.Pr\{A\} = Pr\{A\}$.
In our case, $E[X_{ij}] = Pr\{K_i \textit{ is compared with } K_j\}$. Hence,
$E[X] = \sum\limits_{i=1}^{n-1} \sum\limits_{j=i+1}^{n} Pr\{K_i \textit{ is compared with } K_j\}$
Now, $Pr\{K_i \textit{ is compared with } K_j\}$
$\quad = Pr\{K_i \textit{ or } K_j \textit{ is chosen as pivot}\}$

$= Pr\{K_i \textit{ is chosen as pivot}\} + Pr\{K_j \textit{ is chosen as pivot}\}$
$= \frac{1}{j-i+1} + \frac{1}{j-i+1} = \frac{2}{j-i+1}$ [Since the random variable follows uniform distribution, choosing of i is independent from choosing of j and there are $j-i+1$ elements in the sublist k_{ij}]

Hence , $E[X] = \sum\limits_{i=1}^{n-1} \sum\limits_{j=i+1}^{n} \frac{2}{j-i+1}$

$= \sum\limits_{i=1}^{n-1} \sum\limits_{k=2}^{n-i+1} \frac{2}{k} < \sum\limits_{i=1}^{n-1} \sum\limits_{k=1}^{n} \frac{2}{k}$

Here $H_n = \sum\limits_{k=1}^{n} \frac{1}{k} = 1 + \frac{1}{2} + \cdots + \frac{1}{n}$ is the harmonic series, and we get that

$\sum\limits_{k=1}^{n} \frac{1}{k} \leq \sum\limits_{i=0}^{\lfloor \log_2 n \rfloor} \sum\limits_{j=0}^{2^i-1} \frac{1}{2^i+j}$

$\leq \sum\limits_{i=0}^{\lfloor \log_2 n \rfloor} \sum\limits_{j=0}^{2^i-1} \frac{1}{2^i}$

$\leq \sum\limits_{i=0}^{\lfloor \log_2 n \rfloor} 1 \leq \log_2 n + 1$

That is, $H_n \leq \log_2 n + 1$. Hence, we get that $E[X] < \sum\limits_{i=1}^{n-1} 2(\log_2 n + 1)$. This implies that $E[X] = O(n \log n)$. That is, the expected value of X is $O(n \log n)$. This proves that the average case time complexity of Quick Sort algorithm is $O(n \log n)$.

Method 2:

We can find out average case time complexity directly from Algorithm 7 (not considering its randomized version). Like before, we assume that all inputs are equally probable. To get the average time complexity ($T(n)$) of the algorithm, we consider all possible partitioning of the list. If the pivot is the i^{th} smallest element, then $T(n-1)$ and $T(n-i)$ are the average time required to sort left and right sublists respectively. Hence, we get the following relation:

$$T(n) \leq cn + \frac{1}{n} \sum_{i=1}^{n} \{T(i-1) + T(n-i)\}, \quad \text{if } n \geq 2 \tag{7.1}$$

Clearly, $T(0) = T(1) = b$ (constant). By expanding the above relation, we get the following:

$$T(n) \leq cn + \frac{2}{n}[T(0) + T(1) + \cdots + T(n-1)], \quad \text{if } n \geq 2 \qquad (7.2)$$

Now by unrolling the above recurrences, we can get the asymptotic bound of $T(n)$.

$$\begin{aligned}
T(n) &\leq cn + \tfrac{2}{n}[T(0) + T(1) + \cdots + T(n-2) + T(n-1)] \\
&\leq cn + \tfrac{2}{n}[T(0) + T(1) + \cdots + T(n-2) + c(n-1) + \tfrac{2}{n-1}[T(0) + T(1) + \cdots + T(n-2)]] \\
&= cn + \tfrac{2c(n-1)}{n} + \tfrac{2}{n}[(1 + \tfrac{2}{n-1})(T(0) + T(1) + \cdots + T(n-2))] \\
&= cn + \tfrac{2c(n-1)}{n} + \tfrac{2}{n}.\tfrac{n+1}{n-1})[T(0) + T(1) + \cdots + T(n-2)] \\
&\leq cn + \tfrac{2c(n-1)}{n} + \tfrac{2}{n}.\tfrac{n+1}{n-1})[T(0) + T(1) + \cdots + T(n-3) + c(n-2) + \\
&\quad \tfrac{2}{n-2}[T(0) + T(1) + \cdots + T(n-3)]] \\
&= cn + \tfrac{2c(n-1)}{n} + \tfrac{2c(n-2)}{n}.\tfrac{n+1}{n-1} + \tfrac{2}{n}.\tfrac{n+1}{n-1}.\tfrac{n}{n-2}[T(0) + T(1) + \cdots + T(n-3)] \\
&= cn + \tfrac{2c(n-1)}{n} + \tfrac{2c(n-2)}{n}.\tfrac{n+1}{n-1} + \tfrac{2c(n-3)}{n}.\tfrac{n+1}{n-1}.\tfrac{n}{n-2} + \\
&\quad \tfrac{2}{n}.\tfrac{n+1}{n-1}.\tfrac{n}{n-2}.\tfrac{n-1}{n-3}[T(0) + T(1) + \cdots + T(n-3)] \\
&= cn + \tfrac{2c(n-1)}{n} + \tfrac{2c(n-2)}{n}.\tfrac{n+1}{n-1} + \tfrac{2c(n-3)}{n-1}.\tfrac{n+1}{n-2} + \\
&\quad \tfrac{2(n+1)}{(n-2)(n-3)}[T(0) + T(1) + \cdots + T(n-3)] \\
&\quad \vdots \\
&\leq cn + \tfrac{2c(n-1)}{n} + \tfrac{2c(n-2)}{n}.\tfrac{n+1}{n-1} + \tfrac{2c(n-3)}{n-1}.\tfrac{n+1}{n-2} + \tfrac{2c(n-4)}{n-2}.\tfrac{n+1}{n-3} + \cdots + \tfrac{2c}{5}.\tfrac{3}{4} + \\
&\quad \tfrac{2c}{5}.\tfrac{3}{4} + \tfrac{2(n+1)}{4.3}[T(0) + T(1)] \\
&= cn + \tfrac{2c(n-1)}{n} + 2c(n+1)[\tfrac{n-2}{n(n-1)} + \tfrac{n-3}{(n-1).(n-2)} + \tfrac{n-4}{(n-2).(n-3)} + \cdots + \tfrac{3}{5.4}] + \\
&\quad \tfrac{2(n+1)}{4.3}.2b \\
&< cn + \tfrac{2c(n-1)}{n} + 2c(n+1)[\tfrac{1}{n} + \tfrac{1}{n-1} + \tfrac{1}{n-2} + \cdots + \tfrac{1}{5}] + \tfrac{2(n+1)}{4.3}.2b \\
&< cn + \tfrac{2c(n-1)}{n} + 2c(n+1)\log_2 n + \tfrac{2(n+1)}{4.3}.2b \quad \forall n \geq 5
\end{aligned}$$

Here $\frac{1}{n} + \frac{1}{n-1} + \frac{1}{n-2} + \cdots + \frac{1}{5} < H_n$, and we have seen that $H_n \leq \log_2 n + 1$. Hence, we can rewrite the above as following:

$$T(n) < 2cn\log_2 n + An + 2c\log_2 n + B \quad \forall n \geq 5$$

for some constants A and B. This implies that $T(n) = O(n \log n)$.

Method 3:

This method follows the same style like the above. It uses Relation 7.2 to get the average case complexity. In the above method, we have unrolled the recurrence relations systematically. But here we use Substitution method to solve the above recurrence relation.

Let us guess that $T(n) \leq kn \ln n$ for $n \geq 2$, where $k = 2c + 2b$. We prove through induction that our guess is correct. For the base case $n = 2$, $T(2) \leq 2c + 2b$ follows immediately. Assume for induction that $T(i) \leq ki \ln i$ for $i \leq n - 1$. Then we get

$$\begin{aligned} T(n) &\leq cn + \tfrac{4b}{n} + \tfrac{2}{n}\sum_{i=2}^{n-1} ki \ln i \\ &\leq cn + \tfrac{4b}{n} + \tfrac{2}{n}\int_2^n kx \ln x \, dx \\ &= cn + \tfrac{4b}{n} + \tfrac{2k}{n}[\tfrac{n^2 \ln n}{2} - \tfrac{n^2}{4} - (2\ln 2 - 1)] \\ &\leq cn + \tfrac{4b}{n} + kn \ln n - \tfrac{kn}{2} \end{aligned}$$

Since $n \geq 2$ and $k = 2c + 2b$, it follows that

$$cn + \frac{4b}{n} \leq cn + bn \leq \frac{kn}{2} \quad \text{and} \quad T(n) \leq kn \ln n$$

Hence, our guess was correct. This implies that the average case complexity of the algorithm is $O(n \log n)$.

Chapter 8

Sorting Problem - continued

In this chapter we shall discuss two families of sorting algorithms – Sort by Merging and Sort by Selection.

8.1 Merge Sort

This algorithm follows ‘Sort by Merging’ technique. Merge Sort Algorithm is a classical example of Divide-and-Conquer method of designing algorithms.

Suppose a list of n elements is given for sorting. Merge Sort algorithm divides the list into two sublists of length $n/2$. Each sublist is again dived into two parts. We repeat this process until we get a smallest sublist of single element, which is trivially sorted. Next, the sorted lists are merged to get a new sorted list. Hence, merging is the primary operation by which we prepare the final list.

Suppose two sorted lists as following are given, which are to be merged.

$[212, 319, 512]$ and $[121, 201, 209, 302]$

Initially, two pointers i and j point to the first elements of the lists. If $K_i \leq K_j$, then K_i is put in a temporary array and i is incremented. Otherwise, K_j is put in the temporary array and j is incremented.This process is repeated. If a list gets exhausted, the elements of other list are simply added in the temporary array. For the above example, the second list exhausted first. Following is the result after merging:

$[121, 201, 209, 212, 302, 319, 512]$

If the sizes of sorted lists are k and l, then in worst case $\max(k, l)$ key comparisons are needed. So, the time complexity is $O(\max(k, l))$. In this algorithm, we need a temporary array of size $k+l$ to store the sorted array.

We can write recursive method for Merge Sort. The list of n elements are divided into two sublists of length $\frac{n}{2}$. From the first sublist we get a sorted list, say A_1 and from the second sublist get a sorted sublist, say A_2. Then these two sublists are merged to get the final list. For this method, the worst and best cases against space and time are asymptotically same. Time complexity of the algorithm can be expressed as

$$T(n) = 2T(\frac{n}{2}) + O(n)$$

That is, $T(n) \leq 2T(\frac{n}{2}) + c.n$ for some c. This implies, $T(n) = O(n \log n)$. **Space complexity** for the best and worst cases of this algorithm is $O(n)$.

Quick and Merge sort: In case of Quick Sort, the time complexity is $O(n \log n)$. So, Quick Sort and Merge Sort have asymptotically same time complexity. Now the question arises that which one is better (if we ignore the space complexity)? Firstly we can say that for Quick Sort we can do some intelligent division (taking a pivot element first and then divide the list after some comparisons) whereas for Merge Sort we divide the list blindly. After each pass in Quick Sort, one element (the pivot element) is placed in its final position, from where we need not to move that element further. This does not happen in Merge Sort generally. Practically, movement of records in Merge Sort is higher; it also uses a temporary array for the record movement.

For Merge Sort, elements of two sublists are compared during merging, but in Quick Sort elements from different sublists are never compared. Hence Quick Sort demands less number of key comparisons. Combining all these factors, we can say that the Quick Sort is better than the Merge Sort.

In-place and stable sorting: An algorithm is an *in-place* algorithm if the input (array of) data is not moved to temporary locations, rather the input list is modified directly to give the output. Merge Sort is not an in-place sorting algorithm whereas Quick Sort, Bubble Sort, etc. are in-place algorithms. A sorting algorithm is *stable* if the records with equal keys retains their original order. That is, if $K_i = K_j$ and $i < j$, and if after sorting K_i and K_j are placed is positions i' and j', then for stable algorithms, $i' < j'$. Our Quick Sort algorithm (Algorithm 7) is not a stable algorithm.

8.2 Sort by Selection

In this family of algorithms, minimum (or, maximum) element is selected to consider it as the first (or last) element. From the remaining elements again minimum (or, maximum) elements is selected and considered as second (or second last) element. And so on.

8.2.1 Straight Selection Sort

Suppose there is a list of n elements. Select the maximum element from the list. Then put it in the last position. Then for the rest $(n-1)$ elements again follow the same procedure.

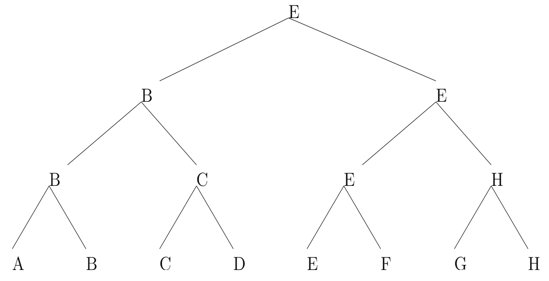
Theorem 5 : *To select minimum or maximum element from a list of n elements, atleast $n-1$ comparisons are needed.*

So total number of comparisons required for this Straight Selection Sort is $(n-1)+(n-2)+\cdots+1=\frac{n(n-1)}{2}$. Hence, time complexity of this algorithm is $T(n)=\Theta(n^2)$. The best and worst cases are asymptotically same here.

However in this process we need to explore always the whole list to select minimum element form this. Can we get an improved way of choosing the maximum element? Let us now discuss the issue and for that we can consider here the scenario of a knock-out tournament.

8.2.2 Knock-out tournament

In a knock-out tournament suppose there are 8 teams A,B,C,D,E,F,G,H. We are *searching* for the Champion. In first stage, suppose following teams in pairs (A,B), (C,D), (E,F), (G,H) play matches against each other and let the winners be B,C, E, H. Then (B,C) and(E,H) matches are played. Suppose B and E are the winners. Then B and E compete with each other and let's say E is the winner. Hence E is the champion in the tournament. We can represent the whole scenario by the following tree.



Here, the parents represent the winners and the root is the champion. Observe that there is no play between some teams such as (B,D),(E,G) etc. For 8 teams we need 7 matches to get the champion. This fact agrees with Theorem 5.

But to get the second best, we need much lesser plays if we use the results of already played matches. We see that there has been no play in the pair (B, G), (B,H) and (B,F). Here, match between B and G is redundant, because H was winner in the match of G and H. So we need two more matches for (B,H) and (B,F). Let the winner of both the cases be B and so B is the second best. Here to choose the second best we consider the results of already-played matches and removing the E from the list. That is, for the search of second best, we use the previously formed tree, due to which we can reduce the number of plays. So, once we form the tree we can use it for further cases. It is called *Tree Selection*.

So, if we can develop an algorithm taking inspiration from this knock-out tournament for selecting maximum (or minimum) element, then that algorithm can perform much better than the Straight Selection Sort.

8.2.3 Heap Sort

We utilize this kind of technique in Heap Sort. Let us say a list of keys $K_1, K_2, \cdots,_n$ is a *heap* if

$$K_{\lfloor j/2 \rfloor} \geq K_j \quad 1 \leq j/2\rfloor < j \leq n.$$

Thus, $K_1 \geq K_2, K_1 \geq K_3, K_2 \geq K_4$, etc. This implies that the largest element appears on the top of the heap.

We first *heapify* the given list of records to efficiently select maximum element. Details of Heap Sort is skipped here. But one can see that the worst time complexity for this algorithm is $T(n) = O(n \log n)$ and space complexity is $O(1)$.

Hence we get two classes of sorting algorithms. One has worst case time complexity $O(n^2)$ and other has complexity $O(n \log n)$. But can we further improve sorting algorithms? Is it possible to design a better algorithm with time complexity $O(n)$?

Chapter 9

Sorting in Linear Time

There are two linear time sorting algorithms – Counting Sort and Bucket Sort. These are called Special Purpose Sorting Algorithms as they do not take any list of records as input but put additional constraints on input list.

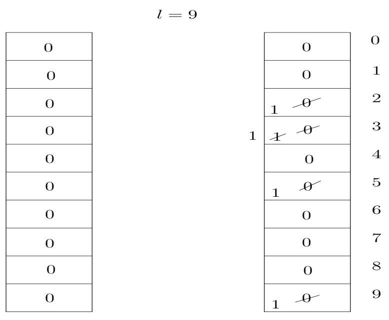
9.1 Counting Sort

We use this algorithm if the following two constraints are satisfied by the input list of records. This technique has following restrictions:

1. Records are non-negative integers.
2. If the highest integer in the input list is l, then $l = O(n)$.

In this algorithm, a temporary array of size $l+1$ is taken, which is initialized to 0. The keys (records) are considered as the indices of the temporary array. The input list is scanned from the beginning and the content of the temporary array against a key is increased by 1.

As example, consider the input list as $2, 9, 3, 5, 3$. Here $l = 9$.All the locations of the temporary array are 0 initially. When we get 2, then the location 2 of the temporary array is increased to 1. Since 3 appears twice in the list, location 3 of the temporary array is raised to 2 after scanning of the list.



To give sorted output, we scan the temporary array from the beginning. Whenever, we get a non-zero value, the index is printed. If the non-zero value is k, then that particular index is printed k times.
Time Complexity: This algorithm uses two loops - one to scan the input list of size n, and the other to scan the temporary array of size l for giving output. Hence, the time complexity is $T(n) = O(\max n + l)$. Since our assumption is $l = O(n)$, so $T(n) = O(n)$.
Space Complexity: The algorithm uses a temporary array of size $l + 1$. So the space complexity is $S(n) = O(l)$. That is, $S(n) = O(n)$.

9.2 Bucket Sort

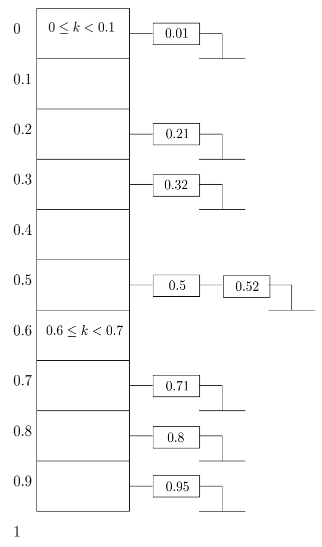
This algorithm is applied on an input list if the following restriction is fulfilled: The records are normalized reals, distributed uniformly over [0,1).

The idea of bucket sort is to divide the interval [0, 1) into n equal-sized *buckets* and then distribute the input n numbers into the buckets. Since uniform distribution is assumed, we do not expect many numbers to be in a bucket. As a next step, we individually sort each bucket, and then output the sorted buckets starting from the first.

Let us take the following list of numbers as input:

$$0.5, 0.21, 0.52, 0.71, 0.01, 0.8, 0.95, 0.32, 0.673$$

Here $n = 10$. So we need a temporary array that can hold these 10 buckets. Under each bucket a list is formed to store the numbers under that particular bucket. The first number is 0.5, so it goes to the 6th bucket. Following is the scenario.



After distribution of numbers, buckets are sorted using any general purpose sorting algorithm. Following is the algorithm.

Algorithm 8 Input: *A list of normalized reals* $K_1, K_2, \cdots, K_n \in [0, 1)$
Output: *Sorted list*
Step 1: For each $i = 1, 2, \cdots, n$*, perform Step 2.*
Step 2: Insert K_i *into* $B[\lfloor n.K_i \rfloor]$ *where* B *is a temporary array of buckets.*
Step 3: For each $i = 0, 1, \cdots, n-1$*, perform Step 4.*
Step 4: Sort the list $B[i]$ *(using any algorithm).*
Step 5: Concatenate the sorted lists $B[0], B[1], \cdots, B[n-1]$ *to output the result.*

Let us now analyse the time complexity of the algorithm. There are two loops in the above algorithm and both run for n steps. We intuitively understand that the algorithm runs in $O(n)$ time, because each list of B contains ideally single element, and so there is nothing to sort. Let us now prove it more formally.

Assume that n_i is the length of list of bucket i. As per Step 4, we can use any sorting algorithm to sort the bucket. Let us use Straight Insertion Sort to sort the buckets. So, worst case complexity for sorting bucket i is $O(n_i^2)$.If $T(n)$ is time complexity of Algorith 8, then we can write:

$$T(n) \le cn + \sum_{i=0}^{n-1} c_i n_i^2 \tag{9.1}$$

We now find out expected value of $T(n)$. Here n_i is the random variable which depends on input data. Taking expectation in both sides of the above equation, we get that

$$\begin{aligned} E[T(n)] &\leq E\left[cn + \sum_{i=0}^{n-1} c_i n_i^2\right] \\ &= cn + \sum_{i=0}^{n-1} c_i E[n_i^2] \quad \text{[By linearity of expectation]} \end{aligned} \tag{9.2}$$

$E[n_i^2]$ depends on the fact that how many elements fall in the bucket i. Let us consider an indicator random variable X_{ij} which notes if K_j falls in bucket i. Hence,

$$n_i = \sum_{j=1}^{n} X_{ij}$$

Now, we can find out $E[n_i^2]$.

$$\begin{aligned} E[n_i^2] &= E\left[\left(\sum_{j=1}^{n} X_{ij}\right)^2\right] \\ &= E\left[\sum_{j=1}^{n}\sum_{k=1}^{n} X_{ij}X_{ik}\right] \\ &= E\left[\sum_{j=1}^{n} X_{ij}^2 + \sum_{1\leq j\leq n}\sum_{\substack{1\leq k\leq n\\ k\neq j}} X_{ij}X_{ik}\right] \\ &= \sum_{j=1}^{n} E[X_{ij}^2] + \sum_{1\leq j\leq n}\sum_{\substack{1\leq k\leq n\\ k\neq j}} E[X_{ij}X_{ik}] \end{aligned} \tag{9.3}$$

Since X_{ij} is an indicator random variable with uniform distribution, it is 1 with probability $1/n$ and 0 otherwise. So

$$E[X_{ij}^2] = 1.\frac{1}{n} + 0.(1 - \frac{1}{n}) = \frac{1}{n}$$

When $k \neq j$, X_{ij} and X_{ik} become independent. So we get

$$E[X_{ij}X_{ik}] = E[X_{ij}]E[X_{ik}] = \frac{1}{n}.\frac{1}{n} = \frac{1}{n^2}$$

Substituting these values in Equation 9.3, we obtain the following:

$$\begin{aligned} E[n_i^2] &= \sum_{j=1}^{n} \tfrac{1}{n} + \sum_{1\le j\le n} \sum_{\substack{1\le k\le n\\ k\ne j}} \tfrac{1}{n^2} \\ &= n.\tfrac{1}{n} + n(n-1).\tfrac{1}{n^2} \\ &= 2 - \tfrac{1}{n} \end{aligned}$$

Putting this result in Relation 9.2, we get that

$$E[T(n)] \le cn + \sum_{i=0}^{n-1} c_i(2 - \frac{1}{n})$$

Now we can easily show from this expression, that is the expected value of $T(n)$ is $O(n)$. This concludes that fact that the bucket sort algorithm runs in linear expected time.

Chapter 10

Lower Bound Theory

We see that there are some Special Purpose Sorting Algorithms which have linear time complexity. But the good General Purpose Sorting Algorithms take $O(n \log n)$ time . Question is, can we develop a better general purpose sorting algorithm with lesser asymptotic time complexity? To answer this question, we need to know the *Lower Bound* of the Sorting Problem.

For a given problem P, there are countably many algorithms. For the problem, the lower bound is the worst case complexity of the best possible algorithm. In other word, the minimum amount of resources (time or space) that the problem demands to get solved by any algorithm is the lower bound of the problem. Note that, for any algorithm we are generally interested in upper bound of resource requirement, whereas for a given problem, we are interested to know the lower bound of its resource requirement. Following are some example problems and their lower bounds.

1. Finding of minimum or maximum from an unsorted list of n elements: we have to visit all the locations to find the minimum (or maximum) element. So, the lower bound of this problem is $\Omega(n)$. It is impossible to find an algorithm with lesser complexity, say $O(\log n)$.

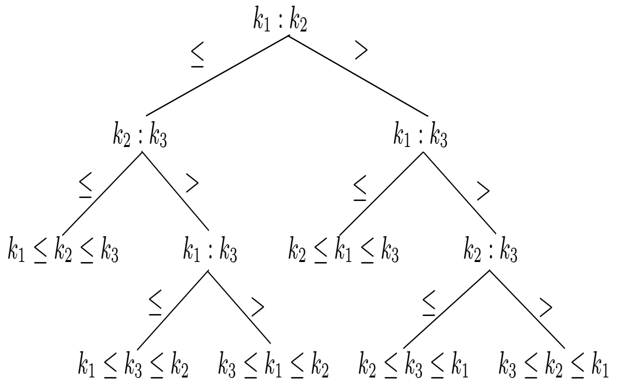
2. Multiplication of two square matrices of dimension $n \times n$: Resultant matrix is of dimension $n \times n$. Each of the $n \times n$ locations are to be visited to get the answer. So the complexity of any algorithm can never be lower than n^2. So, lower bound of this problem is $\Omega(n^2)$.

3. Finding of all possible permutations of elements of a set of n elements: Since the number of possible permutations is $n!$, the lower bound of

this problem is $\Omega(n!)$.

Now we will search for the lower bound of Sorting Problem with respect to time complexity. We deal with general purpose sorting problem where sorting is done by comparing keys only. This class of algorithms are also known as Comparison based Sorting Algorithms. Here comparison is the primary operation. So, minimum number of comparisons required in worst case to sort a list of n elements is to be the lower bound of the problem.

Let us first discuss with an example of three elements. Let K_1, K_2, K_3 be the elements to be sorted. Here 3! permutations of the keys are possible, one of which is the required output. We can first compare K_1 and K_2. If $K_1 \le K_2$, then we compare K_2 and K_3. If $K_2 \le K_3$, then sorted order is $\langle K_1, K_2, K_3 \rangle$. We can use *Decision-Tree model* to get all possibilities. Following is the whole scenario.



Observe that leaves of the tree the all possible ordering of input keys, one of which is the sorted order. What is the worst case here? Worst case is 3 comparisons, which is the height of the tree. Let the height of the tree be h. Hence, the lower bound of this problem is h. If it is a complete binary tree the number of leaves will be 2^h. But the decision tree can have $n!$ nodes. So,

$$2^h \ge n! \Rightarrow h \ge \lceil \log_2 n! \rceil$$

We can use Stirling's Approximation: $n! = \sqrt{2\pi n}(\frac{n}{e})^n[1 + \Theta(\frac{1}{n})]$. That is,

$$\lceil \log_2 n! \rceil = n \log_2 n - \frac{n}{\log_e 2} + \frac{1}{2} \log_2 n + O(1)$$

Hence, roughly $n \log_2 n$ comparisons are needed. That is, $h = \Omega(n \log n)$.

One can find out this bound without using Stirling's approximation:

$$\begin{aligned}
\log_2 n! &= \log_2 n + \log_2(n-1) + \cdots + \log_2 2 \\
&= \sum_{i=2}^{n} \log_2 i \\
&= \sum_{i=2}^{n/2-1} \log_2 i + \sum_{i=n/2}^{n} \log_2 i \\
&\geq 0 + \sum_{i=n/2}^{n} \log_2 \tfrac{n}{2} \\
&= \tfrac{n}{2} \log_2 \tfrac{n}{2}
\end{aligned}$$

Hence, $h = \Omega(n \log n)$. That is, the lower bound of the problem is $\Omega(n \log n)$. This lower bound is called *Information-Theoretic Lower Bound*. It proves that, Heap Sort, Merge Sort are asymptotically optimal algorithms.

Remark: For finding lower bound of a problem, choice of computational model is very important. In case of above problem, we have considered that ordering of input keys are decided by comparing the keys on a single-processor computer. If we choose parallel processing, distributed computing or quantum computing environment, lower bound of a problem may be different.

Chapter 11

External Searching and Sorting

The Searching and Sorting Algorithms we have discussed till now are *internal*, because input lists are stored completely in main memory. Sometime the main memory may not accommodate the whole data, and then the data are to be stored in secondary storage. An algorithm is called *external* if it is designed to process data that are stored in secondary (external) memory.

11.1 External Searching

In case of external searching, the list of records resides in secondary memory. To search an element from the list, a part of the data is brought to the main memory from the secondary storage. And then that part of the list is searched. If unsuccessful, then again another part of data is brought to the main memory. And so on.

To analyse algorithms for external searching, the model of computation that we have considered till now will not work. Because, in those models such as RAM model (similarly in the previous algorithms), we assume that the main memory is large enough to accommodate all input data. But this is not true here.We also need another metric to measure the performance of external algorithms: the number of *disk access*. In general, disk access is an expensive operation in any computer, because it demands a great amount of time compared to accessing data from main memory.

So a general target of the external searching algorithms is to reduce the number of disk access. For that we have to organise data in a special way. To facilitate that organisation, we exploit the structure of *B-tree*.

11.1.1 B-tree

B-trees are multiway balanced trees. Although B-trees are developed for external searching, they are also treated as *self balancing* tree data structure.

Definition 7 *A B-tree of order p is a tree that satisfies the following properties:*

1. *Every node have at most p children.*
2. *Every node, except the root and the leaves, has atleast $\lfloor \frac{p}{2} \rfloor$ children.*
3. *The root has atleast 2 children (unless it is not a leaf).*
4. *All leaves are in same level.*
5. *A node with q children contains $q - 1$ keys.*
6. *Keys in a node are sorted in ascending order.*

Therefore,the B-tress are balanced and can be treated as search trees. Following is an example B-tree of order 5.

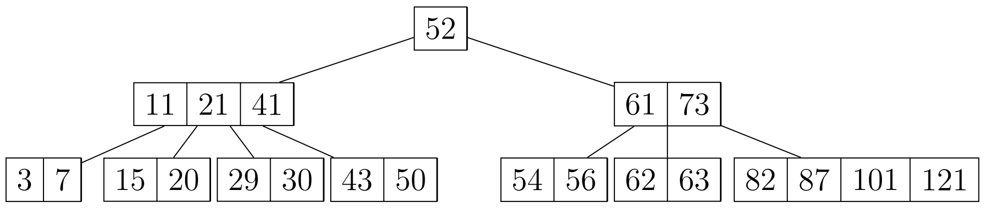


Figure 11.1: B-tree of order 5

Observe that each node is sorted in ascending order. Elements of the left (first) child of a node are less than the first element of the node. Elements of second child of the node are in between the first and second elements of the node. And so one. So this tree can act as search tree.

Let's say, we want to search for element 39. The searching starts from the root. Since 39 is less than 52 (the root), we go to the left child of the root. The left child has three elements, and 39 is in between 21 and 41. So we go to this left child's child whose elements are to be in between 21 and 41. Whenever we go to that node (here it is a leaf), we find that the search key does not exist. So our search remains unsuccessful.

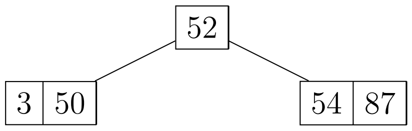
An important thing about B-tree is that insertion is quite simple. A new element is always added at a leaf node.If the number of elements reaches to

p (the order) after addition, the leaf node is split into two and the middle element move to their parent, if any (otherwise, the middle element becomes the root). Since the tree grows from the leaf, it remains balanced always.

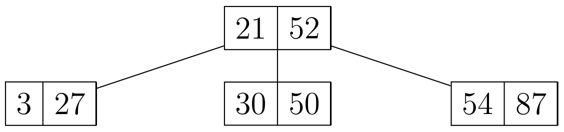
Figure 11.1 is the B-tree of order 5 which we get for the following sequence of data: 50, 87, 3, 52, 54, 21, 7, 30, 15, 62, 73, 43, 101,63, 56, 61, 82, 11, 29, 41, 20, 121. Insertion of upto first four elements,the tree remains as a single-node tree:

3	50	52	87

However, if one more element is inserted then the condition of B-tree is violated as the tree can accommodate at most 4 keys in a node. So the node is split and the middle key becomes the root:



Number of levels of a B-tree increases only when the nodes cannot accommodate newly inserted data. In the above tree, if we insert next two elements (21 and 7), they are added in the left child. But if the next element (30) of the sequence is inserted, then the left child violates the condition, and so it is split into two. And the middle element moves to the upward direction:



In this way, the tree grows, and its levels increases in upward direction, which keeps the tree balanced always. If we insert the elements in above sequence, we get the tree of Figure 11.1. Obviously, the structure of tree depends on the sequence of data.Deletion of an element is also simple though it is comparatively complicated than the insertion.

A (non-leaf) node in a B-tree with $q-1$ keys has q children .So the nodes contains q pointers that point to its children. Hence following is the structure of a node. Here small circles are representing the pointers.

$\circ$	K_1	$\circ$	K_2	$\cdots$	K_{q-1}	$\circ$

11.1.2 Searching from disk

To search records efficiently from secondary memory, B-tree is formed by the keys of records. By accessing the nodes of the B-tree, we can decide whether the record against the search key exists or not. If exists, we wish to output the record by another disk access. To facilitate this, we need to store the disk address of the record where it really exists along with key. Hence the structure of a node of the B-tree looks like the following:

$\circ$	$(K_1, \circ)$	$\circ$	$(K_2, \circ)$	$\cdots$	$(K_{q-1}, \circ)$	$\circ$

Here the small circle within a box represents the pointer that points to a child of the node, whereas the circle along with a key represents a pointer which points to a disk address where the record against the key exists. However, both the pointers hold some disk address, because the records as well as the nodes of B-tree are stored in disk.

To optimize the disk access, we make a node as large as possible so that a block of the disk can accommodate a node. Note that a disk is accessed block-wise, that is, by a single disk access, we get a *block* of data. Size of a node depends on size of a key and on the order (p) of the B-tree.Let's say that keys are 4-byte integers. Size of disk address is assumed as 8 bytes. Then, the maximum possible size of a node is

$$8p + 4(p-1) + 8(p-1) = 2p - 12$$

bytes. Generally size of a block is 4KB. To accommodate a node in a block, following condition is to be satisfied:

$$20p - 12 \leq 4000$$

This implies,

$$p \leq 200.6$$

Hence, for $p = 200$, a node can be accommodated in a single block.Maximum possible height of a B-tree of order p is $\lceil \log_{\lceil \frac{p}{2} \rceil} n \rceil$ where n is the number of keys. Hence, for unsuccessful search,

$$\text{Number of disk access} = \lceil \log_{\lceil \frac{p}{2} \rceil} n \rceil$$

which is much lesser in number if we do the same by sequentially accessing the records from disk.

To further reduce the number of disk access, variations of B-tree, such as B^+-tree have been developed. *Hashing* is also utilized for the same purpose.

11.2 External Sorting

External sorting typically uses a hybrid *sort-merge strategy*. In the sorting phase, chunks of data that are small enough to fit in main memory are read, sorted, and written out to a temporary file. In the merge phase, the sorted sub-files are combined into a single larger file.

One example of external sorting algorithm is the External Merge Sort, which sorts a chunk of data that fits in RAM, then merges the sorted chunks together. We first divide the file into *runs* such that a run can be fitted into main memory. Then we sort each run in main memory using merge sort algorithm. Finally, we merge the resulting runs together into successively bigger runs, until the file is sorted. Following are the steps of Sort-Merge strategy.

1. Read input file such that at most *run_size* elements are read at a time. Do the following two steps for the every run, read in an array.

2. Sort the run (say, using Merge Sort).

3. Store the sorted run in a file.

4. Merge the sorted files to get a single sorted file.

Chapter 12

Hashing

Why do we need sorting and searching? We observe that more than 90% of total computing time in commercial computing is spent in searching. Not only for commercial purpose, even in scientific computing we require searching most of the time. Sorting is required in many cases to felicitate the searching. And, sorting is needed to organise data in memory. So, the sorting and searching are included in the study of *Information Storage and Retrieval*.

As always, we want to have a faster way of information retrieval, that is, of searching technique. For the Searching Problem, however, the lower bound is $\Omega(\log n)$, whereas it is $\Omega(n \log n)$ for Sorting Problem. We are limited by these bounds. There are asymptotically optimal algorithms for searching and sorting, such as Binary Search and Merge Sort.

Now if we want to have even faster searching algorithms, we have to alter the way of searching. We have observed that the searching (and general purpose sorting) are comparison based. If we can search data without solely by key comparisons, the above lower bound of searching will no longer remain valid. We get some hints from Counting Sort that if we can bypass the key comparisons, then we may get faster sorting algorithm.

Thanks to *Hashing*, which finally gives us a very fast technique of searching element. It effectively runs in constant time. Instead of searching by key comparisons, it computes the probable location of the search key and then look at that location. Like Binary Search Trees and B-tree, hashing tells us how to store data in memory. Therefore, if sorting is used to organize data in memory to facilitate efficient access of records, then the same purpose can be fulfilled by hashing itself. Hashing means to chop something up or to make a mess out out of it. The idea in hashing is to scramble some aspects of key and to use this partial information as the basis for searching.

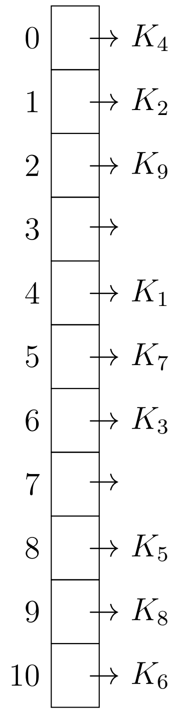
12.1 Hash functions

We compute a *hash address* $h(K)$ using the *hash function* h and start searching there. Before that we store the data in memory (in tabular form) using the same hash function. This memory area is called *hash table*.

Let us consider that the n records with keys $K_1, K_2, \cdots, K_n$ are stored in m locations. So the condition

$$0 \leq h(K) < m$$

for all keys K are to be satisfied. We want to design a good hash function which has very less computational cost and distribute the keys uniformly over m memory locations. That is, if $m \geq n$, then each key should occupy one memory location. Following is an example scenario of a good hash function for $n = 9$ and $m = 11$:



However, design of such hash functions is very difficult.There are m^n possible functions for a domain of size of n and a co-domain of size m. Even for small m and n, say $m = 100$ and $n = 80$, possible number of functions ($100^{80} = 10^{160}$) unmanageable. It is interesting to mention here that the age of this universe is only 10^{43} seconds and number of particles in the observable universe is in between 10^{78} to 10^{82}. These figures hint that how difficult it is to choose the best hash functions from all possibilities.

Therefore, we may not see the above ideal scenario for a chosen hash function. We can see that for two different keys K_i and K_j, $h(K_i) = h(K_j)$. This situation is called *collision*, which is very common in hashing.If such situation arises, we have to resolve the collision, that is, we need to adopt *collision resolution* policy. So in hashing, two issues are important: (a) design of good hash functions, and (b) collision resolution technique.

Task of a good hash function is to generate hash addresses to uniformly distribute the keys (and records) over the hash table. The hashing where the keys are distributed uniformly over hash table is called *simple uniform hashing*. Although it is theoretically impossible to design hash functions for simple uniform hashing, extensive experiment on typical lists of records have shown that two major types of hash functions work quite well. One is based on division, and the other is based on multiplication.

12.1.1 Division Method

The division is particularly easy; we simply use the remainder after dividing the key by m:

$$h(K) = K \mod m$$

For example, if $m = 11$ and $K = 100$, then $h(K) = 1$. Here we consider the keys are natural numbers. Question is, can we consider a key in the form of string or real number as a natural number? Answer is yes. A string of characters can be seen as a sequence of 0s and 1s, hence a natural number. Similarly for other types of keys, they can either be interpreted or be approximated as a natural number.

In this method, some values of m is much better than others. If m is even, then $h(K)$ is even when K is even and odd when K is odd. This bias may lead to bad result in many cases.The scenario becomes even worse if $m = 2^p$ for some p. Experience says that if we choose m as prime number, performance of the hashing is satisfactory.

12.1.2 Multiplicative Method

Multiplicative hashing scheme is equally easy to do, but it is slightly harder to perceive the idea as it works with fractions instead of integers. Let us first fix a constant A such that $0 < A < 1$. Then, the define the hash function as following

$$h(K) = \lfloor m(K.A \mod 1) \rfloor$$

for all keys K. Here by the operation $K.A \mod 1$, we extract the fractional part of $K.A$. In this case we usually consider $m = 2^p$ for some integer p, so that $h(K)$ consists of the leading bits of the fractional part of $K.A$.

Here quality of hashing does not only depend on m, but also on A. It is generally difficult to choose right value of A.It is argued that if A is approximately the golden ratio, that is, if $A \approx \frac{\sqrt{5}-1}{2} = 0.6180339887$, then the hashing performs better.

Note that computational complexity of hash function of both the method is very less and constant. That is, the time complexity is $O(1)$, which is another requirement of designing good hash functions.

12.1.3 Universal Hashing

In the above cases, fixed hash functions are proposed for hashing, which may yield bad result for some data sets. To achieve uniform hashing criterion, we can use a set of suitably designed hash functions. One function from that set is to be chosen randomly for a given list of records. This approach is known as *universal hashing*. Due to random selection of hash function, for a single list of given records, we can get different scenario for two different execution. So, on an average, the performance of universal hashing scheme is better than others. By performance, we mean here that number of collision gets minimized.

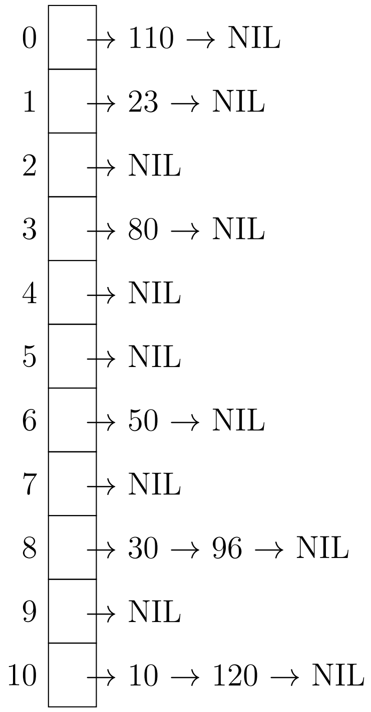
Let H be the set of hash functions that map the keys to m memory locations. This set is called *universal* if for each pair of distinct keys K and K', the number of hash functions $h \in H$ for which $h(K) = h(K')$ is at most $|H|/m$. In other words, for a randomly chosen hash function, the chance of collision for these two distinct keys is at most $\frac{1}{m}$.

By taking all the above measure and designing good hash function, however, we cannot avoid collision completely. So we need collision resolution technique.

12.2 Collision resolution by chaining

When two keys possess same hash address, that is, when $h(K_i) = h(K_j)$ for $K_i \neq K_j$, both the records against the keys are kept in same address by forming a linked list (*chain* of records). Therefore, we get here chained hash table, and the m locations of the table contains starting address of m linked lists.

We illustrate this method by following example. Suppose we have eight records with keys 96, 50, 120, 10, 110, 30, 23, 80, and are using the hash function; $h(K) = K \mod 11$. We get that $96 \mod 11 = 30 \mod 11 = 8$. So these two records are placed at location 8 by forming a chain. Similarly, records with keys 120 and 10 form a chain to share a single hash address. Final hash table looks like the following. Here we have shown only the keys (not the whole record).



When we insert a record in the table, we first find the hash address and then insert the record in the front of the linked list. Hence the time complexity for insertion is $\Theta(1)$. To delete a record from the table, we have search first if it exists in the table. If exists, then it takes constant time to remove the record from the table. So deletion is bounder by the searching complexity.

Let us define *load factor* (α) of a hash table as

$$\alpha = \frac{n}{m}$$

Let n_i is the length of chain i. So $n_0 + n_1 + \cdots + n_{m-1} = n$. Under the simple uniform hashing assumption, expected length of n_i is $E[n_i] = \alpha = \frac{n}{m}$.

Theorem 6 : *In a hash table in which collisions are resolved by chaining, an unsuccessful search takes expected time $\Theta(1+\alpha)$ under the assumption of simple uniform hashing. A successful search also takes time $\Theta(1+\alpha)$ on an average under the same assumption.*

Proof for unsuccessful search: In this case, the record does not exist in the table. First, hash address $i \in \{0, 1, \cdots, m-1\}$ is computed, then the chain of length n_i is searched. We consider that the computing of hash address takes $O(1)$ time. Since $E(n_i) = \alpha$, total time required in this case is $\Theta(1+\alpha)$.[I]

Proof for successful search: Let us consider that we search for the record with key K_i. So for successful search, we have to search 1 more than the number of keys before K_i in K_i's chain. And, the keys before K_i in its list are the keys K_j that are inserted after K_i and $h(K_i) = h(K_j)$. For keys K_i and K_j, let us define an indicator random variable X_{ij} that notes if $h(K_i) = h(K_j)$. Under simple uniform hashing, $Pr\{h(K_i) = h(K_j)\} = 1/m$, and so $E[X_{ij}] = 1/m$.

Now, K_i can be any key in the list of n records.So, to get the average time, we find the average number of keys searched in successful search. If this number if X, then

[I]Note here that the '+' is equivalent to max. If $\alpha < 1$, then the bound is $\Theta(1)$, otherwise it is $\Theta(\alpha)$

$$
\begin{aligned}
X &= \tfrac{1}{n}\sum_{i=1}^{n}\Big(1+\sum_{j=i+1}^{n}X_{ij}\Big)\\
E[X] &= E\Big[\tfrac{1}{n}\sum_{i=1}^{n}\Big(1+\sum_{j=i+1}^{n}X_{ij}\Big)\Big]\\
&= \tfrac{1}{n}\sum_{i=1}^{n}\Big(1+\sum_{j=i+1}^{n}E[X_{ij}]\Big)\\
&= \tfrac{1}{n}\sum_{i=1}^{n}\Big(1+\sum_{j=i+1}^{n}\tfrac{1}{m}\Big)\\
&= 1+\tfrac{1}{nm}\sum_{i=1}^{n}(n-i)\\
&= 1+\tfrac{1}{nm}\tfrac{n(n-1)}{2}\\
&= 1+\tfrac{\alpha}{2}+\tfrac{\alpha}{2n}\\
&= \Theta(1+\alpha)
\end{aligned}
$$

Hence, expected time required for successful search is $\Theta(1+\alpha)$.

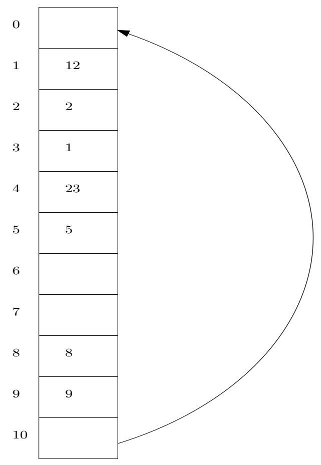
Therefore, the searching complexity heavily depends on α. If $\alpha < 1$, we can achieve $O(1)$ searching complexity. Even for $\alpha \geq 1$, if $n = O(m)$, we can search the hash table in constant time. However, the hashing scheme may degenerate if most of the elements of keys are clustered in few places of the hash table. In the worst case, searching becomes Sequential Searching!

12.3 Collision resolution by open addressing

Another way to resolve the issue of collisions is to do away with links completely, simply looking at various locations of the table one by one until either finding the key K or finding an empty position.

Let us adopt the simplest strategy to insert or search a record in the hash table: first find the hash address $h(K)$, then inspect the positions $h(K), h(K)+1, h(K)+2, \cdots$. If we encounter an open position while searching for K, we conclude that the record is absent in the table. When we want to insert a record, then we search for an empty location in the table.

As example, consider the following sequence of integers is to be into a hash table of eleven locations ($m = 11$): 2, 12, 9, 5, 8, 1, 23. Our hash function is $h(K) = K \mod 11$. The first two integers are inserted into locations two and one. Later when we want to insert 1, its hash address is 1, which is already occupied. So we sequentially search for a free place. We find that location three is empty, where 1 is inserted. Following is the hash table after insertion of above integers.



Here the table is considered as a circular array. And $\alpha = \frac{n}{m}$ is always less or equal to 1. For better result, we can keep at most 80% of the table as filled up. Observe that we are searching successively. This successively search is called Probe, and the sequence of search locations is called *probe sequence*.

At this stage, we can define an *auxiliary hash function* h' which takes the hash address as input and generates the probe sequence: $h'(h(K), i)$ gives the i^{th} entry in the probe sequence. Hence, the probe sequence is $\langle h'(h(K), 0), h'(h(K), 1), \cdots, h'(h(K), m-1)\rangle$. That is, the maximum length of the sequence is m.

So, we have to search and insert according to the probe sequence. The length of probe sequence determines the time complexity for search and insertion of the scheme. Following are the well-known probing techniques in open addressing.

1. Linear Probing
2. Quadratic Probing
3. Double Probing

Linear probing: In the above example, we have adopted linear probing technique. Here the auxiliary hash function is

$$h'(K, i) = (h(K) + i) \pmod m$$

Linear probing is easy to implement, but it sometimes suffers from the problem of *primary clustering.* That is, the records may get clustered in a single place.

Quadratic probing: For this probing technique, we choose the auxiliary hash function as

$$h'(K, i) = (h(K) + c_1.i + c_2.i^2) \pmod{m}$$

where $c_2 \neq 0$. The issue of primary clustering can be avoided here, though other clustering may be observed here.

Double hashing: For this technique, we use two hash functions $h_1(K)$ and $h_2(K)$, and find the probe sequence by

$$h'(k, i) = (h_1(k) + i.h_2(k)) \pmod{m}$$

This method is considered as the best method available for open addressing.

The analysis of open addressing depends, like chaining, on the load factor α. In our analysis, we assume that each key is equally likely to be a part of any possible probe sequence. This is called *uniform hashing.*

Theorem 7 : *The expected number of probes in an unsuccessful search is at most* $\frac{1}{1-\alpha}$*, whereas for successful case it is at most* $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$ *in an open-address hash table, assuming uniform hashing and* $\alpha < 1$.

Therefore, searching and insertion of records in open addressing can be very efficient. Deletion of records, however, is little bit cumbersome here. We cannot directly delete an element after locating it in the table. If we do so, the hash table may become inconsistent. While deleting, we generally put a special symbol to indicate that location is empty.

Chapter 13

Dynamic Programming

Although there is no general way of designing algorithms for a given problem, some standard methods have been developed which can be exercised to solve a problem. Some of the standard methods are

1. Divide-and-Conquer
2. Dynamic Programming
3. Greedy method

We have discussed Divide-and-Conquer method during discussion of Binary Search and Merge Sort algorithms. Here we shall discuss about Dynamic Programming. Dynamic programming is typically used to solve *Optimization Problems*. For this class of problems, we may get a number of possible solutions with different *cost*. Task here is to declare the solution with optimal (minimum or maximum) cost as *desired solution* to the problem.

However, an arbitrary optimization problem may not be solved using dynamic programming method. We next discuss the basics of dynamic programming method through the following example.

13.1 Matrix Chain Multiplication Problem

Let us consider a sequence of three matrices $A_1A_2A_3$ to be multiplied. There are two ways to multiply these three matrices – $((A_1A_2)A_3)$ and $(A_1(A_2A_3))$. Note that matrix multiplication is a binary operator; so we need to put parenthesis to depict the order of multiplication. One property of this matrix multiplication operator is that it is *associative*. So, each one of these gives the correct result. But both of them do not incur same computational cost.

Time complexity of matrix multiplication operation depends on number of scalar multiplications. Assume that A_1, A_2 and A_3 have dimensions 10×100, 100×80 and 80×20 respectively. So, total number of scalar multiplications required
for $((A_1A_2)A_3)$ is $(10 \times 100 \times 80) + (10 \times 80 \times 20) = 96000$ and
for $(A_1(A_2A_3))$ is $(100 \times 80 \times 20) + (10 \times 100 \times 20) = 180000$

Obviously, the first option of multiplication is noticeably better than the second one. Since both the options give same result, the first option should be chosen. Now if a sequence of n matrices are given, we get many options of multiplication. Here the problem is to find out an order of multiplication which demands minimum number of scalar multiplications (not to find the final matrix after multiplication). That is, we need to put parenthesis over the sequence to indicate the order of multiplication. So this problem is also known as *parenthesization* problem.

Problem Statement: Given a chain (sequence) of matrices $(A_1A_2 \cdots A_n)$ of different dimensions such that multiplication is allowed, find the order of multiplication so that number of scalar multiplications gets minimized. In other word, parenthesize the sequence of matrices so that the required number of scalar multiplications gets optimal.

To solve this problem, we need to see all possibilities of ordering before choosing the optimal one. Assume that $P(n)$ is the number of possibilities for the chain of n matrices. Obviously, $P(1) = P(2) = 1$ and $P(3) = 2$.

We can find out $P(n)$ for $n \geq 2$ through following recurrence relation. Suppose that parenthesis over the chain of matrices is put as follows: $(A_1 \times A_2 \times \cdots \times A_k) \times (A_{k+1} \times A_{k+2} \times \cdots \times A_n)$. Now, number of possibilities for $(A_1 \times A_2 \times \cdots \times A_k)$ is $P(k)$, and for $(A_{k+1} \times A_{k+2} \times \cdots \times A_n)$ possibilities are $P(n-k)$. Hence, we get the total possibilities

$$P(n) = \begin{cases} 1 & n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & n \geq 2 \end{cases} \tag{13.1}$$

Let us now see how $P(n)$ grows. Consider the following Table 13.1. Here we see that after $n = 6$, $P(n)$ is higher than 2^n. Hence $P(n) = \Omega(2^n)$. In fact,

Table 13.1: Growth of $P(n)$ of Equation 13.1

n	$P(n)$	2^n
1	1	2
2	1	4
3	2	8
4	5	16
5	14	32
6	42	64
7	132	128
8	542	256

one can show that $P(n) = \Omega(\frac{4^n}{n^{3/2}})$. So practically the number of possibilities $P(n)$ is unmanageable. That is, if we want to find all possibilities in a brute-force way, and target to find the optimal solution, then we cannot manage it. Dynamic programming can help us to dramatically improve the scenario.

It can be observed that in a number of possible parenthesizations, same parenthesization to a subsequence $A_iA_{i+1}\cdots A_j$ (*subproblems*) is repeated. For example, when $n = 4$, in two possibilities $((A_1 \times A_2) \times (A_3 \times A_4))$ and $(A_1 \times (A_2 \times (A_3 \times A_4)))$, the subproblem $(A_3 \times A_4)$ is repeated. So, if we can find the number of scalar multiplications required for $(A_3 \times A_4)$ once, that can be utilized later for the other occurrence. The repeated occurrence of a subproblem in several possible solution is called *overlapping subproblems*. In that case, we can store the solutions to subproblems so that they can be used later.

This problem has another interesting and important property, named *Optimal Substructure* property. Let us consider that the optimal parenthesization of $A_1A_2\cdots A_n$ splits the product between A_k and A_{k+1}. Then the parenthesization of the subchain $A_1A_2\cdots A_k$ (and $A_{k+1}A_{k+2}\cdots A_n$) is also optimal. Why is it so? Because, if there were a less costly way to parenthesize $A_1A_2\cdots A_k$, then that would produce another parenthesization with lesser cost than the optimal one - a contradiction. This property is known as optimal substructure property.This property is indeed essential to use dynamic programming to any problem.

Let us now use a 2D array m to store the solutions to subproblems. Consider that $m[i, j]$ store the minimum number of scalar multiplications needed for multiplying the sequence $A_iA_{i+1}\cdots A_j$. We can define $m[i, j]$

recursively.

$$m[i,j] = \begin{cases} 0 & i = j \\ \min\limits_{i \le k \le j} \{m[i,k] + m[k+1,j] + p_{i-1}.p_k.p_j\} & i < j \end{cases}$$

If $i = j$ then there is a single matrix. If we can solve this recurrence relation, then $m[1,n]$ is to be our desired solution. Let us take an example with $n = 6$ and following dimensions:

n	Dimension
A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25

We shall use the above recurrence relation to find out optimal number of scalar multiplications required. We need a 2-D array to store the values of $m[i,k]$.

	1	2	3	4	5	6
1	0	15750	7875	9375	11875	15125
2		0	2625	4375	7125	10500
3			0	750	2500	5375
4				0	1000	3500
5					0	5000
6						0

When $i = j$ then we put 0. So in the diagonal cells we put 0. Lower part of the diagonal is useless because of $i > j$. Now,
$m[1,2] = m[1,1] + m[2,2] + p_0 \times p_1 \times p_2 = 0 + 0 + 30 \times 35 \times 15 = 15750$
$m[2,3] = m[2,2] + m[3,3] + p_1 \times p_2 \times p_3 = 0 + 0 + 35 \times 15 \times 5 = 2625$
$m[1,3] = \min\{m[1,1] + m[2,3] + p_0 \times p_1 \times p_3, m[1,2] + m[3,3] + p_0 \times p_2 \times p_3\}$

$= \min\{0 + 2625 + 30 \times 35 \times 5, 15750 + 0 = 30 \times 15 \times 5\}$
$= min\{7875, 17900\} = 7875$
$m[1,4] = \min\{m[1,1] + m[2,4] + p_0 \times p_1 \times p_4, m[1,2] + m[3,4] + p_0 \times p_2 \times p_4, m[1,3] + m[4,4] + p_0 \times p_3 \times p_4\}$
$= \min\{0 + 4375 + 30 \times 35 \times 10, 15750 + 750 + 30 \times 15 \times 10, 7875 + 0 + 30 \times 5 \times 10\} = 9375$
In this way the m is filled up. The final result, that is, the optimal number of scalar multiplications required gets stored in $m[1,6]$.

We have discussed the basic steps of the algorithm in the above example. To get $m[i,j]$, we use min function over the set $\{m[i,k] + m[k+1,j] + p_{i-1} \times p_k \times p_j\}$ for $i \leq k < j$. The value of k against the minimum value is the position to split the chain: $(A_i \times \cdots \times A_k) \times (A_{k+1} \times \cdots \times A_j)$. We need another array $s[i,j]$ to store this k. Let us write a pseudo code for Matrix Chain Multiplication. Here, let p denotes dimensions of matrices: dimension of A_i is $p_{i-1} \times p_i$.

```
 1. for $i \leftarrow 1$ to $n$
 2.     do $m[i,i] \leftarrow 0$
 3. for $l \leftarrow 2$ to $n$
 4.     do for $i \leftarrow 1$ to $n - l + 1$
 5.         do $j \leftarrow i + l - 1$
 6.             $m[i,j] \leftarrow \infty$
 7.                 for $k \leftarrow i$ to $j - 1$
 8.                     do $q \leftarrow m[i,k] + m[k+1,j] + p_{i-1} \times p_k \times p_j$
 9.                         if $q < m[i,j]$
10.                             then $m[i,j] \leftarrow q$
11.                                 $s[i,j] \leftarrow k$
```

Major part of this algorithm is line 6 to 10 where we can find the k for which we get $m[i,j]$. In line 3 and 4 we run the loop to get all possible cases.

Time Complexity: If there are k number of nested loops for n number of inputs then the time complexity for most of the cases will be bounded by $O(n^k)$. Here $k = 3$. We see that line 2 is bounded by n, line 3 is bounded by n. Line 4 can run at most n times. So line can run at most n times. Hence the time complexity of this algorithm cannot exceed $c.n^3$ for some constant c. That is, the upper bound of time complexity is $O(n^3)$.

To compute time complexity we are considering the loops. Why the loops are so important? Let there be two nested loops.

```
for ($i = 1$ $to$ $n$)
{
  for ($j = 1$ $to$ $m$)
  {
      ⋮
  }
  ⋮
}
```

Let the time required for the inner loop be c and for the outer loop be c'. Then time complexity is $(cm + c')n = cnm + c'n = \Theta(nm)$.

Space Complexity: For the array $m[i, j]$ we need n^2 locations and for the array $s[i, j]$ we need another n^2 locations. So it is $\Theta(n^2)$ for all the cases - best case, worst case and average case.

Let us now understand the impact of extra space on time complexity. To do so, we shall not store solutions to the subproblems in $m[i, j]$, rather we shall recompute them if required. Let us write a pseudo code for that.

```
Recursive-Matrix-Chain($i$,$j$)
{
      if $i = j$
         then return 0
      $m[i, j] \leftarrow \infty$
         for $k \leftarrow 1$ or $j - 1$
           do $q \leftarrow$ Recursive-Matrix-Chain($i, k$)+Recursive-Matrix-Chain($k+$
$1, j$)+$p_{i-1}p_kp_j$
             if $q < m[i, j]$
                then $m[i, j] \leftarrow q$
      return $m[i, j]$
}
```

Time Complexity: Let $T(n)$ be the time complexity of above algorithm.

$$T(n) \geq \sum_{k=1}^{n-1} [T(k) + T(n-k) + c] + c'$$

Here c' is the time taken by first three lines of the algorithm.

$$\Rightarrow T(n) \geq \sum_{k=1}^{n-1} T(k) + \sum_{k=1}^{n-1} T(n-k) + c.(n-1) + c'$$

$$= 2\sum_{k=1}^{n-1} T(k) + c.n - c + c'$$
$$= 2\sum_{k=1}^{n-1} T(k) + n(c - \tfrac{c}{n} + \tfrac{c'}{n})$$
$$\Rightarrow T(n) \geq 2\sum_{k=1}^{n-1} T(k) + n$$

We shall solve this using substitution method. We shall show that $T(n) = \Omega(2^n)$.
Let us assume that $T(n) \geq 2^{n-1}$ for all $n \geq 1$. Tha basis is easy, since $T(1) \geq 1 = 2^0$. Inductively, for $n \geq 2$, we have

$$\begin{aligned} T(n) &\geq 2\sum_{k=1}^{n-1} 2^{k-1} + n \\ &= 2(2^0 + 2^1 + \cdots + 2^{n-2}) + n \\ &= 2 \times 1 \times \tfrac{2^{n-1}-1}{2-1} + n \\ &= 2 \times (2^{n-1} - 1) + n \\ &= 2^n - 2 + n \\ &\geq 2^{n-1} \end{aligned}$$

It shows that our assumption is correct. Hence, $T(n) = \Omega(2^n)$. So lower bound is exponential. However, our matrix chain multiplication by dynamic programming takes only $O(n^3)$ time, which is dramatically less than this exponential time requirement. This shows the impact of extra space use on time complexity and power of Dynamic Programming.

13.2 Conditions of using Dynamic Programming method

Dynamic programming method can be applied to a problem if the problem shows *optimal substructure* property and it has *overlapping subproblems.*

Optimal Substructure Property: The problem has to show the optimal substructure property. Suppose a problem P is given. It has two subproblems P_1 and P_2 with solutions S_1 and S_2 respectively. Let S be the optimal solution to the problem P that includes S_1 and S_2. Then S_1 is the optimal solution for the problem P_1 and S_2 is the optimal solution for the problem P_2.

For example, Shortest Path Problem shows optimal substructure property. Consider a graph with s as the source and d as the destination. Let u be

a node on the optimal path from s to d. In this case we get two subproblems:
1. s is the source and u is the destination.
2. u is the source and d is the destination.
Then the path from s to u and from u to d must be optimal otherwise the path $s \rightsquigarrow u \rightsquigarrow d$ cannot be optimal. So *Shortest path problem* has the optimal substructure property.

However, *Longest path problem* for undirected graph does not have optimal substructure property. For example, let us take a triangle ΔABC. The longest path between A and B is $A \to C \to B$. But $A \to C$ is not the longest path between A and C.

Overlapping Subproblem: Whenever an algorithm for a problem revisits the same subproblem over and over again to construct its solution, we call that the problem has overlapping substructure. Dynamic programming algorithms typically take advantage of overlapping subproblems by solving each subproblem once and then storing the solution in a table where it can be looked up when needed. In contrast, a problem for which Divide-and-Conquer is suitable, generally visits new subproblems at each step.

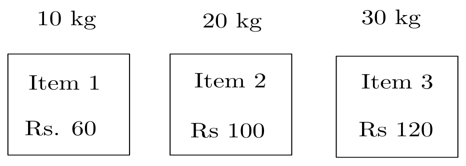
A question may arise here that why is the word *Programming* is used here? What is a Program then? Programs should contain all the properties of algorithms. But the difference is in algorithms we can use any language but for programming we have to use some formal language. Note that programming languages are formal languages but all formal languages are not programming languages. In Dynamic Programming, we follow some specific steps with proper syntax to solve a problem. We use a particular way to fill up the table.So this way of solving an optimization problem is called *programming.*

Chapter 14

Greedy method

A thief is robbing a store. There are n items in the store. The item i is of weight w_i and of cost v_i. The thief has a knapsack which can carry at most W weight. The thief, as usual, wants to maximize the total cost of robbed items, and generally, $W < \sum_{i=1}^{n} w_i$. Question is, how the thief fill up the knapsack? So it is an optimization problem.

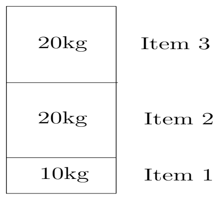
Let us take an example with $n = 3$.



Here $w_1 = 10$kg, $v_1 =$Rs.60;
$w_2 = 20$kg, $v_2 =$Rs.100;
$w_3 = 30$kg, $v_3 =$Rs.120.
Size of knapsack is $W = 50$, i.e. the thief can carry at most 50kg. The thief takes the items one by one. Among the above items which one will the thief like to put into the knapsack?

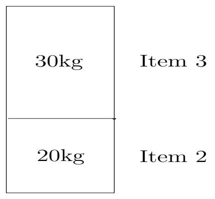
In our example, Item 3 has highest cost and Item 1 has the least cost. But Item 1 is the most valuable item, as its per kg price is Rs. 6. Per kg price of Item 2 and Item 3 are Rs. 5 and Rs. 4 respectively. What the thief does is, the most valuable item is taken as much as possible, then the second most valuable item is taken. And so on. The thief does not explore all possibilities to *maximize* the cost of knapsack. Rather, a *Greedy strategy* is adopted here. The greedy strategy can provide optimal solution to the thief.

10 kg of Item 1, 20 kg of Item 2 are taken first. 30kg of Item 3 is available but the knapsack can accommodate 20kg more. So the thief takes 20kg of Item 3.



Here, total cost =Rs.60 + 100 + 80 =Rs.240

Is this the proper way of approaching the problem? Is the above solution correct? Let the items cannot be broken into parts. Then the above solution is not correct. Then we should take



Then the total cost is Rs.100 + 120 =Rs.220. We are claiming that this is the optimal one. How can we say this?

14.1 Knapsack Problem

These problems are known as knapsack problem. The first problem is called **Fractional Knapsack Problem** and the second one is called **0-1** or **Integer Knapsack Problem**. Here $0-1$ stands for either one item is taken completely or not. taken at all First problem is more natural and our approach to solve the problem is also very appealing. But with that approach we cannot reach to the optimal solution for the second problem. First approach is called Greedy Method.

To solve the second problem we need Dynamic Programming. We consider all possible cases and from there we have to choose the optimal one. In the first problem we sort the items by their unit price in descending order and then we fill up the knapsack by taking items one by one following the sorted order.

We understand that Greedy Method is computationally very good. But if an arbitrary problem is given how can we say that Greedy Method is applicable for it?

To attempt a problem with greedy method, the problem has to satisfy two properties – *Optimal Substructure Property*, which we have discussed during discussion of dynamic programming, and *Greedy Choice Property.* A problem satisfies greedy choice property, if (*globbaly*) optimal solutions can be reached by choosing (*locally*) optimal solutions to the subproblems. In case of fractional knapsack problem, first the most valuable item is chosen, second the most valuable from the remaining items is chosen, and so on. Hence, an item that seems the best choice at that moment (locally optimal solution to a subproblem) is chosen, though we want global optimal solution. This phenomenon is named as Greedy-Choice Property. However, we have to prove first that a given problem really satisfies greedy choice property.

Theorem 8 : *Fractional Knapsack Problem shows Greedy-Choice Property.*

Suppose we have n items and we put the items in the knapsack according to the above mentioned greedy strategy. Our claim is we are able to achieve the optimal solution.

Assume by contradiction that the claim is false, that is, this does not lead to optimal solution. Let item x is the jth valuable item with cost c_j. If our assumption is true, then we can replace Item x by Item y having cost $c'_j < c_j$.

$$\begin{tikzcd} & & c'_j \arrow[dl, bend right] \\ c_1 + c_2 + \ldots + & c_j \arrow[dr, bend right] & \\ & & c_j \end{tikzcd}$$

But, total cost is $c_1 + c_2 + \cdots + c'_j + \cdots$ which is less than $c_1 + c_2 + \cdots + c_j + \cdots$. That means, we get a contradiction. Hence our assumption is wrong. So, Fractional Knapsack Problem shows Greedy-Choice Property.

14.2 Huffman Code

Huffman code was introduced for data compression. It uses greedy method. Consider a text file having some characters with different frequencies of us-age. We want to code these characters so that data of the file can be stored compactly. Let us consider that there are six letters only - a, b, c, d, e, f. If we use *fixed length coding* to code these characters, we need three bits for each character. Following may be a possible coding.

a	b	c	d	e	f
000	001	010	011	100	101

This is better than the ASCII code, as ASCII code uses eight bits per character. However, if frequencies of characters of the file differ, we can get better codes, with variable lengths of codes, which can help to store data in a more compact way. Let us consider the frequencies of the characters as following:

character	a	b	c	d	e	f
Frequency of the characters	600	130	240	95	360	50

For 3-bit long fixed size coding, total bits needed is $(600 + 130 + 240 + 95 + 360 + 50) \times 3 = 1475 \times 3 = 4425$ bits.

Let us now look at *variable length coding schemes*. The most popular variable length coding scheme is *Prefix coding*, where no codeword is a prefix of any other codeword. For example, if '0' is the code for a, then code for any letter cannot start with '0'. This prefix coding can achieve good data compression.

Huffman code is an optimal prefix code which is constructed by a greedy algorithm. The problem of assigning optimal prefix code to characters follows greedy choice property. Huffman's algorithm constructs a binary tree to optimally assign the codes to the characters. Following are the steps of constructing Huffman tree.

1. Form leaves of the tree with each character, and sort the leaves with increasing order of frequencies.

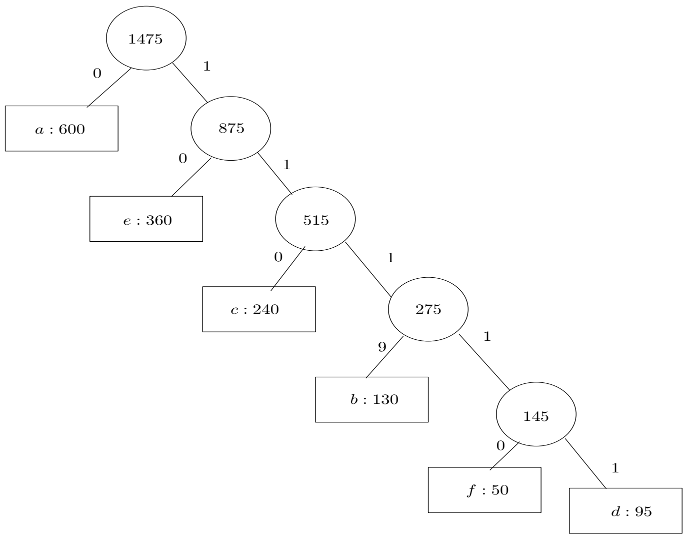
2. Pick up two nodes with minimum frequencies and form an internal node as their parent. The node with lowest frequency is the left child and other is the right child. Th parent contains the sum of two frequencies.

3. Replace the the first two nodes of above step by their parent. Insert the parent in proper position so that the list of nodes remains sorted.

4. Repeat steps 2 and 3 until the list contains a single node. The remaining node is the root node and the tree is complete.

The algorithm can run in $O(n \log n)$ time. After construction of tree, we traverse the tree to give a code to a character, by labelling left edge by 0 and right edge by 1.

Following the above steps, we can form the Huffman tree for the above example.



Here we first form an internal node by considering f as its left child (as its frequency is minimum) and d as right child (as its frequency is second minimum). Then we form another internal node by considering the leaf for b as left child and the newly formed internal node as right child. And so on. After forming the tree, we get the following code for the characters:

a is coded as 0,
e is coded as 10,
c is coded as 110,
b is coded as 1110,
f is coded as 11110,
and d is coded as 11111.

Then total number of bits required is $= (600 + 130 \times 4 + 240 \times 3 + 95 \times 5 + 360 \times 2 + 50 \times 5)$ bits $= 3285$ bits. So there is a significant improvement with respect to data compression, and it is the optimal prefix coding.

Decoding in this scheme is also very simple. Since no codeword is a prefix of any other, the codeword that begins an encoded file is unambiguous. We can simply identify the initial codeword, translate it back to the original character, and the repeat the decoding process on the remainder of the encoded file.

Chapter 15

Amortized Analysis

Amortized analysis is applied on data structure operations that are performed repeatedly. Classical asymptotic analysis gives worst case analysis of each operation without taking the effect of one operation on the other, whereas amortized analysis focuses on a sequence of operations, an interplay between operations, and thus yielding an analysis which is precise and depicts a micro-level analysis. In amortized analysis, all the operations performed are averaged to get an *amortized cost* of the operation. However, it is not an average case analysis.

Let us consider an example of k-bit binary counter which is incremented in an algorithm by following procedure:

```
void     Increment(char* A, k)
{
     int    i;

     for(i=0;i<k;i++)
     {
         if(A[i]==1)      A[i]=0;
         else            break;
     }
     if(i<k)        A[i]=1;
}
```

The worst case scenario for this procedure is that A is filled up with all-1. In that case, time complexity is $O(k)$. If this procedure is called from an algorithm again and again, and we consider its time requirement is $O(k)$ for each call, then that will be an overestimation. And that does not lead us to find the tight bound of time complexity of the algorithm. So we look for amortized cost of *Increment* operation. Following are three ways of finding the cost.

1. Aggregate analysis
2. Accounting method
3. Potential method

15.1 Aggregate analysis

In aggregate analysis, we first find the total cost $T(n)$ for a sequence of n operations, then we divide $T(n)$ by the number n to obtain the amortized cost of the operation. For all operations the same amortized cost $T(n)/n$ is assigned, even if they are of different types. The other two methods may allow for assigning different amortized costs to different types of operations in the same sequence.

Let us consider the above example of binary counter. We observe that number of bit flips in a call *Increment* determines its cost (that is, time requirement). We further observe that the LSB ($A[0]$) is flipped in each call, $A[1]$ is flipped $\lfloor n/2 \rfloor$ times in a sequence of n calls, $A[2]$ is flipped $\lfloor n/4 \rfloor$ times, and so on.

Counter value	A 3 2 1 0	Number of flips
0	0 0 0 0	0
1	0 0 0 1	1
2	0 0 1 0	2
3	0 0 1 1	1
4	0 1 0 0	3
5	0 1 0 1	1
6	0 1 1 0	2
7	0 1 1 1	1
	Total cost	11

Hence, total number of flips in successive n calls with initial value counter values as 0 is

$$\sum_{i=0}^{\lfloor \log_2 n \rfloor} \left\lfloor \frac{1}{2^i} \right\rfloor < n \sum_{i=0}^{\infty} \frac{n}{2^i} = 2n$$

That is, the total cost of n calls is bounded by $2n$, which implies that the average cost of each operation is $O(1)$.

15.2 Accounting method

In this method of amortized analysis, we assign different changes to different operations and maintain an account with the underlying data structure. The amount we charge an operation is the amortized cost of the operation. If it exceeds the actual cost, the difference is stored as *credits*, which is used later for other operation. Let $\hat{c_i}$ and c_i denote the amortized cost and actual cost of ith operation. Then, following is to be maintained.

$$\sum_{i=1}^{n} \hat{c_i} \geq \sum_{i=1}^{n} c_i$$

Let us consider the above example here. Assume that cost of flip from 0 to 1 is 2, and cost of 1 to 0 is 0. When a bit flip from 0 to 1, we use 1 unit of cost, and the rest is stored as credit for later use. We understand that a bit is flipped from 1 to 0, only if it was set earlier.Actually, already stored credit is used here to pay for resetting a bit. Hence, above condition is always maintained.

Now we get that in each call of *Increment*, only once a bit is set (flipped from 0 to 1). So the amortized cost is 2, that is $O(1)$. This analysis even easier than the previous method.

15.3 Potential method

Instead of storing prepaid works as credit, the potential method represents the prepaid work as “potential energy” that can be released to pay the future operations. For an operation, after performing the operation, the change in a structural parameter, such as the number of elements, the height, the number of property violations of a data structure, is captured as a function, called *potential function* which is stored at a data structure. As part of the analysis, we work with non-negative potential functions. If the change in potential is positive, then that operation is over charged and similar to accounting method, the excess potential will be stored at the data structure. If the change in potential is negative, then that operation is under charged which would be compensated by excess potential available at the data structure

The potential function method defines a function ϕ (potential function) that maps a data structure onto a real valued non-negative number. In the potential method, the amortized cost $\hat{c_i}$ of operation i is equal to the actual cost c_i plus the increase in potential due to that operation:

$$\hat{c_i} = c_i + \phi(D_i) - \phi(D_{i-1})$$

where D_i is the data structure at ith operation. Hence we get the total amortized cost as

$$\begin{aligned} \sum_{i=1}^{n} \hat{c_i} &= \sum_{i=1}^{n}(c_i + \phi(D_i) - \phi(D_{i-1})) \\ &= \sum_{i=1}^{n} c_i + \phi(D_n) - \phi(D_0) \end{aligned}$$

Let us consider the same example of *Increment* once again. Here, the structural parameter of interest is the number of 1's in the counter. During ith iteration, all ones after the last zero is set to zero and the last zero is set to 1. For example, when increment is called on a counter with its contents being '11001111', the result is '11010000'. Let x denotes the total number of 1's before the ith operation and t denote the number of ones after the last zero. At the end of ith operation there will be $x - t + 1$ ones, t ones are changed to 0 and the last zero is changed to 1. Thus, the actual cost for the increment is $1 + t$. Therefore, the amortized cost is

$$\hat{c_i} = c_i + \phi(D_i) - \phi(D_{i-1}) = 1 + t + (x - t + 1) - x = 2$$

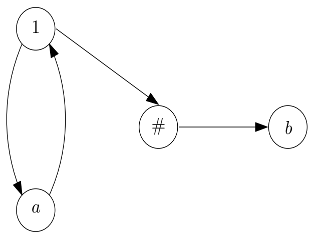
Hence, amortized cost of each *Increment* call is $O(1)$.

Chapter 16

Graphs and traversal algorithms

Graphs are binary relations over a non-empty set. Formally, a graph $G = (V, E)$ is mathematical structure where V is a non-empty set and $E \subseteq V \times V$ is a binary relation over V. V and E are generally called as the set of *vertices* and set of *edges* respectively.

Let us consider $V = \{1, a, \#, b\}$ and a binary relation $E = \{(1, a), (a, 1), (1, \#), (\#, b)\}$. This binary relation is a graph. Generally we love to present a graph by its pictorial presentation. Following is the pictorial presentation of the graph.

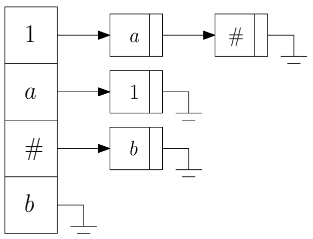


Sometimes graphs are classified as directed and undirected. Since a binary relation is a set of ordered pairs and an ordered pair (x, y) represents 'from x to y', a graph is always directed. If the relation E is symmetric (that is, (x, y) and (y, x) both exist in E), we omit direction from edges. This graph is called undirected graph.

There are two ways of representing a graph in computer - *adjacency matrix* representation and *adjacency list* representation. In matrix representation, a 2D array is used as data structure. Rows and columns of the matrix represent the vertices. For $(x, y) \in E$, (x, y) location of the array is 1. Following is this matrix representation of the graph.

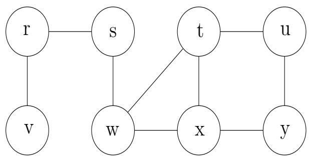
$$\begin{array}{c|c|c|c|c|}
 & 1 & a & \# & b \\ \hline
1 & 0 & 1 & 1 & 0 \\ \hline
a & 1 & 0 & 0 & 0 \\ \hline
\# & 0 & 0 & 0 & 1 \\ \hline
b & 0 & 0 & 0 & 0 \\ \hline
\end{array}$$

In list representation, the adjacent vertices of a vertex are represented by a linked list. Following is the list representation of the graph.



16.1 Breadth First Search(BFS)

BFS is a graph traversal algorithm which systematically explores the nodes of a given graph. We assume that the graphs are taken by their list representation. Let us consider the following graph.

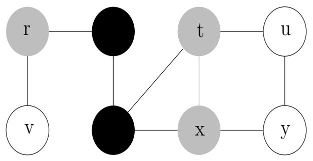


For this algorithm, we consider undirected graph but the scheme is also applicable to any graph. Graphs may also be disconnected, though we have considered here connected graph. We want to explore all the vertices. We can start from any node. Let the start vertex here be s.

For BFS, we explore the nodes *breadth wise*, starting from a given source vertex (here s). Initially all the nodes of the graph are *undiscovered*. Since r

and w are immediate neighbors of s, we can *discover* any one of them. Let's say we discover w first. In next step we do not discover the neighbors of w, but discover r. Whenever r has been discovered, we declare s has been explored, as there is no node to go from s. Next, in a similar fashion, we start discovering the neighbors of w one by one.

Nodes in the algorithm can, therefore, be in three states - (1) UNDISCOVERED, (2) DISCOVERED, and (3) EXPLORED which are identified by three colors - WHITE, GRAY and BLACK respectively. Initially all the nodes are colored as WHITE, which are step by step colored to GRAY, and then BLACK. Following figure shows a snapshot where s and w are EXPLORED, r, x and t are DISCOVERED and the rest remain UNDISCOVERED.



Data structure: Apart from the adjacency list for representing input graph, few more data structure are needed by the algorithm.

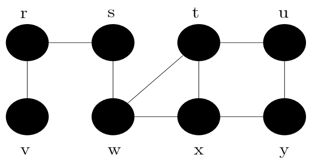
1. A queue (Q) – It stores the nodes just discovered. A node is dequeued from Q to explore it, and when a new node discovered, it is enqueued. FIFO nature of queue ensures that the nodes are explored breadth-wise.

2. Distance array d – It stores the distance from the source vertex.

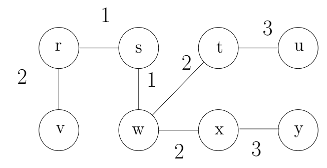
3. Predecessor array π – If we note down the edges of exploring the nodes, we get a tree, called *breadth-first tree*. This tree is stored in π, which notes down predecessor of a node in the tree.

4. State array *color* – It stores the color of each vertex.

The algorithm terminates when Q is empty. That is, when all the nodes that are reachable from the source have been explored, the algorithm stops. Following is the final graph.



The BFS tree along with the distance of nodes from s, is shown below.



BFS(G,s)

```
1.  for each vertex $u \in V[G] - \{s\}$
2.          do color[$u$] $\leftarrow$ WHITE
3.                  $d[u] \leftarrow \infty$
4.                  $\pi[u] \leftarrow NIL$
5.  color[$u$] $\leftarrow$ GREY
6.  $d[s] \leftarrow 0$
7.  $\pi[s] \leftarrow NIL$
8.  $Q \leftarrow \emptyset$
9.  ENQUEUE (Q,s)
10. While $Q \neq \emptyset$
11.         do $u \leftarrow$ DEQUEUE($Q$)
12.             for each $v \in$ Adj[$u$]                  (Adjacent vertex)
13.                 do if color[$v$] = WHITE
14.                 then color[$v$] $\leftarrow$ GREY
15.                     $d[v] \leftarrow d[u] + 1$
16.                     $\pi[v] \leftarrow u$
17.                     ENQUEUE (Q,u)
18. color[$v$] $\leftarrow$ BLACK
```

Complexity Analysis:

Time Complexity: In Steps 1-4 there is a loop that runs for $|V|$ times. So for these steps, we need $O(V)$ time. For Steps 5-9, time requirement is constant. For Steps 10-18 there are two loops. There are both ENQUEUE and DEQUEUE operations. We use here aggregate analysis (see Chapter 15). Observe that we enqueue only undiscovered vertices. So ENQUEUE operation is performed $O(V)$ times. Obviously, if there is no ENQUEUE there is no DEQUEUE. So DEQUEUE operation is also performed $O(V)$ times.

How many times the 'for loop' of Step 12 is executed? Each vertex is dequeued once, and then the adjacency list against the vertex is scanned. Total length of adjacency list is bounded by the number of edges. So the time requirement is $O(E)$ for the loop of Step 12. Hence for Steps 10-18, total time requirement is $O(V+E)$. Combing all the steps, we get the total complexity of the algorithm is $O(V+E)$. (Here '+' denotes the maximum between V and E).

Space Complexity: The algorithm uses a queue, size of which is bounded by $|V|$. Three more array of size $|V|$ are also used by the algorithm. Hence the space complexity is $O(V)$.

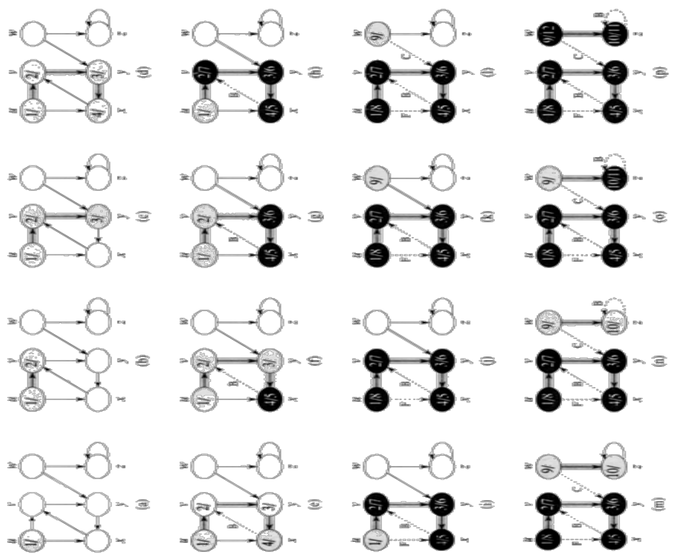
If a vertex v is reachable from s, then it is discovered by the algorithm of BFS. All the nodes which are reachable from s can be explored by BFS. If the graph is unweighted then we can use BFS to get the shortest path of nodes from s.

Theorem 9 : *If the vertex v is reachable from the start vertex s, $d[v]$ stores the shortest path from s to v.*

Proof : Let $\delta(s,v)$ represents the distance of shortest path from s to v. If v is not reachable from s, $\delta(s,v)=\infty$. We want to show $d[v]=\delta(s,v)$ where $d[v]$ is the shortest path. We do it by induction.
<u>Base Case:</u> $s \rightsquigarrow s \Rightarrow \delta(s,s)=0$. In algorithm $d[s]=0$ [by Step 6]. When $v=s$ then $d[v]=\delta(s,v)$.
<u>Induction:</u> Let $(u,v)\in E$. Let the graph is explored up to the vertex u. Hence $d[u]=\delta(s,u)$ [by induction hypothesis]. But v remains undiscovered. Following Step 15 of the algorithm of BFS
$d[v]=d[u]+1=\delta(s,u)+1$.
According to the algorithm, u is discovered first, then is v. So, $\delta(s,u)<\delta(s,v)$. Moreover u is the immediate predecessor of v. So, from Step 15, $\delta(s,v)=\delta(s,u)+1$.
$\therefore$ 1 and 2 gives, $d[v]=\delta(s,v)$. If u is not reachable from s, then $\delta(s,v)=\infty$ and then it is trivially true. Hence, the induction is proved. So, the data structure $d[u]$ contains the shortest distance. □

16.2 Depth First Search (DFS)

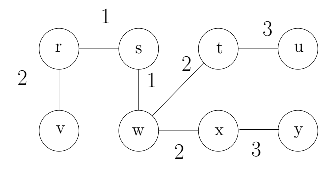
This is another graph traversal algorithm which discovers the nodes *depth-wise*. Normally DFS is used for directed graph and BFS is used for undirected graph. Let the following graph be given



We are taking here directed graph. Normally DFS is used for directed graph and BFS is used for undirected graph. For BFS we go from u to v and x. But for DFS we go $u \to v \to y \to x$. Here, we are going deeper and deeper. There are three types of nodes- UNDISCOVERED=WHITE, DISCOVERED=GREY, EXPLORED=BLACK. Initially all vertices are white.

Let us start from u. So it is GREY now. There are two options which can be reached from u– v and x. Let us choose v. Now from v we go to y. From y we have to go x. Primarily we are discovering the nodes without exploring them. From x we go to v. But there is nothing to discover here. Since from x we have nowhere to go so we can say that x is explored and it becomes BLACK. Then we go to y. But there is no node to discover. So y is explored. Then we go to v and since there is nothing to discover so v is explored. Then we go back to u and then go to x. Since from x we cannot go anywhere, so u is also explored.

Time of discovering and time of exploring are different. We note down the time gap. At first time step we discover u. At time 2 we discover v. Then at time 3 and 4 we discover y and x respectively. Then at time step 5 we understand x is explored; y, v, u are explored at time step 6,7,8 respectively. The path is shown below:



We use data structure d and f to note down the finishing time and π to note down the tree. For BFS, we needs Q, but here we need a stack i.e. it works in the rule 'last in first out'. This is the primary data structure for DFS. We cannot avoid it.

Now, we start from w. We go to y then. But it is explored already. Then we go to z. Then z is explored and finally w is explored. Therefore,
Time Step 9: w discovered
Time Step 10: z discovered
Time Step 11: z explored
Time Step 12: w explored

DFS(G)

```
1. For each vertex $u \in v[G]$
2.         do color$[u] \leftarrow WHITE$
3.             $\pi[u] \leftarrow NIL$
4. time $\leftarrow 0$
5. for each vertex $u \in V[G]$
6.             do if color$[u] = WHITE$
7.                 then DFS-VISIT$(u)$
```

DFS-VISIT(u)

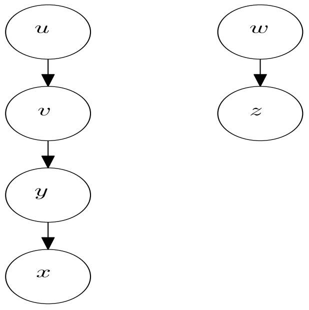
```
1. color$[u] \leftarrow GREY$
2. $time \leftarrow time + 1$
3. $d[u] \leftarrow time$
4. for each $v \in Adj[u]$
5.         do it color$[v] = WHITE$
6.             then $\pi[v] \leftarrow u$
7.                 DFS-VISIT$(v)$
8. color$[u] \leftarrow BLACK$
9. $f[u] \leftarrow time \leftarrow time + 1$
```

Step 5-7 denote if one node is unreachable from starting node then we have to use the for loop. Here system tracking is used. As the nodes are called recursively they are put in a stack.
Time Complexity: For Step 1-3 we need $O(V)$. For Step 5-7 we need $O(V)$ if we do not consider DFS-VISIT(u). For DFS-VISIT(u) we again use aggregate analysis. In Step 4-7 the 'for loop' is executed depending on the total length of the list:
Total length $= \sum_{u\in V} Adj[u] = \Theta(E)$
The number of execution is limited by the number of edges. Number of calling DFS-VISIT= number of undiscovered vertices.
$\therefore$ Total time is limited by $\Theta(V+E)$. So combining Step 5-7 and DFS-VISIT we need $\Theta(V+E)$. Again Step 1-3 need time $O(V)$. Hence, the running time is $\Theta(V+E)$.

Number of vertices does not always indicate number of edges. Number of edges is much higher than number of vertices. But for trivial case $V > E$. So we use V and E both the parameters. If $E = O(V^2)$, it is $\Theta(E)$. But since we do not know the relation, we use both V and E.

16.3 Application of DFS

We may not get a tree in DFS. For example, if the input graph is the previous graph then we get two trees forming a forest.



Using DFS, we can put parenthesis on the graph. We have discovered u, v, y & x and we have explored them in the manner x, y, v, u. So, $(u(v(y(x\ x)y)v)u)$ $(w(z\ z)w)$. This parenthesization is consistent.

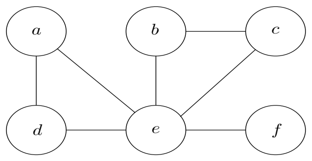
DFS can also be used to find the strongly connected component of a directed graph. Let $G = (V, E)$. Our problem is to find a maximal set $C \subseteq V$

such that $x, y \in C$, $x \rightsquigarrow y \Rightarrow y \rightsquigarrow x$. We solve this using DFS.
Let us take the previous graph. $E = \{(u,v),(v,y),(y,x),(x,v),(u,x),(w,y),(w,z),(z,z)\}$
So, $C = \{v, x, y\}, E_s = \{(v,y),(y,x),(x,v)\}$

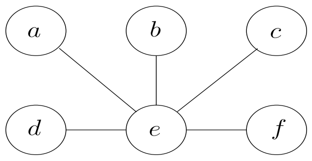
Chapter 17

Minimum Spanning Tree

A Spanning Tree of a graph $G = (V, E)$ is a tree which spans over the graph. Hence, a spanning tree T_Π of G includes all the vertices V and a subset of edges $E_\Pi \subseteq E$ of G. That is, $T_\Pi = (V, E_\Pi)$. However, if G is connected, then only we can get a spanning tree. Let us consider the following graph.



The spanning tree of this graph is



We can get many spanning trees of a given graph. Let us now introduce weight in the graph. We put weight on edges. Hence, now a graph is a triplet $G = (V, E, w)$ where w is a function, $w : E \to \mathbb{R}$. We call this graph *Weighted Graph*. We define weight of a spanning tree as the sum of weights of all edges of the spanning tree. We can now introduce the following optimization problem:

> Given a connected and weighted graph, find a spanning tree of the graph, weight of which is minimum.

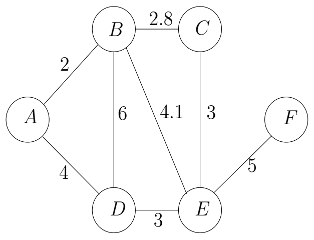
This problem is the *minimum spanning tree* problem. This optimization problem can be solved by Greedy method. This problem shows optimal substructure property. It also has greedy-choice property.

Let a graph G be given against which we are searching for the minimum spanning tree. Let T be a minimum spanning tree of the graph. Now, take a vertex v of T. If we exclude v from the graph then we get a new graph G' which is a subgraph of the original graph G. Let us now remove the vertex v from T to get T'. Will the tree T' be the spanning tree of the new graph G'? Obviously, the answer is 'yes'. So the minimum spanning tree problem has optimal substructure property.

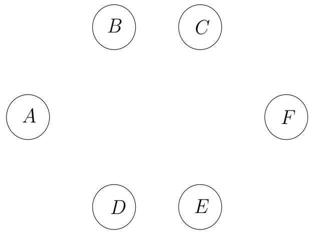
There are two classic algorithms for solving the problem. One is Kruskal's algorithm, the other is Prim's algorithm. Both are greedy algorithms.

17.1 Kruskal's Algorithm

We explain this algorithm with following example. The graph is undirected but weighted. For minimum spanning problem, undirected graphs are generally considered. Kruskal's algorithm says to develop first a forest with all

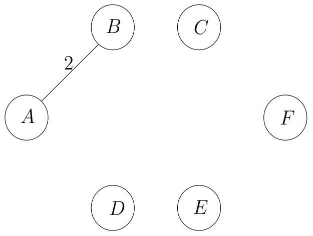


the nodes of the given graph.

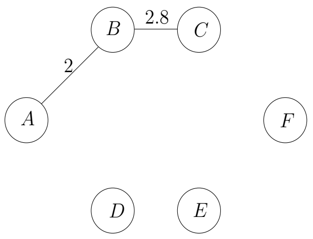


We sort the edges according to their weights in ascending order.Then we take edges from the sorted list one by one to join the individual trees. We exclude an edge to join two trees if the inclusion of the edge introduces a cycle. Finally we get a tree with all the vertices.

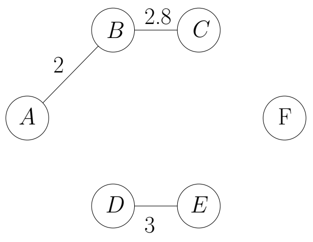
In this example, the minimum weight is 2 and the corresponding edge is AB. This edge is used to join two null trees A and B.



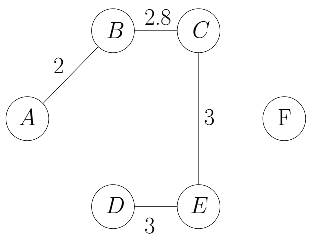
Next minimum is 2.8, which is used to join B and c. However, we cannot join the edges blindly. We have to avoid the cycle formation.



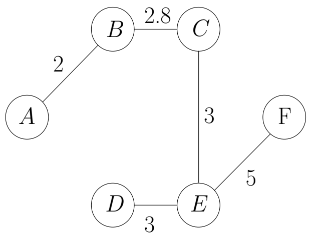
Then minimum value is 3. Here conflict arises. We take the edge DE.



Then again minimum value is 3. So we join that edge.



Then we get 4 as the minimum value for the edge AD. But if we include this edge, a cycle is formed. So we do not use this edge. Same case arises for the edge BE with weight 4.1. So that edge also cannot be a part of the spanning tree. Then next minimum weight is 5.



We stop here as we have received a single tree. The weight of the tree is 15.8, which is the minimum. We have used greedy method here. We have not explored all possibilities to reach to the optimal solution.

Now we have to prove that the tree which we get by above method is the minimal spanning tree. This can be proved by method of contradiction. If the obtained spanning tree is not the minimum one, then there must be another edge which has minimum weight and can be included in the minimum spanning tree. But we have chosen the edges according to their weights in ascending order. Hence, no edge with less weight was left which has possibility to be included in the tree. Hence our assumption is wrong. So, the algorithm gives the minimum spanning tree.

```
MST-KRUSKAL($G$)

1. $A \leftarrow \phi$
2. For each vertex $v \in V[G]$
3.         do MAKE-SET($v$)
4. Sort the edge into non-decreasing order
5. For each edge $(u, v) \in E$, taken in non-decreasing order by weight.
6.         do if FIND-SET($u$)$\neq$ FIND-SET($v$)
7.             then $A \leftarrow A \cup \{(u, v)\}$
8.                 UNION $(u, v)$
9. return $A$
```

Three operations are used here:

1. MAKE-SET

2. FIND-SET
3. UNION

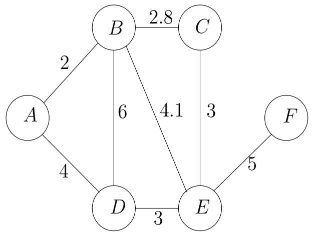
$S_1, S_2, \cdots S_k$ are disjoint sets. Let each set is identified by one representative. For example, 3 can be representative of the set $\{2, A, 3, 9\}$. MAKE-SET creates disjoint sets. FIND-SET(u) checks whether u belongs to a set or not. It returns the representative of the set if u belongs to the set. UNION(u, v) means two sets S_u and S_v (S_u for u and S_v for v) $S_u \cup S_v$ performs. Then we get a new set $S_u \cup S_v$ and it returns the representative of this set.

In Step 3, if there are k nodes, k forests are formed. Then we sort the edges based on their weights. Step 5-7 (these steps are the main steps) we are taking the edges after sorting. In Step 6 let there are two sets S_x and S_y. Let $u \in S_x$ and $v \in S_y$. We do not know whether S_x and S_y are same or not. If FIND-SET operation returns same value then there must be a loop. If they are different then we go to next step.

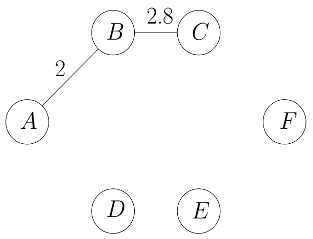
Complexity: Step 5-8 depends on $E + V$. Most costly operation is in Step 4. It takes $O(E \log E)$ to sort the edges. So the complexity is dominated by the sorting operation and it is $O(E \log E)$. But can we write the complexity as $O(E \log V)$? Generally $|E| < |V^2|$ holds i.e. $\log |E| < 2 \log |V| \Rightarrow \log E = O(\log V)$. Hence the complexity becomes $O(E \log V)$.

17.2 Prim's Algorithm

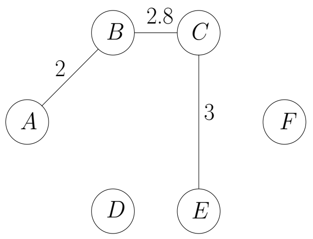
It tells that, instead of starting with forest, start from a vertex of the graph as initial tree, and let the tree grow by including edges one by one. Let us take the previous example again to explain the algorithm.



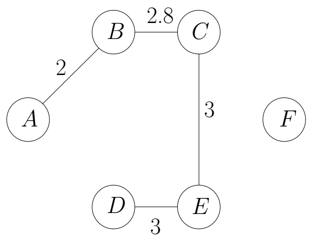
Let us choose A as the starting vertex (the initial tree). We look at all outgoing edges from the initial tree to get the minimum weight edge. Here it AB with weight 2. So we include this edge and the vertex B to grow the tree. Then, the tree has four outgoing edges with weights 4, 6, 4.1 and 2.8. As 2.8 is the minimum, we include BC in the tree to grow it further.



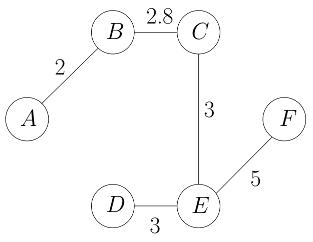
Then we have AD, BD, BE, CE. As 3 is minimum so we choose CE.



Now we have the outgoing edges AD,BD,DE,BE,EF. Again minimum value is 3. So we choose DE.



Then AD has minimum weight. But as it is forming a cycle we do not use this edge. Similar case arises for BE. Then the next minimum value is 5. So we include the edge EF.



We have received the spanning tree of the graph as all the vertices have been covered. The weight of the tree is 15.8. The stopping criteria of the algorithm is the inclusion of all vertices of the graph in the tree. It can be proved like before that the algorithm is correct.

Asymptotically Kruskal's algorithm and Prim's algorithm have same complexity. Since minimum spanning tree of an arbitrary graph is not unique, the algorithms may give different spanning trees.

Chapter 18

Single Source Shortest Path

Let us consider a *weighted graph* $G = (V, E, w)$, where a function $w : E \to \mathbb{R}$ assigns a real number called *weight* to each edge of the graph. Suppose $u, v \in V$ are two vertices of the graph, and let there exists a path $P = (u, v_1, v_2, \cdots, v_k, v)$ from u to v (that is, $u \rightsquigarrow v$, which implies that v is reachable from u). Weight of the path P is $w(u, v_1)+w(v_1, v_2)+\cdots+w(v_k, v)$. The path P is the shortest path if its weight is minimum among all possible paths from u to v. Hence the shortest path problem is to find the shortest path from a *source* vertex to *destination* vertex of a given.

There are a number of variations of the shortest path problem.

1. **Pair-wise shortest path:** Given a pair of vertices $u, v \in V$, find the shortest path from u (called *source* vertex) to v (called *destination* vertex).

2. **Single source shortest paths:** Given a source vertex $s \in V$,find the shortest paths of all possible vertices reachable from s.

3. **Single destination shortest paths:** Given a destination vertex $d \in V$, find shortest paths from all possible nodes to d. This problem is similar to single source shortest paths problem, only difference is that the direction is reversed. Hence the algorithm that solves the first problem can solve this problem with only one extra step to reverse the direction.

4. **All-pair shortest paths problems:** Find the shortest paths from u to v for every pair of vertices u and v. We can repeatedly use the algorithm for single source shortest paths problem to solve this problem.

Among these four variants of the shortest path problem, The Single Source Shortest Paths problem is considered the fundamental one. Although by shortest path problem we intuitively understand the Pair-wise Shortest Path problem, no algorithm is known for Pair-wise Shortest Path problem which runs asymptotically faster than the best algorithm for Single Source Shortest Path problem. In case of Computer Networking, single source shortest paths algorithms are used as routing algorithms.

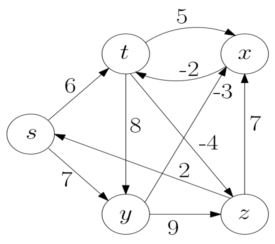
Theorem 10 : *Single Source Shortest Paths problem shows optimal substructure property.*

Proof : Let u,x,y and v be four vertices of a graph such that $u \rightsquigarrow x \rightsquigarrow y \rightsquigarrow v$. Let $P = (u, x, y, v)$ be the shortest path between u and v. Let $w(P) =$ weight of $P = w(u \rightsquigarrow x) + w(x \rightsquigarrow y) + w(y \rightsquigarrow v)$. Here, $w(x \rightsquigarrow y)$ is the weight of path from x to y. When optimal substructure property is followed, then $w(x \rightsquigarrow y)$, $w(x \rightsquigarrow y)$ and $w(y \rightsquigarrow v)$ are also optimal.
Now by contradiction, assume that $w(x \rightsquigarrow y)$ is not the optimal one. This assumption implies that there exists a path from x to y with weight $w'(x \rightsquigarrow y)$ such that $w'(x \rightsquigarrow y) < w(x \rightsquigarrow y)$.
Then $w(P) > w(u \rightsquigarrow x) + w'(x \rightsquigarrow y) + w(y \rightsquigarrow v)$, that is, P is not the shortest path, which is a contradiction. So such w' cannot exist. Hence the problem follows optimal substructure property. □

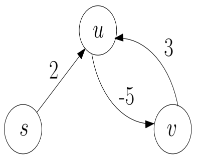
This result indicates that , we can use dynamic programming or greedy method as the problem shows optimal substructure property. We next discuss about *Bellman-Ford Algorithm*, which solves the Single Source Shortest Paths problem.

18.1 Bellman-Ford Algorithm

Let us take the following example. We shall proceed with this example. In general, the edges can assume any real number as its weight.The Bellman-Ford Algorithm allows negative weights on edges.

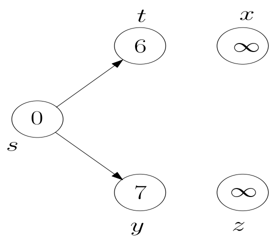


Finding of the shortest paths starts from a given *source* vertex. Let s be the source vertex in this example. However, it may not be always possible to find shortest paths of the vertices of an arbitrary graph from a given source vertex. Let us consider the following graph:

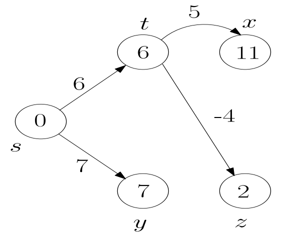


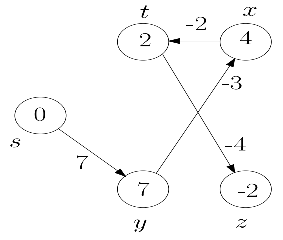
Can we find the shortest path from s to u? If the path is $s \rightarrow u$ then the weight is 2. If it is $s \rightarrow u \rightarrow v \rightarrow u$ or $s \rightarrow u \rightarrow v \rightarrow u \rightarrow v \rightarrow u$ then the weight is 0 and -2 respectively. This signifies that if there is a cycle with negative weight then shortest path cannot be found out. We call that a graph has negative weight cycle if there exists a cycle in the graph so that the sum of the weights of edges in the cycle is negative. In the initially given graph, however, there is no negative weight cycle.
Note: *In graph theory we always consider static nodes. In case of mobile network the nodes are dynamic. It is very difficult to deal with dynamic nodes.*

The Bellman-Ford Algorithm reuses the data structure that we have used for BFS (see page 99). At initialization, 0 is assigned as distance from s to s, and ∞ to all other nodes. So, $d[s] \leftarrow 0$ and $d[u] \leftarrow \infty$ for each u except s. In next step, s *discovers* only t and y, but remain unaware of the rest.



In this step, 6 is minimum distance of t from s. So, the shortest path is $s \rightsquigarrow t$ with weight 6. But it is not the final one. We have to run it further.





We have $s \rightsquigarrow t \rightsquigarrow x = 11$ and $s \rightsquigarrow y \rightsquigarrow x = 7 - 3 = 4$. Since 4 is minimum, we take $s \rightsquigarrow y \rightsquigarrow x$. Again $s \rightsquigarrow t \rightsquigarrow z = 6 - 4 = 2$ and $s \rightsquigarrow t \rightsquigarrow y = 6 + 8 = 14 > 7$. Hence the required shortest path is $s \rightsquigarrow y \rightsquigarrow x \rightsquigarrow t \rightsquigarrow z$. Here each node is computing. It has a flavour of distributed algorithm. If there are n nodes algorithm stops after $n-1$ times.

Bellman-Ford(G,w,s)

```
1.  For each vertex $v \in V[G]$
2.          do $d[v] \leftarrow \infty$
3.             $\pi[v] \leftarrow NIL$
4.  $d[s] \leftarrow 0$
5.  For $i \leftarrow 1$ to $|V(G) - 1|$
6.          do for each edge $(u, v) \in E[G]$
7.             if $d[v] > d[u] + w(u, v)$
8.             then $d[v] \leftarrow d[u] + w(u, v)$
9.                $\pi[v] \leftarrow u$
10. For each edge $(u, v) \in E[G]$
11.         do if $d[v] > d[u] + w(u, v)$
12.            then return FALSE
13. return TRUE
```

Steps 5-9 are the main part of the algorithm. Here, d is the distance. Through the tree π we get the graph. Step 6, 7, 8 tell that the nodes are independent. Step 10-12 are checking whether there is some negative weight cycle. After execution of Step 9, if $d[v]$ is still greater than $d[u] + w(u, v)$ then it indicates that there exists negative weighted cycle. If there is one then there must not be any shortest path and Bellman-Ford Algorithm fails in that case.

Let $P = (v_1, v_2, \cdots, v_k)$ where $v_1 = s$ and $v_k = v$ is the shortest path. Then there are k vertices and $k - 1$ number of edges. Obviously, $k \leq |V|$. During Step 6-8, we are checking each edge. So the algorithm will run $|V| - 1$ times. Number of edges $|(v_i, v_{i+1})| \leq V - 1$. So $w(P) = d[v]$. But if it has negative cycle it will return false. This is because, we have checked all possible paths. After that if we can still reduce the weight then there exists a negative cycle.

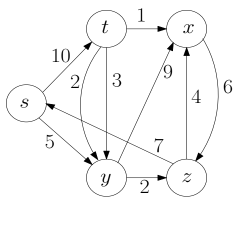
Complexity: For Step 1-4 the required time is $O(V)$.
For Step 10-13 the time requirement is $O(E)$.
For Step 5-9 the time requirement is $O(V \times E)$.
So, the time complexity of the algorithm is $O(V \times E)$. However, the space complexity is $O(V + E)$.

Next section discusses another algorithm which uses Greedy Method to solve the same problem.

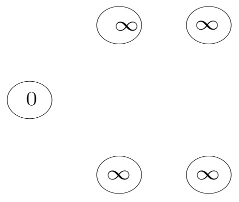
18.2 Dijkstra's Algorithm

Let us take $G = (V, E, w), E \subseteq V \times V, w : E \to \mathbb{R}^+ \cup \{0\}$ where weights are non-negative. Dijkstra's algorithm provides a greedy solution to single source shortest path problem. This algorithm is faster than Bellman-Ford algorithm and is closely related to Breadth First Search. In fact, the procedure is almost similar to BFS.

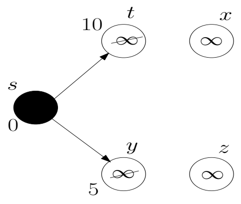
Let us take an example.



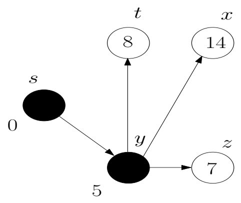
Let s is the source vertex. Initially, we do not know whether any other nodes are reachable from s. So we initialize distance of s as 0 and distance of other nodes as ∞. So,



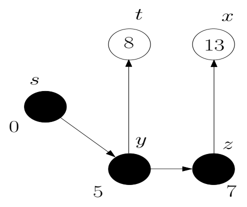
From s, we can reach t and y. When we discover a path from s to t we write distance of t as the weight of s + the weight of the connecting edge, which in this case is 10 and similarly it is 5 for y. Once t and y are discovered, s is explored. We make the explored node as black. In this algorithm we systematically explore all the nodes.



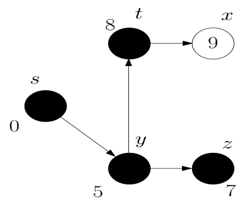
Now we choose the node with minimum distance among all the nodes excluding discovered nodes. So, here we choose y.



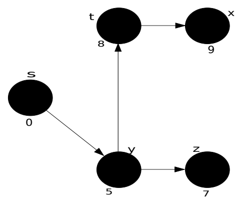
From y we can go to z with distance $5 + 2 = 7 < \infty$ and to x, whose distance is $5 + 9 = 14 < \infty$. We can also go to t with distance $5 + 3 = 8 < 10$. As, this distance to reach t is lesser than t's previous value, we update distance of t as 8 with the path as $s \to y \to t$. Note that, in BFS, we do not reconsider discovered node. But here we do that. So it has some difference from BFS. Now, y is explored. So we make it BLACK.



As z has minimum weight, we choose it. From z we can go to x and distance will be $7 + 6 = 13 < 14$, hence cost of x is updated with new distance. We can go to s also. But then distance will be $7 + 7 = 14 > 0$. So it remains same.



Now we take t and go to x. The distance is $8 + 1 = 9 < 13$. Lastly we take x and it is explored. Finally we get the following tree.



In this way we get the shortest path. Initially the set contains all the nodes. Then the set of explored vertices is increased. When all the nodes are in the set of explored vertices then the algorithm stops. If one node is explored then its cost is final. For example, from t we can come back to y, but we cannot get lesser weight. So, when we get the set of explored vertices we do not have to re-look to those nodes. Here we can use MinHeap (or binary heap) to find the minimum weights. Using heap, searching for minimum weight is minimum.

In original Dijkstra's Algorithm there is no min heap but now we use it for betterment of the algorithm.

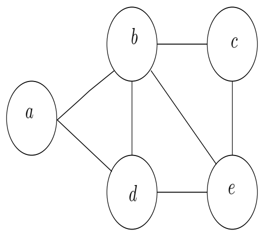
```
DIJKSTRA$(G, w, s)$

1. For each vertex $v \in V[G]$
2.         do $d[v] \leftarrow \infty$
3.            $\pi[v] \leftarrow NIL$
4. $d[s] \leftarrow 0$
5. $s \leftarrow \phi$
6. $Q \leftarrow V[G]$
7. While $Q \neq \phi$
8.         do $u \leftarrow$EXTRACT-MIN$(Q)$
9.            $s \leftarrow S \cup \{u\}$
10.           For each vertex $v \in Adj[u]$
11.              do if $d[v] > d[u] + w(u, v)$
12.                 then $d[v] \leftarrow d[u] + w(u, v)$
13.                 $\pi[v] \leftarrow u$
```

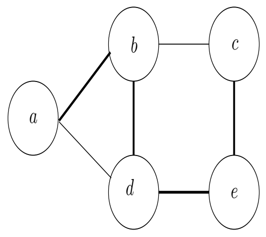
Q is initialized with all the nodes. S = set of explored vertices. In Step 8, we have to extract exactly one vertex. So the algorithm will run for V times. In Step 10, there is a 'for' loop. So the total time taken for Step 10-12 is $O(V)$. Total length of the linked list $\sum_{u \in V} Adj(u) = \Theta(E)$. So Step 10-12 takes $\Theta(E)$ time. If we use binary heap Step 8 will take $\log V$ time. So upper bound for Step 7-12 is $O(E \log V)$

Correctness: Suppose s is explored first with weight 0. Let the algorithm is correct before discovering v where $(u, v) \in E$ and weight of u is known. If we reach x from v and weights are non-negative $d[x] < d[v] + w(v, x)$ as x is explored already. There are two steps. Let we can go back to x(explored) from v. But $d[x] \leq d[v]+w(v, x)$ as weights are non-negative. If v is reachable from u only then $d[v]$ will be min. Let there is another vertex w from where v is reachable. By executing Step 10-13, we can get $d[v]$ which is minimum.

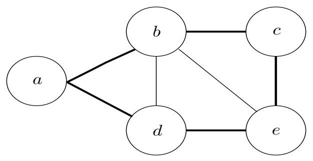
18.3 Hamiltonian Cycle

Hamiltonian cycle is a cycle which covers all the vertices of a graph exactly once. Suppose an undirected graph is given.





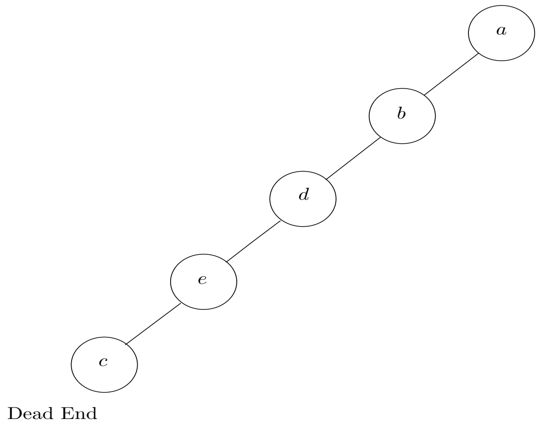
If we choose the above path, shown in bold line, then we cannot get a Hamiltonian cycle.



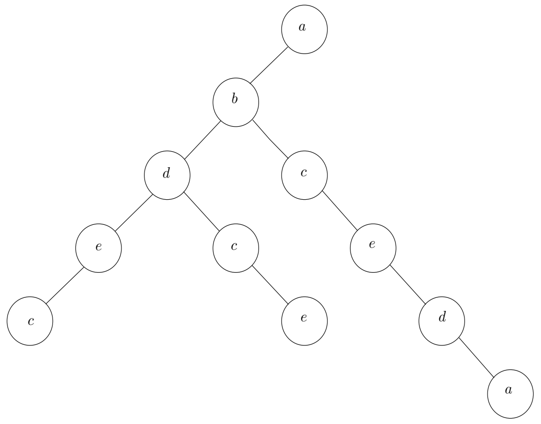
However, the above cycle is a Hamiltonian Cycle. Our problem is to decide the existence of a Hamiltonian cycle in a graph. By exhaustive search we can do it – find all possible permutations and then check if Hamiltonian cycle exists or not. But, apart from this exhaustive procedure, can we make it better?

An algorithm is *efficient* if it takes polynomial amount of time. For the previous procedure, there are $n!$ permutations in worst case. So it takes exponential amount of time.

In DFS, we use backtracking. This problem is close to DFS, so let us see, if we can use this here. Let us choose a. Then go to b. Then we take d, e and c respectively.



We have covered all the vertices and $c \neq a$. We cannot go to a as it is already discovered.



If we choose c instead of e then again we go to a dead end. But if we choose the path $a - b - c - e - d - a$ we succeed.

Backtracking is better than exhaustive method. Worst case is when the graph has no Hamiltonian Cycle. If we consider the tree, there are n children and so here also the complexity is exponential.

18.4 Travelling Salesman Problem(TSP)

Here we take an weighted graph. The problem is to find the Hamiltonian Cycle with minimum weight. This is an optimization problem. Backtracking may not be appropriate here. So for this we can use *Branch-and-Bound*.

Chapter 19

Limit of Computing and NP-Completeness

Given an arbitrary problem, can we develop an algorithm to solve the problem? To get an answer of this question, let us first understand the possible number of problems that can exist, and the maximum possible number of algorithms.

19.1 Problems and algorithms

A problem (P) can be seen as a binary relation from a set of possible inputs (I) to possible outputs (S). We can enumerate the inputs as natural number, such as $0, 1, 2$, etc. Since in a digital computer an input is given as a sequence of 0s and 1s, one may interpret the sequence as a natural number. Outputs can similarly be interpreted as some natural numbers.

Let us now consider a set $\mathbb{U}_I$ as the universe of inputs where from any problem takes input. That is, $\mathbb{U}_I \subseteq I$. Obviously, $\mathbb{U}_I$ is a countably infinite set. Similarly, the universe of possible outputs $\mathbb{U}_S$ is also a countably infinite set. Hence, $|\mathbb{U}_I| = |\mathbb{U}_S| = |\mathbb{N}| = \aleph_0$ ($\aleph_0$ is assumed as cardinality of the set of naturals).

Therefore, we can write $P \subseteq \mathbb{U}_I \times \mathbb{U}_S$. Hence, possible number of problems is limited by the possible number of subsets of $\mathbb{U}_I \times \mathbb{U}_S$. Possible number of subsets is $2^{|\mathbb{U}_I \times \mathbb{U}_S|}$, which is the cardinality of the power set of $\mathbb{U}_I \times \mathbb{U}_S$. However, $\mathbb{U}_I \times \mathbb{U}_S$ is also a countably infinite set. That is, $|\mathbb{U}_I \times \mathbb{U}_S| = \aleph_0$. From Cantor's theorem we know that cardinality of the power set of $\mathbb{U}_I \times \mathbb{U}_S$

is strictly greater than the cardinality of the set.

Let us consider $2^{\aleph_0} = \aleph_1$. That is, $\aleph_1$ is the cardinality of a set which is strictly larger than the set of naturals. It is known that cardinality of the set of real numbers ($\mathbb{R}$) is strictly higher than set of naturals[I]. Hence, maximum possible number of problems is equal to cardinality of $\mathbb{R}$.

Let us now find out maximum possible number of algorithms that can exist. Possible number of algorithms is bounded by the number of possible strings in any language. Because, an algorithm can be seen as a string of symbols of a language which is used to write an algorithm. However, the set of possible strings in any language is a countably infinite set. Hence, the maximum number of possible algorithms is $\aleph_0$.

Now, the point is, $\aleph_0$ is really very tiny with respect to $\aleph_1$. In fact, $\frac{\aleph_0}{\aleph_1} \approx 0$.This scenario answers our question – whether all problems are solvable by some algorithms. The answer is 'NO'. That is, almost all problems are not solvable through algorithms, hence are computationally unsolvable. However, we are lucky to find that many of real-life problems are solvable!

Halting problem is an example of unsolvable problem. However, we do not use the word 'unsolvable' in the theory of computing. We use 'undecidable' in place of 'unsolvable'.

19.2 Decidable problems and decision problems

We can write algorithms for decidable problems. There are different types of decidable problems. Some of the important types are the following.

1. *Decision problems:* In this class of problems, the output is either 'yes' or 'no'. Searching problem is a decision problem. Similarly, whether a given number is prime is a decision problem.

2. *Counting problem:* The desired output of this class of problem is a natural number. An example problem is, given a number, find the total number of distinct factors of the number.

[I]This we get from *Continuum Hypothesis*, which says – there is no set whose cardinality is strictly between that of the integers and the real numbers. However, the set of naturals, set of integers, set of even numbers, set of rational numbers, etc. have same cardinality

3. *Optimization problem:* This class of problems requires optimization of some objective function based on the problem instance. For example, finding of minimum spanning tree of a given (connected) graph is an optimization problem.

4. *Permutation problem:* A permutation of the input is the desired output in this class of problems. Sorting problem is an example of permutation problem.

However, we can transpose a problem of a type to its equivalent problem of another type. An optimization problem can always be transposed to its equivalent decision problem. For example, minimum spanning tree problem can be written as following:

> Given a connected graph, decide if its minimum spanning tree has weight less than k.

This is decision version of the original optimization problem. Both the problems ask to find minimum spanning tree, but the second one wants to additionally check if the final tree has weight less than k. Obviously, the decision problem is no more harder than the optimization problem.

In the similar fashion, any problem can be transposed into its equivalent decision problem with (almost) same complexity. So when we classify the decidable problems depending on their complexity, we consider only decision problems.

19.3 Decision problems and languages

Any decision problem can be interpreted as a language over some alphabet. For example, we can interpret the Searching problem as the following language.

Suppose we want to search a number in a set of natural numbers. If we search, for example, 3 in {5, 10, 2, 3, 15}, we will succeed. That is, the answer is 'yes'. But if we search 6 in the same set, the answer is 'no'. We can represent these inputs by concatenating the search key and the given list:

$$3\{5, 10, 2, 3, 15\} \quad \text{and} \quad 6\{5, 10, 2, 3, 15\}$$

We include those representation of inputs as words in the language for which the expected output is 'yes'. Hence, $3\{5, 10, 2, 3, 15\}$ is a word of the language. Therefore, the language for the searching problem is

$$L = \{xS \mid S \subseteq \mathbb{N} \ \wedge \ x \in S\}$$

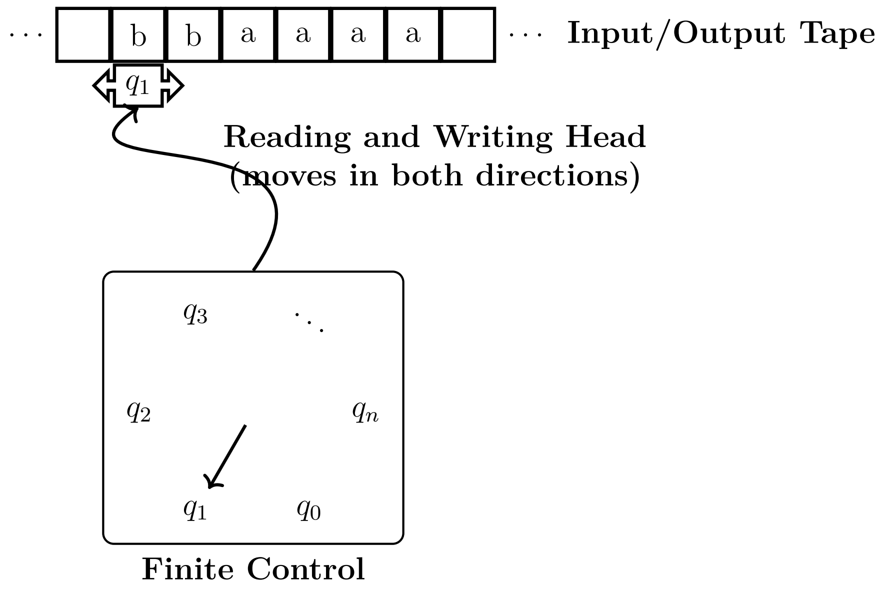
Observe that we have included all possible inputs of the searching problem in L for which the desired output is 'yes'. Hence, this language can represent all aspect of the searching problem. This shows us that we can treat a problem by a formal language. We can now define decidability of a language (hence a problem) as following:

Definition 8 *A language L is decidable if there exists an* effective procedure *that decides whether an element x is in L or not.*

By *effective procedure*, we mean here that an algorithm exits and x is the input to the algorithm. If the output of the algorithm is 'yes', we consider that the input x is in L; otherwise, x is not in L. More formally, existence of an effective procedure implies the existence of a Turing Machine that accepts the language L.

19.4 Deterministic and non-deterministic Turing machines

A Turing machine consists of an infinite tape, a read/write head that can read the content of a box on the tape and write to the box, an arm that holds the read/write head and that can move to left and right box, and a finite control that controls the movement and reading/writing of the head.



Formally, a Turing machine (M) is a 7-tuple $M = (Q, T, I, \delta, b, q_0, F)$ where Q is the set of possible states of the machine, T is the set of tape symbols, $I \subseteq T$ is the set of input symbols, b represents the blank space, and $q_0 \in Q$ is the initial state whereas $F \subseteq Q$ is the set of final/accepting states. However, the most important part of the machine is its *transition function* δ, which defines the transition of the machine from one state to another. The transition function takes the present state and the symbol under the read/write head and produces the next state along with the symbol to be written on the box and the instruction to move the head to the left (L), to the right (R) or to stay on its old position (S).

Classically, a Turing machine is deterministic. The transition function

$$\delta : Q \times I \to Q \times (T \cup \{L, R, S\})$$

is generally expressed in tabular format as shown bellow.

Present		Next		
State	Input	State	Type Symbol	L/R/S

Since for any algorithm there is a Turing machine, we need to express the algorithm through the above mentioned tabular form to get the Turing machine. The idea of single-tape Turing machine has been extended to *multi-tape* Turing machine for the purpose of analysis of algorithms. This machine has $k \geq 1$ tapes which are infinite to the right side only. For any multi-tape Turing machine, it is possible to design an equivalent single-tape Turing machine. Following is an important result which relates complexities of an algorithm implemented on RAM model and (multi-tape) Turing machine.

Theorem 11 : *If a RAM program accepts a language L in $T(n)$ time under logarithmic cost criteria, then the same language can be accepted by multi-tape Turing machine in $O(T^2(n))$ time.*

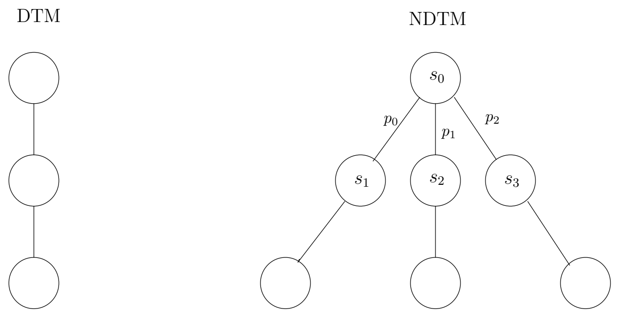
Therefore, if the time complexity of an algorithm is found as polynomial, the time complexity will remain as polynomial even the algorithm is implemented on Turing machine.

Let us now introduce non-deterministic Turing machine (NDTM) which will be used to classify the problems depending on their complexities. In case of NDTM, everything but δ is same with deterministic Turing machine (DTM). Unlike DTM, NDTM can move from a given present state and tape symbol to one of the several possibilities. This makes the machine non-deterministic, because we do not know in advance which possibility the machine will assume during a move. Hence, the transition function of a single-tape NDTM takes the following form:

$$\delta : Q \times I \to \mathscr{P}(Q \times (T \cup \{L, R, S\}))$$

where $\mathscr{P}(.)$ denotes the power set.

Therefore, from one state, an NDTM has more than one possibility to go. For DTM, we have a single *path* from the initial state to a final state of the machine, whereas we get a tree for NDTM that shows all possible options of movement starting from initial state.



An input x is deemed accepted if at least one sequence of moves with x as input leads to an accepting state of the machine. In the above tree, there may exist more than one sequence of moves that lead to accepting state. However, we consider the sequence with minimum number of moves while we find the time required to accept the input x.

Definition 9 *We say that an NDTM is of time complexity $T(n)$ if for an accepted input of size n, $T(n)$ is the minimum number of moves from initial state to an accepting state.*

DTM is a special case of NDTM, where the machine can move from one state to exactly one state. A language that is accepted by a NDTM is also accepted by some DTM. But that DTM demands more time and space to accept the same language.

Algorithms that we have studied till now are deterministic in nature. However, the NDTM induces the idea of non-deterministic algorithms. Let us now introduce a non-deterministic algorithm for the Searching problem to understand reduction in complexity.

If a list of n records along with a search key is given, then our non-deterministic search algorithm can start searching from any of the n records. If the record against the search key exists in the list, one of the start points is the target record. If the record does not exist, no such match will be found. Following are two steps of this algorithm.

Step 1: Pick up (non-deterministically) an index i from $\{1, 2, \cdots, n\}$.
Step 2: If $K_i = K$, output 'yes'.

Above two steps can be placed in a loop. Then, we may need arbitrary long time to reach to an accepting state. However, time complexity of the algorithm is $O(1)$, since we can non-deterministically choose the correct answer in one step (see Definition 9). So a non-deterministic algorithm can run much faster than its deterministic counterpart.

Theorem 12 : *Given an NDTM, it is always possible to derive an equivalent DTM. Furthermore, a constant c exists such that, for any input x, if NDTM accepts x in t time steps, then the DTM accepts x in, at most, c^t steps.*

This result indicates that, if an NDTM has polynomial time complexity then the equivalent DTM which simulates the NDTM can take exponential time.

The intuitive reason of this result is, the equivalent DTM needs to visit all the possible options before it halts.

19.5 The complexity class P and NP

We now classify the problems (hence languages) depending on the time complexities of the best available algorithms for the problems. We call an algorithm *efficient* if its worst-case time complexity is polynomial.

Definition 10 *An algorithm is efficient if the time taken by it in worst case is $O(n^k)$ for some $k \in \mathbb{N}$. Here, n is the size of input.*

Interestingly, we are ready to call an algorithm efficient even if it demands huge, say $O(n^{100})$ time. Important thing here is, whether the algorithm demands polynomial time or beyond polynomial time. Based on this, we extract a set of problems, called as P problems or P-time problems from the pool of decidable problems. For these problems, at least one efficient algorithm exists. For example, Searching problem, Sorting problem, Matrix-Chain Multiplication problem etc. are examples of P problems. We can define now this class of problems (languages) more formally:

Definition 11 **P**$= \{L \mid$ there exists a DTM that accepts $w \in L$ in $O(n^k)$ time, where $n = |w|\}$

Let us now turn our attention to NDTM. Like the above, we extract a set of languages that are accepted by some NDTM in polynomial time. This set of languages (hence problems) are named as NP-time (Non-deterministic Polynomial time) problems.

Definition 12 **NP**$= \{L \mid$ there exists an NDTM that accepts $w \in L$ in $O(n^k)$ time, where $n = |w|\}$

This complexity class (NP) is sometime alternatively defined without referring to NDTM. This alternative definition introduces the concept of *verifiability*. It says that, if a possible solution is given then, whether the solution is correct can be verified in polynomial time. Hence, according to this definition, a class of languages is NP if and only if there exists an algorithm that verifies whether a give sting is a word of the language in polynomial time.

However, "verifiability in polynomial time" and "acceptance by an NDTM in polynomial time" are two equivalent concepts. If a sequence of moves from initial state to accepting state is given and if the length of the sequence is polynomial (see Definition 9), then the sequence can also be verified in polynomial time. Similarly, if a solution is verifiable in polynomial time then there exists a sequence of moves from initial state to accepting state in an NDTM of polynomial size.

From any of these two definition, we get that a problem is in P is also in NP. Hence $P \subseteq NP$. However, the reverse is not yet known. In fact, it is an open question to answer whether $P = NP$ or not. Though, many believe that $P \neq NP$.

19.6 NP-Completeness

A subclass of NP, named *NP-Complete* languages has been extracted from NP which are known as the most difficult problem of NP. Existence of NP-Complete language is sometime thought as an evidence that $P \neq NP$. To define NP-Complete language, we next define the concept of *Reducibility*.

We say that a language L_1 is reducible to another language L_2 if there exists an algorithm which transforms the words of L_1 to the words of L_2. If the algorithm runs in polynomial time, we call that L_1 is polynomially reducible to L_2. This is represented as $L_1 \leq_P L_2$.

Definition 13 *A language L_1 is polynomially reducible to L_2, ($L_1 \leq_P L_2$) if and only if there exists a polynomial-time algorithm which converts each $w \in L_1$ to $w_0 \in L_2$.*

If $L_1 \leq_P L_2$ and if L_2 is acceptable in polynomial time (say, by a DTM), then L_1 is also acceptable in polynomial time (by some DTM). That is, L_1 is not more than a polynomial factor harder than L_2.

Definition 14 *A language L is NP-Complete (NPC) if and only if:*

1. $L \in NP$
2. $L' \leq_P L$ *for every* $L' \in NP$

Intuitively, there exists no problem in NP which is more than a polynomial factor harder than an NP-Complete problem. So, if an NP-Complete problem can be solved by a polynomial-time algorithm, then any problem of NP can be solved by a polynomial-time algorithm. However, no polynomial-time algorithm for an NP-Complete is invented till date.If we get an algorithm some day, that would imply that any problem in NP can be solved by some polynomial time algorithm. This would mean that $P = NP$.

The first problem that was proved as NP-Complete (by Stephen Cook in 1971) is the Boolean Satisfiability Problem, popularly called as SAT problem. The problem is:

SAT Given a Boolean formula, decide if there exists an assignment (TRUE or FALSE) to the variables such that the formula evaluates to TRUE.

For example, the formula $(P_1 + P_2).P_3$ is *satisfiable* because it is true for some assignments, say $P_1 = 0$ (FALSE), $P_2 = P_3 = 1$ (TRUE). Another formula $P_1.\overline{P_1}$ is not satisfiable. Now to show SAT as NP-Complete, we need to show first that the SAT is in NP, and next we have to prove that an arbitrary problem in NP can be polynomially reducible to SAT.

Theorem 13 : *SAT is NP-Complete.*

Proof : $\underline{\text{(1) SAT} \in \text{NP:}}$ Given a Boolean expression of length n, any assignment of Boolean values to Boolean variables that is claimed to satisfy the given expression can be verified in polynomial. In other words, an NDTM can guess a truth assignment in polynomial time. Hence, SAT is in NP.
$\underline{\text{(2) For any language } L' \in \text{NP}, L \leq_P \text{SAT :}}$ Let M be an NDTM which accepts a string w in polynomial time. That is, the language, accepted by the NDTM is in NP. Consider that $n = |w|$ and the time complexity of M is $p(n)$, a polynomial. Suppose that M uses $q_1, q_2, \cdots, q_s$ states with q_1 as the initial state and q_s as an accepting state, and $X_1, X_2, \cdots, X_m$ tape symbols.

Let us now construct a *computation history* of M as shown below. In the computation, therefore, M uses at most $p(n)$ cells.

In the above computation history, the cells used in the computation are shown horizontally, whereas the time steps are plotted vertically. In the initial configuration, some cells are used for state and input symbols (which are also tape symbols) and the rest cells are blank. Name the history as TABLE where TABLE$[i][j]$ denotes the content of j^{th} cell on the tape at step i. Let us now introduce following propositional variables, which will form a Boolean expression w_0.

	Tape					
Steps	1	2	3	4	$\cdots$	$p(n)$
1	#	q_1	X_1	X_2	$\cdots$	#
2						
3						
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$p(n)$		q_s				

1. $C_{i,j,t} = 1$ iff TABLE$[t][i] = X_j \quad 1 \le i, t \le p(n)$
   That is, i^{th} cell on M's tape contains the tape symbol X_j at time t.

2. $S_{k,t} = 1$ iff M is in state q_k at time t.

3. $H_{i,t} = 1$ iff the tape head scanning tape cell i.

Hence, we can get $O(p^2(n))$ propositional variables in total considering the above computation history. Let us now construct a Boolean expression (w_0) using these variables such that the expression reflects an accepting computing of M. To do so, consider the following predicate which is 1 when exactly one of its argument is 1.

$$U(x_1, x_2, \cdots, x_r) = (x_1 + x_2 + \cdots + x_r)(\prod_{i \neq j}(\overline{x_i} + \overline{x_j}))$$

Here, length of U is $O(r^2)$.

Now, in the computing history of M, we get the following.

1. The tape head is scanning exactly one cell in each step.

2. Each step has exactly one tape symbol in each tape cell.

3. Each step has exactly state.

4. At most one tape cell, the cell scanned by the tape head, is modified from one step to the next.

5. The change in state, head location and tape cell contents between successive steps is allowed by the transition function of M.

6. The first step is the initial configuration where M is in q_1 and the tape contains exactly w followed by blanks symbols.

7. The machine M is in accepting state at the last step.

We can now construct Boolean expression from the above points. From point 1, we get A_t for step t, and so we get

$$A = A_1.A_2.\cdots.A_{p(n)} \qquad \text{where} \;\; A_t = U(H_{1,t}, H_{2,t}, \cdots, H_{p(n),t})$$

Length of A is $O(p^3(n))$. From point 2, we get the predicate B_{it} at time t for cell i, and so we get

$$B = \prod_{i,t} B_{it} \qquad \text{where} \;\; B_{it} = U(C_{i,1,t}, C_{i,2,t}, \cdots, C_{i,m,t})$$

Length of B is $O(p^2(n))$. From point 3, we can get

$$C = \prod_{1 \le t \le p(n)} U(S_{1,t}, S_{2,t}, \cdots, S_{s,t})$$

length of which is $O(p(n))$. From point 4, we can similarly get that

$$D = \prod_{i,j,t} [(C_{i,j,t}.C_{i,j,t+1} + \overline{C_{i,j,t}}.\overline{C_{i,j,t+1}}) + H_{i,t}]$$

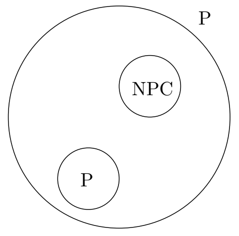
This means, either j^{th} symbol of cell i remains same at $t+1$ time step, or the tape head is scanning cell i at t.

In this way, we can get predicates for the above properties of our computing history. Finally we get a predicate for the accepting state (point 7). Now if we make a conjunction of all these predicates, we get a Boolean expression, say w_0. From our above construction, we can say that the Boolean formula w_0 is satisfiable if and only if M accepts w.

Hence, an arbitrary word of the language that M accepts can be reduced to a word of SAT. Since $p(n)$ is polynomial, length of w_0 is polynomial and the construction of w_0 takes polynomial time.Therefore, the language accepted by M is reducible to SAT in polynomial time. Since M is arbitrary, any language of NP can be reduced to SAT in polynomial time.This completes the proof. $\square$

The above proof points out that if we get an efficient algorithm for SAT, then to solve any problem in NP, we need to construct a Boolean expression from the input of the given NP. This construction will take polynomial time. Then, we can use the efficient algorithm to decide acceptability of the Boolean expression. However, if we don't get an efficient algorithm, but try to design a DTM then the DTM can take exponential time to accept the language (see Theorem 12).

After the proof of SAT as NP-Complete, many problems were shown as NP-Complete. The way of showing a problem as NP-Complete is – first show that the problem is in NP and then show that SAT (or any already-proven NP-Complete problem) is reducible to the given problem. Some well-known NP-Complete problems are the Clique problem, Hamiltonian cycle problem, Travelling Salesman problem, etc. As already said, these problems are treated as the most difficult problems in NP. Hence, NP-Complete $\subset$ NP. Since no efficient algorithm for an NP-Complete problem has been invented and there is little hope to invent so, people believe that $P \subset NP$. Following is the widely-believed relationship among P, NP and NP-Complete languages.



Apart from the NP-Complete problems, more difficult decidable problems exist.There are some problems which are called NP-hard. If all the problems of NP are reducible to a problem L in polynomial time which is not necessarily in NP, then L is called NP-hard. There is another class of problems called PSPACE-Complete problems, which are believed as more difficult than NP-Complete problems. These problems are not tractable and demands exponential time by the best algorithms, available presently.