Dr. Ambedkar Memorial Institute of Information Technology & Management Science

LECTURE NOTES

ON

Design and analysis of algorithm

MCPC2001 MCA (3rd Sem)

Prepared By

Prof. Pujarani Nanda

Module-1

Introduction to design and analysis of algorithms

What is an algorithm?

Algorithm is a set of steps to complete a task.

What is Computer algorithm?

''a set of steps to accomplish or complete a task that is described precisely enough that a computer can run it''.

A **computer algorithm** is a set of instructions that is:

- **Precisely defined** so a computer can execute it
- Designed to accomplish a specific task

Characteristics of an algorithm:-

Property	Description
Input	Takes one or more inputs
Output	Produces at least one result (value, yes/no, etc.)
Definiteness	Each step is clearly and unambiguously defined
Finiteness	Algorithm completes after a finite number of steps
Effectiveness	Each instruction is simple and can be carried out easily

Expectation from an algorithm

- **Correctness:-**

- Must give the correct output for all valid inputs
- Some algorithms are **probabilistic**, giving correct answers most of the time
    - **Example**: Rabin-Miller Primality Test (used in RSA encryption)
        - May give a wrong result **1 out of 250 times**, but still acceptable in practice
        - Approximation algorithm: Exact solution is not found, but near optimal solution can be found out. (Applied to optimization problem.)
    - **Less resource usage:**

      Algorithms should use less resources (time and space).

      Resource usage:

      Here, the time is considered to be the primary measure of efficiency .We are also concerned with how much the respective algorithm involves the computer memory.But mostly time is the resource that is dealt with. And the actual running time depends on a variety of backgrounds: like the speed of the Computer, the language in which the algorithm is implemented, the compiler/interpreter, skill of the programmers etc.

So, mainly the resource usage can be divided into: 1.Memory (space) 2.Time

Time taken by an algorithm?

- performance measurement or Apostoriori Analysis: Implementing the algorithm in a machine and then calculating the time taken by the system to execute the program successfully.
- Performance Evaluation or Apriori Analysis. Before implementing the algorithm in a system. This is done as follows

    1. How long the algorithm takes :-will be represented as a function of the size of the input.

    f(n)→how long it takes if 'n' is the size of input.

    2. How fast the function that characterizes the running time grows with the input size.
    "Rate of growth of running time".

The algorithm with less rate of growth of running time is considered better.

Growth of Functions (Asymptotic notations)

Before going for growth of functions and asymptotic notation let us see how to analyase an algorithm.

How to analyse an Algorithm

Let us form an algorithm for Insertion sort (which sort a sequence of numbers).The pseudo code for the algorithm is give below.

Pseudo code:

for j=2 to A length---------- C_1

key=A[j]----------C_2

//Insert A[j] into sorted Array A[1.....j-1]---------- C_3

i=j-1----------C_4

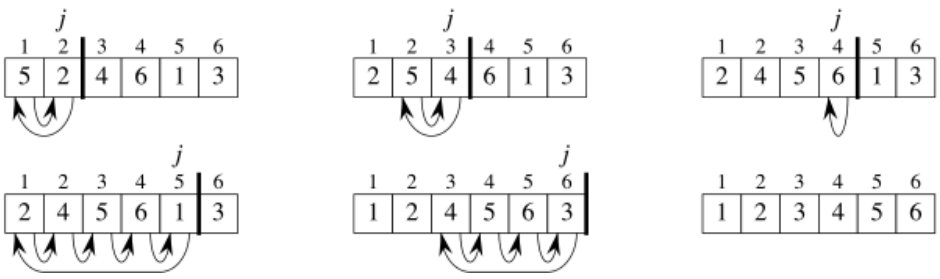
while i>0 & A[j]>key----------C_5

A[i+1]=A[i]----------C_6

i=i-1----------C_7

A[i+1]=key----------C_8

Example:



Let C_i be the cost of i^{th} line. Since comment lines will not incur any cost

C_3=0. Cost No. Of times Executed

C_1n

C_2 n-1

C_3=0 n-1

C_4n-1

$C_5\sum_{j=2}^{n-1} t_j$

$C_6\sum_{j=2}^{n}(t_j - 1)$

$C_7\sum_{j=2}^{n}(t_j - 1)$

C_8n-1

Running time of the algorithm is:

$T(n)=C_1n+C_2(n-1)+0(n-1)+C_4(n-1)+C_5(\sum_{j=2}^{n-1} t_j)+C_6(\sum_{j=2}^{n} t_j - 1)+C_7(\sum_{j=2}^{n} t_j - 1)+ C_8(n-1)$

Best case:

It occurs when Array is sorted.

Example: We found out that for insertion sort the worst-case running time is of the form $an^2 + bn + c$.

Drop lower-order terms. What remains is an^2.Ignore constant coefficient. It results in n^2.But we cannot say that the worst-case running time T(n) equals n^2 .Rather It grows like n^2 . But it doesn't equal n^2.We say that the running time is Θ (n^2) to capture the notion that the order of growthis n^2.

We usually consider one algorithm to be more efficient than another if its worst-case running time has a smaller order of growth.

Asymptotic notation

- It is a way to describe the characteristics of a function in the limit.
- It describes the rate of growth of functions.
- Focus on what's important by abstracting away low-order terms and constant factors.
- It is a way to compare "sizes" of functions: O≈ ≤

Ω≈ ≥

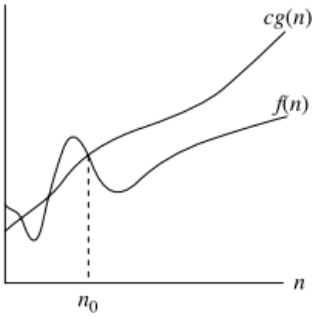
Θ ≈ =

o ≈ <

ω ≈ >

O-notation

$O(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that
$0 \le f(n) \le cg(n)$ for all $n \ge n_0\}$.



$g(n)$ is an ***asymptotic upper bound*** for $f(n)$.

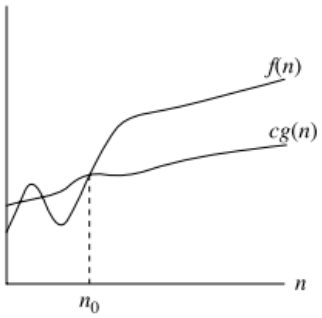
Example: $2n^2 = O(n^3)$, with $c = 1$ and $n_0 = 2$.

Examples of functions in $O(n^2)$:

n^2
$n^2 + n$
$n^2 + 1000n$
$1000n^2 + 1000n$
Also,
n
$n/1000$
$n^{1.99999}$
$n^2/\lg\lg\lg n$

Ω-notation

$\Omega(g(n)) = \{f(n) :$ there exist positive constants c and n_0 such that
$0 \le cg(n) \le f(n)$ for all $n \ge n_0\}$.



$g(n)$ is an ***asymptotic lower bound*** for $f(n)$.

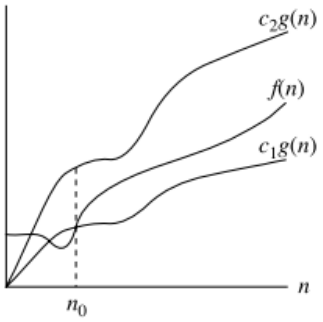
Example: $\sqrt{n} = \Omega(\lg n)$, with $c = 1$ and $n_0 = 16$.

Examples of functions in $\Omega(n^2)$:

n^2
$n^2 + n$
$n^2 - n$
$1000n^2 + 1000n$
$1000n^2 - 1000n$
Also,
n^3
$n^{2.00001}$
$n^2 \lg\lg\lg n$
2^{2^n}

Θ-notation

$\Theta(g(n)) = \{f(n) :$ there exist positive constants c_1, c_2, and n_0 such that
$0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0\}$.



$g(n)$ is an ***asymptotically tight bound*** for $f(n)$.

Example: $n^2/2 - 2n = \Theta(n^2)$, with $c_1 = 1/4$, $c_2 = 1/2$, and $n_0 = 8$.

o-notation

$o(g(n)) = \{f(n) :$ for all constants $c > 0$, there exists a constant $n_0 > 0$ such that $0 \le f(n) < cg(n)$ for all $n \ge n_0\}$.

Another view, probably easier to use: $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

$n^{1.9999} = o(n^2)$
$n^2/\lg n = o(n^2)$
$n^2 \neq o(n^2)$ (just like $2 \not< 2$)
$n^2/1000 \neq o(n^2)$

ω-notation

$\omega(g(n)) = \{f(n) :$ for all constants $c > 0$, there exists a constant $n_0 > 0$ such that $0 \le cg(n) < f(n)$ for all $n \ge n_0\}$.

Another view, again, probably easier to use: $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$.

$n^{2.0001} = \omega(n^2)$
$n^2 \lg n = \omega(n^2)$
$n^2 \neq \omega(n^2)$

Recurrences, Solution of Recurrences by substitution, Recursion Tree and Master Method

Recursion is a particularly powerful kind of reduction, which can be described loosely as follows:

• If the given instance of the problem is small or simple enough, just solve it.

• Otherwise, reduce the problem to one or more simpler instances of the same problem.

Recursion is generally expressed in terms of recurrences. In other words, when an algorithm calls to itself, we can often describe its running time by a **recurrence equation** which describes the overall running time of a problem of size n in terms of the running time on smaller inputs.

E.g.the worst case running time T(n) of the merge sort procedure by recurrence can be expressed as

$T(n) = \Theta(1)$; if n=1

$2T(n/2) + \Theta(n)$;if n>1

whose solution can be found as $T(n)=\Theta(n\log n)$

There are various techniques to solve recurrences.

1. **SUBSTITUTION METHOD**:

   The substitution method comprises of 3 steps

   i. Guess the form of the solution
   ii. Verify by induction
   iii. Solve for constants

We substitute the guessed solution for the function when applying the inductive hypothesis to smaller values. Hence the name "substitution method". This method is powerful, but we must be able to guess the form of the answer in order to apply it. e.g.recurrence equation: T(n)=4T(n/2)+n

step 1: guess the form of

solution T(n)=4T(n/2)

- ⇨ F(n)=4f(n/2)
- ⇨ F(2n)=4f(n)
- ⇨ $F(n)=n^2$

So, T(n) is order of n^2

Guess $T(n)=O(n^3)$

Step 2: verify the induction

Assume $T(k)<=ck^3$

T(n)=4T(n/2)+n

$<=4c(n/2)^3 +n$

$<=cn^3/2+n$

$<=cn^3-(cn^3/2-n)$

$T(n)<=cn^3$ as $(cn^3/2 –n)$ is always positive

So what we assumed was true.

⇨ $T(n)=O(n^3)$

Step 3: solve for

constants $Cn^3/2-$

$n>=0$

⇨ $n>=1$

⇨ $c>=2$

Now suppose we guess that $T(n)=O(n^2)$ which is tight upper

bound Assume,$T(k)<=ck^2$

so,we should prove that $T(n)<=cn^2$

$T(n)=4T(n/2)+n$

⇨ $4c(n/2)^2+n$

⇨ cn^2+n

So,T(n) will never be less than cn^2. But if we will take the assumption of $T(k)=c_1 k^2-c_2k$, then we can find that $T(n) = O(n^2)$

2. BY ITERATIVE METHOD:

e.g. $T(n)=2T(n/2)+n$

$=> 2[2T(n/4) + n/2]+n$

$=>2^2T(n/4)+n+n$

$=> 2^2[2T(n/8)+ n/4]+2n$

$=>2^3T(n/2^{3)} +3n$

After k iterations ,$T(n)=2^kT(n/2^k)+kn$ ---------- (1)

Sub problem size is 1 after $n/2^k=1$ => k=logn

So,afterlogn iterations ,the sub-problem size will be

1. So,when k=logn is put in equation 1

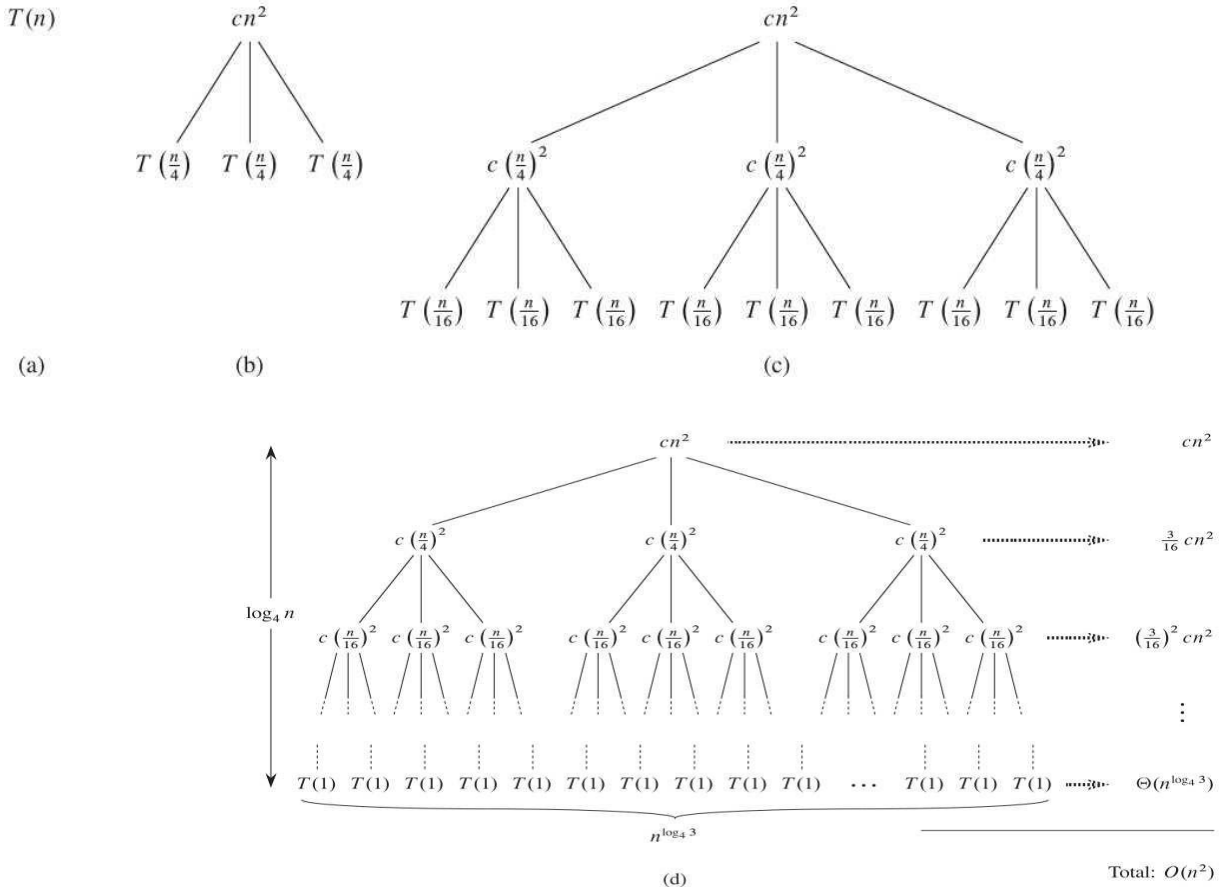
T(n)=nT(1)+nlogn

- ⇨ nc+nlogn (say c=T(1))
- ⇨ O(nlogn)

3.BY RECURSSION TREE METHOD:

In a recursion tree ,each node represents the cost of a single sub-problem somewhere in the set of recursive problems invocations .we sum the cost within each level of the tree to obtain a set of per level cost,and then we sum all the per level cost to determine the total cost of all levels of recursion .

Constructing a recursion tree for the recurrence $T(n)=3T(n/4)+cn^2$



Constructing a recursion tree for the recurrence T (n)= 3T (n=4) + cn^2.. Part (a) shows T (n), which progressively expands in (b)–(d) to form the recursion tree. The fully expanded tree in part (d) has height $\log_4 n$ (it has $\log_4 n + 1$ levels).

Sub problem size at depth i =$n/4^i$

Sub problem size is 1 when $n/4^i=1$ => $i=\log_4 n$

So, no. of levels =$1+ \log_4 n$

Cost of each level = (no. of nodes) x (cost of each node)

No. Of nodes at depth i=3^i

Cost of each node at depth i=$c\ (n/4^i)^2$

Cost of each level at depth i=$3^i\, c\ (n/4^i)^2 = (3/16)^i cn^2$

$T(n)= \sum_{i=0}^{\log_4 n} cn^2(3/16)^i$

$T(n)= \sum_{i=0}^{\log_4 n -1} cn^2(3/16)^i$ + cost of last level

Cost of nodes in last level =$3^i T(1)$

- ⇨ $c3^{\log_4 n}$ (at last level $i=\log_4 n$)
- ⇨ $cn^{\log 3}$

$$T(n)=\sum_{i=0}^{\log_4 n -1} cn^2(3/16)^i + c\, n^{\log \frac{3}{4}}$$

$$<= cn^2 \sum_{i=0}^{\infty} (3/16)^i + cn^{\log_4 3}$$

- ⇨ $<= cn^2*(16/13)+ cn^{\log \frac{3}{4}}$ => $T(n)=O(n^2)$

4.<u>BY MASTER METHOD:</u>

The master method solves recurrences of the form

$$T(n)=aT(n/b)+f(n)$$

where a>=1 and b>1 are constants and f(n) is a asymptotically positive

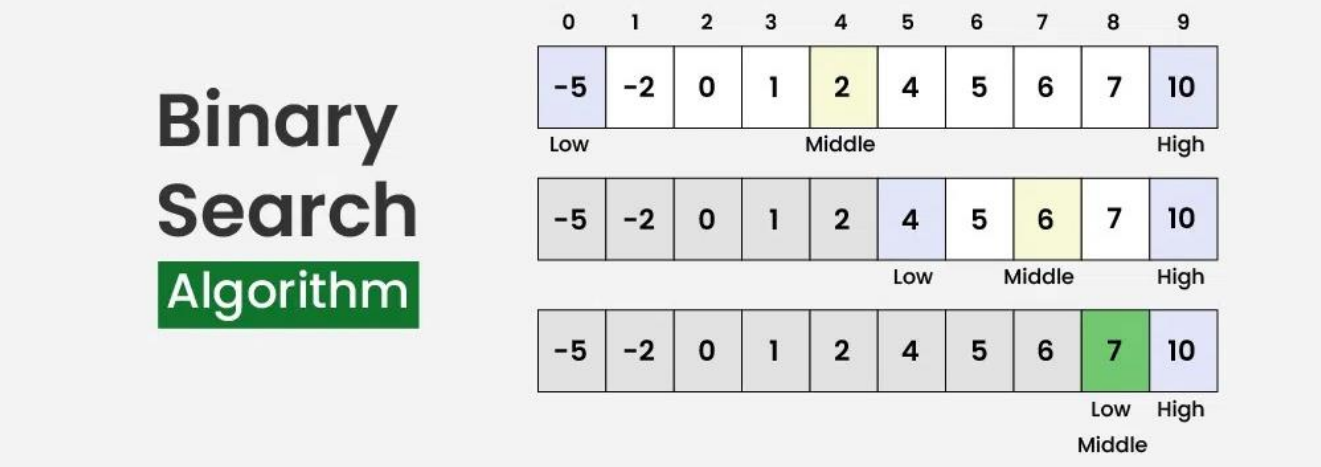
function . To use the master method, we have to remember 3 cases:

1. If $f(n)=O(n^{\log_b a - \varepsilon})$ for some constants Ɛ >0,then $T(n)=\Theta(n^{\log_b a})$
2. If $f(n)=\Theta(n^{\log_b a})$ then $T(n)=\Theta(n^{\log_b a}\log n)$

If $f(n)=Ω(n^{\log_b a+\varepsilon})$ for some constant Ɛ>0 ,and if a*f(n/b)<=c*f(n) for some constant c<1 and all sufficiently large n,then $T(n)=\Theta(f(n)$

Binary Search

Binary Search is a searching algorithm that operates on a sorted or monotonic search space, repeatedly dividing it into halves to find a target value or optimal answer in logarithmic time O(log N).



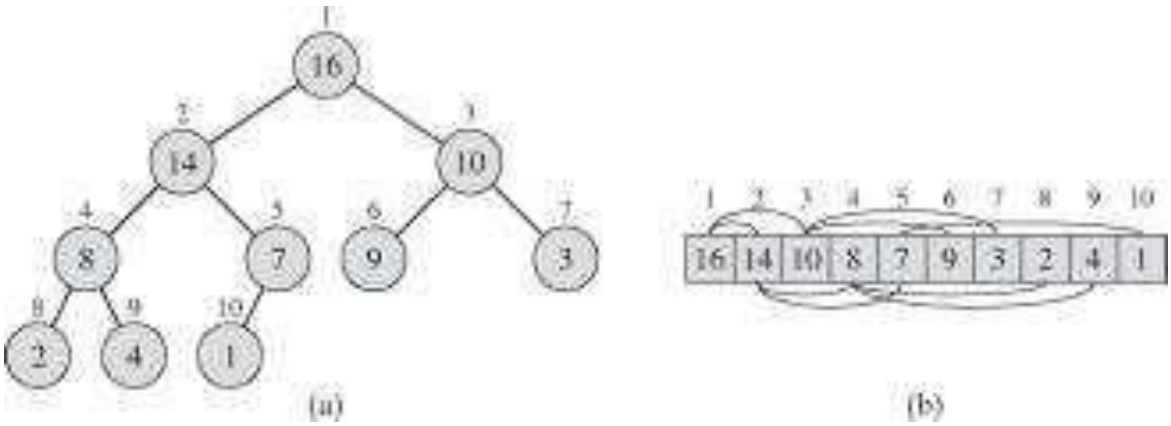
Heaps and Heap sort

HEAPSORT

- Inplace algorithm
- Running Time: O(n log n)
- Complete Binary Tree

The ***(binary) heap*** data structure is an array object that we can view as a nearly complete binary tree.Each node of the tree corresponds to an element of the array. The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point.

The root of the tree is A[1], and given the index *i* of a node, we can easily compute the indices of its parent, left child, and right child:



- PARENT (i) => ***return*** [i / 2]
- LEFT (i) => ***return*** 2*i*
- RIGHT (i) => ***return*** 2i+ 1

On most computers, the LEFT procedure can compute 2i in one instruction by simply shifting the binary representation of i left by one bit position.

Similarly, the RIGHT procedure can quickly compute 2i + 1 by shifting the binary representation of i left by one bit position and then adding in a 1 as the low-order bit.

The PARENT procedure can compute *[i/2]* by shifting i right one bit position. Good implementations of heapsort often implement these procedures as "macros" or "inline" procedures.

There are two kinds of binary heaps: **max-heaps** and **min-heaps**.

- In a ***max-heap,***the ***max-heap property*** is that for every node *i* other than the root, **A[PARENT(i)] >= A[i] ,**that is, the value of a node is at most the value of its parent. Thus, the largest element in a max-heap is stored at the root, and the subtree rooted at a

node contains values no larger than that contained at the node itself.

- A ***min-heap*** is organized in the opposite way; the ***min-heap property*** is that for every node i other than the root, **A[PARENT(i)<=A[i] ,**

  The smallest element in a min-heap is at the root.

- ✓ The height of a node in a heap is the number of edges on the longest simple downward path from the node to a leaf and
- ✓ The height of the heap is the height of its root.
- ✓ Height of a heap of n elements which is based on a complete binary tree is **O(log n).**

Maintaining the heap property

MAX-HEAPIFY lets the value at A[i] "float down" in the max-heap so that the subtree rooted at index i obeys the max-heap property.

MAX-HEAPIFY(A,i)

1. l ← LEFT(i)
2. r ←RIGHT(i)
3. if A[l] > A[i]
4. largest ←l
5. if A[r] > A[largest]
6. Largest ←r
7. if largest != i
8. Then exchange A[i] ↔A[largest]
9. MAX-HEAPIFY(A,largest)

At each step, the largest of the elements A[i], A[LEFT(i)], and A[RIGHT(i)] is determined, and its index is stored in *largest.* If A[i] is largest, then the subtree rooted at node i is already a max-heap and the procedure terminates. Otherwise, one of the two children has the largest element, and A[i] is swapped with A[largest], which causes node i and its children to satisfy the max-heap property. The node indexed by *largest,* however, now has the original value A[i], and thus the subtree rooted at *largest* might violate the max-heap property. Consequently, we call MAX-HEAPIFY recursively on that subtree.

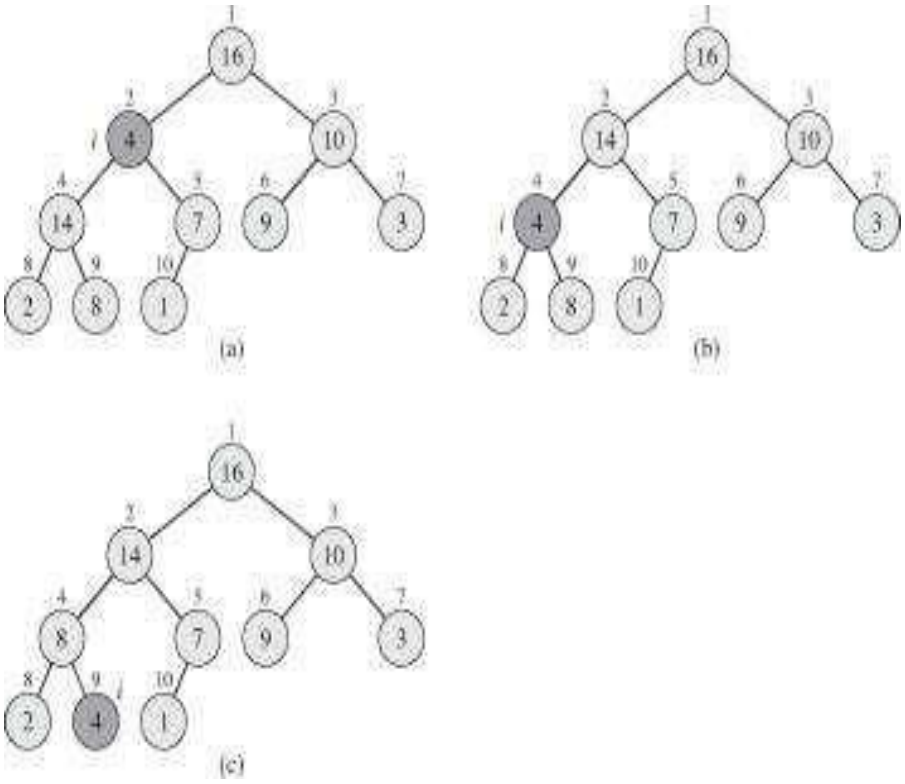


Figure: The action of MAX-HEAPIFY (A, 2), where *heap-size* = 10. **(a)** The initial con- figuration, with A [2] at node i = 2 violating the max-heap property since it is not larger than both children. The max-heap property is restored for node 2 in **(b)** by exchanging A [2] with A[4], which destroys the max-heap property for node 4. The recursive call MAX-HEAPIFY (A,4)

now has i = 4. After swapping A[4] with A[9], as shown in **(c),** node 4 is fixed up, and the recursive call MAX-HEAPIFY(A, 9) yields no further change to the data structure.

The running time of MAX-HEAPIFY by the recurrence can be

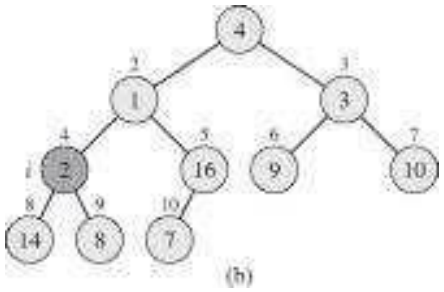
described as T (n) < = T (2n/3) + O (1)

The solution to this recurrence is T(n)=O(log n)

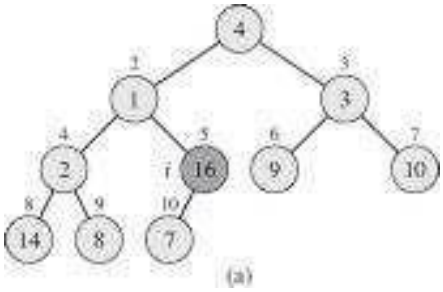
Building a heap

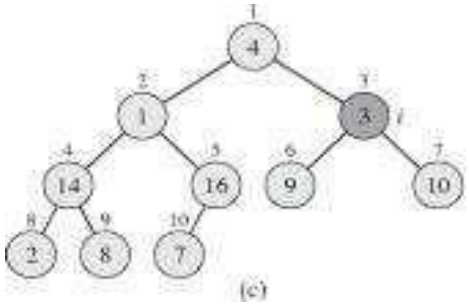
Build-Max-Heap(A)

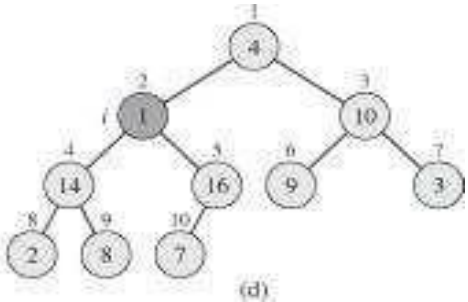
1. for i⌧[n/2] to 1
2. do MAX-HEAPIFY(A,i)

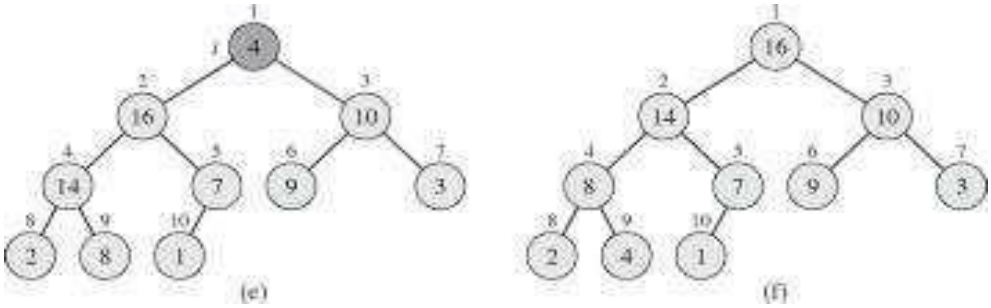


4	1	3	2	1	9	1	1	8	7









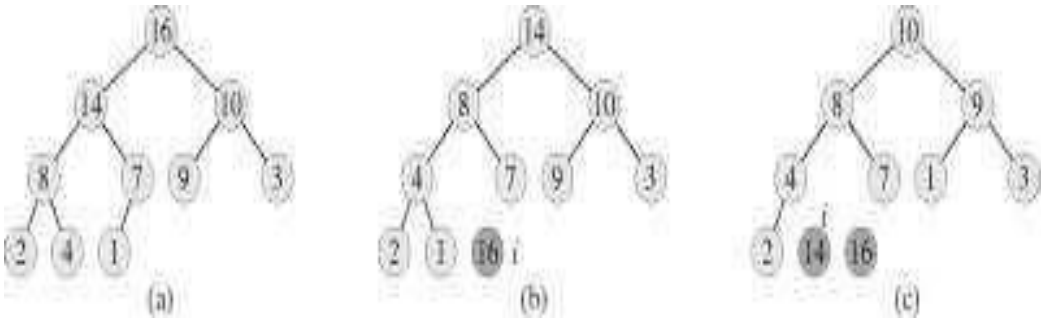
We can derive a tighter bound by observing that the time for MAX-HEAPIFY to run at a node varies with the height of the node in the tree, and the heights of most nodes are small. Our tighter analysis relies on the properties that an n-element heap has height [log n] and at most $[n/2^{h+1}]$ nodes of any height h.

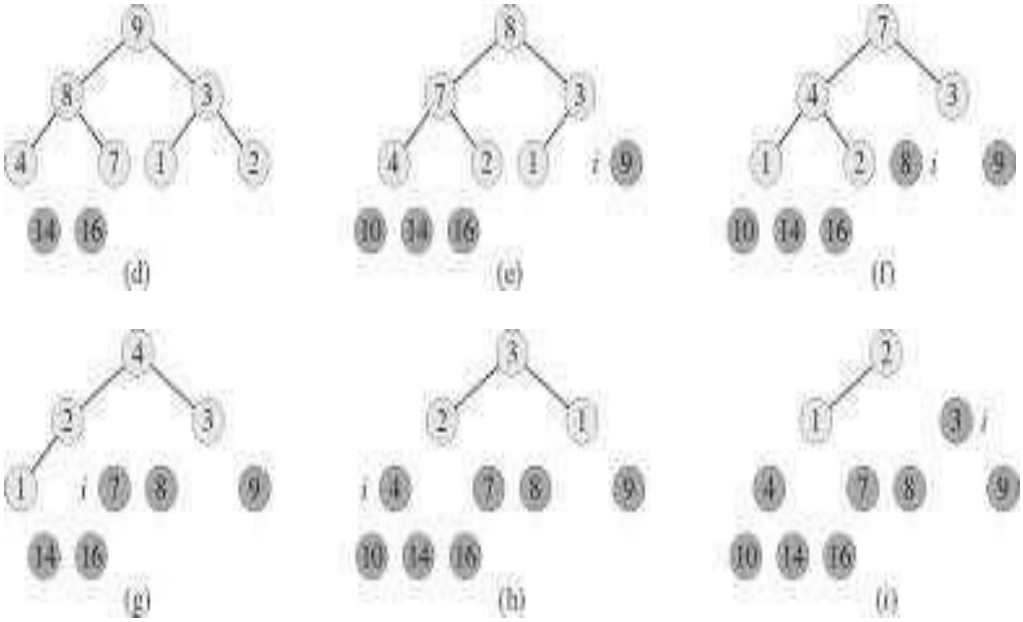
The total cost of BUILD-MAX-HEAP as being bounded is T(n)=O(n)

The HEAPSORT Algorithm

HEAPSORT(A)

1. BUILD MAX-HEAP(A)
2. for i=n to 2
3. exchange A[1] with A[i]
4. MAX-HEAPIFY(A,1)





A

1	2	3	4	7	8	9	10	14	16

TheHEAPSORT procedure takes time O(n log n), since the call to BUILD-MAX- HEAP takes time O(n) and each of the n - 1 calls to MAX-HEAPIFY takes time O(log n).

e.g. T(n)=2T(n/2)+nlogn

Lower Bounds For Sorting

Review of Sorting: So far we have seen a number of algorithms for sorting a list of numbers in ascendingorder. Recall that an *in-place* sorting algorithm is one that uses no additional array storage (however,we allow Quicksort to be called in-place even though they need a stack of size $O(\log n)$ for keepingtrack of the recursion). A sorting algorithm is *stable* if duplicate elements remain in the same relativeposition after sorting.

Slow Algorithms: Include BubbleSort, InsertionSort, and SelectionSort. These are all simple $\Theta(n^2)$in-place sorting algorithms. BubbleSort and InsertionSort can be implemented as stable algorithms,but SelectionSort cannot (without significant modifications).

Mergesort: Mergesort is a stable Θ(*n* log *n*) sorting algorithm. The downside is that MergeSort isthe only algorithm of the three that requires additional array storage, implying that it is not anin-place algorithm.

Quicksort: Widely regarded as the *fastest* of the fast algorithms. This algorithm is *O*(*n* log *n*) in the*expected case*, and *O*(*n*2) in the worst case. The probability that the algorithm takes asymptoticallylonger (assuming that the pivot is chosen randomly) is extremely small for large *n*. It is an(almost) in-place sorting algorithm but is not stable.

Heapsort: Heapsort is based on a nice data structure, called a *heap*, which is a fast priority queue.Elements can be inserted into a heap in *O*(log *n*) time, and the largest item can be extracted in*O*(log *n*) time. (It is also easy to set up a heap for extracting the smallest item.) If you only wantto extract the *k* largest values, a heap can allow you to do this is *O*(*n* + *k* log *n*) time. It is anin-place algorithm, but it is not stable.

Lower Bounds for Comparison-Based Sorting: Can we sort faster than *O*(*n* log *n*) time?

We will give anargument that if the sorting algorithm is based solely on making comparisons between the keys in thearray, then it is impossible to sort more efficiently than (*n* log *n*) time. Such an algorithm is called a*comparison-based* sorting algorithm, and includes all of the algorithms given above.Virtually all known general purpose sorting algorithms are based on making comparisons, so this isnot a very restrictive assumption. This does not preclude the possibility of a sorting algorithm whoseactions are determined by other types of operations, for example, consulting the individual bits ofnumbers, performing arithmetic operations, indexing into an array based on arithmetic operations onkeys.We will show that any *comparison-based* sorting algorithm for a input sequence *ha*1*; a*2*; : : : ; ani*must

make at least (*n* log *n*) comparisons in the worst-case. This is still a difficult task if you think about it.It is easy to show that a problem *can* be solved fast (just give an algorithm). But to show that a problem*cannot* be solved fast you need to reason in some way about all the possible algorithms that might everbe written. In fact, it seems surprising that you could even hope to prove such a thing. The catch hereis that we are limited to using comparison-based algorithms, and there is a clean mathematical way ofcharacterizing all such algorithms.

Decision Tree Argument: In order to prove lower bounds, we need an abstract way of modeling "any possible"comparison-based sorting algorithm, we model such algorithms in terms of an abstract modelcalled a *decision tree*.In a *comparison-based* sorting algorithm only comparisons between the keys are used to determinethe action of the algorithm. Let *ha*1*; a*2*; : : : ; ani*be the input sequence. Given two elements, *ai*and*aj*, their relative order can only be determined by the results of comparisons like*ai<aj*, *ai<=aj*,*ai=aj*, *ai>=aj*, and *ai>aj*.A decision tree is a mathematical representation of a sorting algorithm (for a fixed value of *n*). Eachnode of the decision tree represents a comparison made in the algorithm (e.g., *a*4 : *a*7), and the twobranches represent the possible results, for example, the left subtree consists of the remaining comparisonsmade under the assumption that *a*4 _ *a*7 and the right subtree for *a*4 *> a*7. (Alternatively, onemight be labeled with *a*4 *< a*7 and the other with *a*4 _ *a*7.)Observe that once we know the value of *n*, then the "action" of the sorting algorithm is completelydetermined by the results of its comparisons. This action may involve moving elements around in thearray, copying them to other locations in memory, performing various arithmetic operations on non- keydata. But the bottom-line is that at the end of the algorithm, the keys will be permuted in the final array in some way. Each leaf in the decision tree is labeled with the final permutation that the algorithmgenerates after making all of its comparisons.To make this more concrete, let us assume that *n* = 3, and let's build a decision tree for SelectionSort.Recall that the algorithm consists of two phases. The first finds the smallest element of the entire list,and swaps it with the first element. The second finds the smaller of the remaining two items, and swapsit with the second element. Here is the decision tree (in outline form). The first comparison is between*a*1 and *a*2. The possible results are:

*a*1 *<= a*2**:** Then *a*1 is the current minimum. Next we compare *a*1 with *a*3 whose results might be either:

*a*1 *<=a*3**:** Then we know that *a*1 is the minimum overall, and the elements remain in their originalpositions. Then we pass to phase 2 and compare *a*2 with *a*3. The possible results are:

*a*2 *<=a*3**:** Final output is *ha*1*; a*2*; a*3*i*.

*a*2 *> a*3**:** These two are swapped and the final output is *ha*1*; a*3*; a*2*i*.

*a*1 > *a*3**:** Then we know that *a*3 is the minimum is the overall minimum, and it is swapped with*a*1. The we pass to phase 2 and compare *a*2 with *a*1 (which is now in the third position ofthe array) yielding either:

*a*2 <=*a*1**:** Final output is *ha*3*; a*2*; a*1*i*.

*a*2 > *a*1**:** These two are swapped and the final output is *ha*3*; a*1*; a*2*i*.

*a*1 > *a*2**:** Then *a*2 is the current minimum. Next we compare *a*2 with *a*3 whose results might be either:

*a*2 <=*a*3**:** Then we know that *a*2 is the minimum overall. We swap *a*2 with *a*1, and then pass to phase 2, and compare the remaining items *a*1 and *a*3. The possible results are:

*a*1 <=*a*3**:** Final output is *ha*2*; a*1*; a*3*i*.

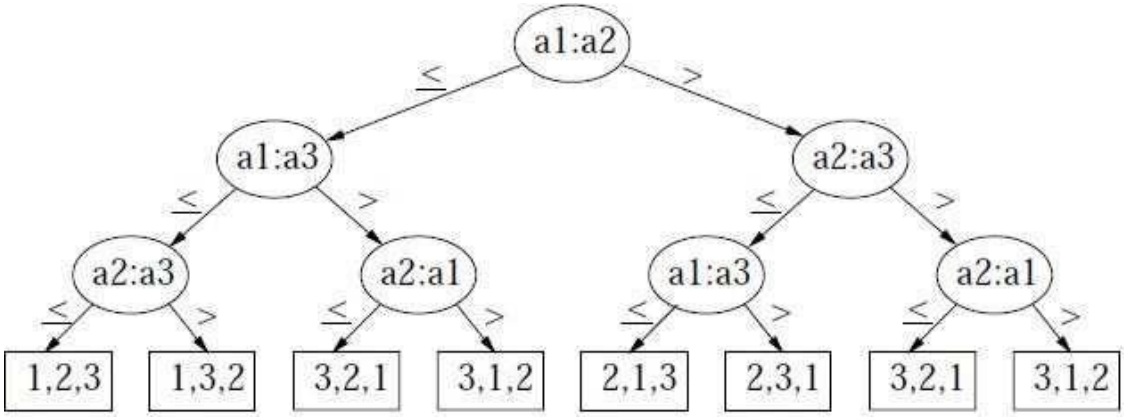
*a*1 > *a*3**:** These two are swapped and the final output is *ha*2*; a*3*; a*1*i*.

*a*2 > *a*3**:** Then we know that *a*3 is the minimum is the overall minimum, and it is swapped with*a*1. We pass to phase 2 and compare *a*2 with *a*1 (which is now in the third position of thearray) yielding either:

*a*2<= *a*1**:** Final output is *ha*3*; a*2*; a*1*i*.

*a*2 > *a*1**:** These two are swapped and the final output is *ha*3*; a*1*; a*2*i*.

The final decision tree is shown below. Note that there are some nodes that are unreachable. For example,in order to reach the fourth leaf from the left it must be that *a*1 _ *a*2 and *a*1 > *a*2, which cannotboth be true. Can you explain this? (The answer is that virtually all sorting algorithms, especiallyinefficient ones like selection sort, may make comparisons that are redundant, in the sense that theiroutcome has already been determined by earlier comparisons.) As you can see, converting a complexsorting algorithm like HeapSort into a decision tree for a large value of *n* will be very tedious andcomplex, but I hope you are convinced by this exercise that it can be done in a simple mechanical way.



(Decision Tree for SelectionSort on 3 keys.)

Using Decision Trees for Analyzing Sorting: Consider any sorting algorithm. Let $T(n)$ be the maximumnumber of comparisons that this algorithm makes on any input of size n. Notice that the running timefo the algorithm must be at least as large as $T(n)$, since we are not counting data movement or othercomputations at all. The algorithm defines a decision tree. Observe that the height of the decisiontree is exactly equal to $T(n)$, because any path from the root to a leaf corresponds to a sequence ofcomparisons made by the algorithm.

As we have seen earlier, any binary tree of height $T(n)$ has at most $2T(n)$ leaves. This means that thissorting algorithm can *distinguish* between at most $2T(n)$ different final actions. Let's call this quantity$A(n)$, for the number of different final actions the algorithm can take. Each action can be thought of asa specific way of permuting the original input to get the sorted output.How many possible actions must any sorting algorithm distinguish between? If the input consists of $n$distinct numbers, then those numbers could be presented in any of $n!$ different permutations. For eachdifferent permutation, the algorithm must "unscramble" the numbers in an essentially different way,that is it must take a different action, implying that $A(n) >= n!$. (Again, $A(n)$ is usually not exactlyequal to $n!$ because most algorithms contain some redundant unreachable leaves.)

Since $A(n) \leq 2T(n)$ we have $2T(n) \geq n!$, implying that

$T(n) \geq \lg(n!)$*:*

We can use *Stirling's approximation* for *n*! yielding:

$n! \geq \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$

$T(n) \geq \log\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\right)$

$= \log\sqrt{2\pi n} + n\log n - n\log e \in \Omega(n\log n)$

Thus we have the following theorem.

Theorem: Any comparison-based sorting algorithm has worst-case running time (*n* log *n*).

This can be generalized to show that the *average-case* time to sort is also (*n* log *n*) (by arguing aboutthe average height of a leaf in a tree with at least *n*! leaves). The lower bound on sorting can begeneralized to provide lower bounds to a number of other problems as well.

ans: a=2 b=2

f(n)=nlogn

using 2nd formula

$f(n)=\Theta(n^{\log_2 2}\log^k n)$

$\Rightarrow\Theta(n^1\log^k n)=n\log n$ $\Rightarrow K=1$

$T(n)=\Theta(n^{\log_2 2}\log^1 n)$

$\Rightarrow\Theta(n\log^2 n)$

Worst case analysis of merge sort, quick sort

Merge sort

It is one of the well-known divide-and-conquer algorithm. This is a simple and very efficient algorithm for sorting a list of numbers.

We are given a sequence of n numberswhich we will assume is stored in an array A [1...n]. Theobjective is to output a permutation of this sequence, sorted in increasing order. This is normally done by permuting the elements within the array A.

How can we apply divide-and-conquer to sorting? Here are the major elements of the Merge Sort algorithm.

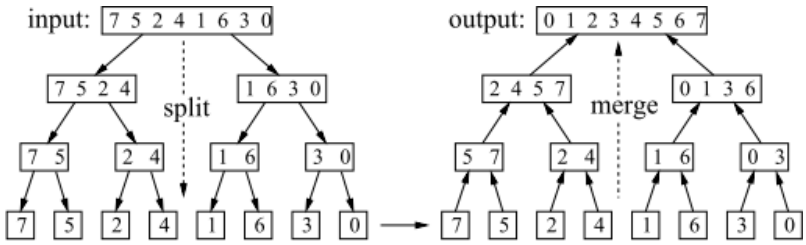
Divide: Split A down the middle into two sub-sequences, each of size roughly n/2

. Conquer: Sort each subsequence (by calling MergeSort recursively on each).

Combine: Merge the two sorted sub-sequences into a single sorted list.

The dividing process ends when we have split the sub-sequences down to a single item. A sequence of length one is trivially sorted. The key operation where all the work is done is in the combine stage,which merges together two sorted lists into a single sorted list. It turns out that the merging process is quite easy to implement.

The following figure gives a high-level view of the algorithm. The “divide” phase is shown on the left. It works top-down splitting up the list into smaller sublists. The “conquer and combine” phases areshown on the right. They work bottom-up, merging sorted lists together into larger sorted lists.



Merge Sort

Designing the Merge Sort algorithm top-down. We’ll assume that the procedure thatmerges two sortedlist is available to us. We’ll implement it later. Because the algorithm is called recursively on sublists,in addition to passing in the array itself, we will pass in two indices, which indicate the first and lastindices of the subarray that we are to sort. The call MergeSort(A, p, r) will sort the sub-arrayA [p..r] and return the sorted result in the same subarray.

Here is the overview. If r = p, then this means that there is only one element to sort, and we may returnimmediately. Otherwise (if p < r) there are at least two elements, and we will invoke the divide-and-conquer. We find the index q, midway between p and r, namely q = (p + r) / 2 (rounded down to thenearest integer). Then we split the array into subarrays A [p..q] and A [q

+ 1 ..r] . Call Merge Sort recursively to sort each subarray. Finally, we invoke a procedure (which we have yet to write) whichmerges these two subarrays into a single sorted array.

```
MergeSort(array A, int p, int r) {
    if      (p < r) {                          // we have at least 2 items
    q = (p + r)/2

    MergeSort(A, p, q)                         // sort A[p..q]

    MergeSort(A, q+1, r)                       // sort A[q+1..r]
```

```
    Merge(A, p, q, r)                    // merge everything together
    }                                                              }
```

Merging: All that is left is to describe the procedure that merges two sorted lists. Merge(A, p, q, r)assumes that the left subarray, A [p..q] , and the right subarray, A [q + 1 ..r] , have already been sorted.We merge these two subarrays by copying the elements to a temporary working array called B. Forconvenience, we will assume that the array B has the same index range A, that is, B [p..r] . We have to indices i and j, that point to the current elements ofeach subarray. We move the smaller element into the next position of B (indicated by index k) andthen increment the corresponding index (either i or j). When we run out of elements in one array, thenwe just copy the rest of the other array into B. Finally, we copy the entire contents of B back into A.

```
Merge(array A, int p, int q, int r) {          // merges A[p..q] with
    A[q+1..r] array B[p..r]
    i = k = p                                  //initialize     pointers
    j = q+1
    while (i <= q and j <= r) {                // while both subarrays are nonempty
    if (A[i] <= A[j]) B[k++] = A[i++]          // copy from left subarray
    else B[k++] = A[j++]                       // copy from right subarray
    }
    while (i <= q) B[k++] = A[i++]             // copy any leftover to
    B while (j <= r) B[k++] = A[j++]
    for i = p to r do A[i] = B[i]          // copy B back to A                    }
```

Analysis: What remains is to analyze the running time of MergeSort. First let us consider the running timeof the procedure Merge(A, p, q, r). Let n = r − p + 1 denote the total length of both the leftand right subarrays. What is the running time of Merge as a function of n? The algorithm contains fourloops (none nested in the other). It is easy to see that each loop can

be executed at most n times. (Ifyou are a bit more careful you can actually see that all the while-loops

together can only be executed ntimes in total, because each execution copies one new element to the array B, and B only has space forn elements.) Thus the running time to Merge n items is $\Theta(n)$. Let us write this without the asymptoticnotation, simply as n. (We'll see later why we do this.)

Now, how do we describe the running time of the entire MergeSort algorithm? We will do this throughthe use of a recurrence, that is, a function that is defined recursively in terms of itself. To avoidcircularity, the recurrence for a given value of n is defined in terms of values that are strictly smallerthan n. Finally, a recurrence has some basis values (e.g. for $n = 1$), which are defined explicitly.

Let's see how to apply this to MergeSort. Let $T(n)$ denote the worst case running time of MergeSort onan array of length n. For concreteness we could count whatever we like: number of lines of pseudocode,number of comparisons, number of array accesses, since these will only differ by a constant factor.Since all of the real work is done in the Merge procedure, we will count the total time spent in theMerge procedure.

First observe that if we call MergeSort with a list containing a single element, then the running time is aconstant. Since we are ignoring constant factors, we can just write $T(n) = 1$. When we call MergeSortwith a list of length $n > 1$, e.g. Merge(A, p, r), where $r - p + 1 = n$, the algorithm first computes $q = (p + r) / 2$. The subarray $A[p..q]$, which contains $q - p + 1$ elements. You can verify that is of size $n/2$. Thus the remaining subarray $A[q + 1..r]$ has $n/2$ elements in it. How long does it taketo sort the left subarray? We do not know this, but because $n/2 < n$ for $n > 1$, we can express this as $T(n/2)$. Similarly, we can express the time that it takes to sort the right subarray as $T(n/2)$.

Finally, to merge both sorted lists takes n time, by the comments made above. In conclusion we have

$T(n) = 1$ if $n = 1$,

$2T(n/2) + n$ otherwise.

Solving the above recurrence we can see that merge sort has a time complexity of $\Theta(n \log n)$.

QUICKSORT

- Worst-case running time: $O(n2)$.
- Expected running time: $O(n \lg n)$.
- Sorts in place.

Description of quicksort

Quicksort is based on the three-step process of divide-and-conquer.

• To sort the subarray$A[p \ldots r]$:

Divide: Partition $A[p \ldots r]$, into two (possibly empty) subarrays$A[p \ldots q-1]$ and $A[q+1 \ldots r]$, such that each element in the Þrstsubarray$A[p \ldots q-1]$ is $\leq A[q]$ and $A[q]$ is ≤ each element in the second subarray$A[q+1 \ldots r]$.

Conquer: Sort the two subarrays by recursive calls to QUICKSORT.

Combine: No work is needed to combine the subarrays, because they are sorted in place.

• Perform the divide step by a procedure PARTITION, which returns the index q that marks the position separating the subarrays.

QUICKSORT *(A, p, r)*

if $p < r$

then $q \leftarrow$PARTITION*(A, p, r)*

QUICKSORT *(A, p, q − 1)*

QUICKSORT *(A, q + 1, r)*

Initial call is QUICKSORT *(A,* 1*, n)*

Partitioning

Partition subarray$A[p \ldots r]$ by the following procedure:

PARTITION *(A, p, r)*

$x \leftarrow A[r]$

$i \leftarrow p - 1$

for $j \leftarrow p$ **to** $r - 1$

do if $A[j] \leq x$

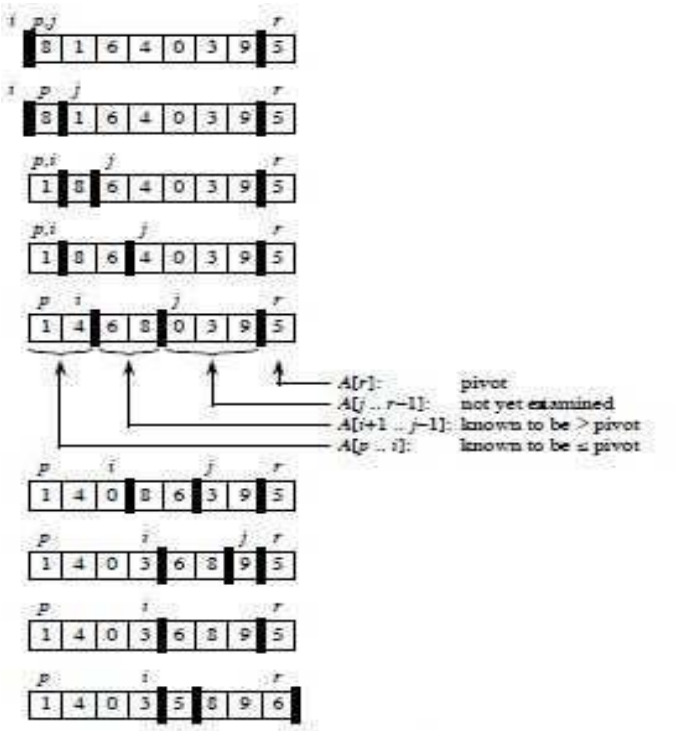
then $i \leftarrow i + 1$

exchange$A[i] \leftrightarrow A[j]$

exchange$A[i+1] \leftrightarrow A[r]$

return $i + 1$

- PARTITION always selects the last element $A[r]$ in the subarray $A[p \ldots r]$ as the ***pivot*** the element around which to partition.
- As the procedure executes, the array is partitioned into four regions, some of which may be empty:



[The index j disappears because it is no longer needed once the for loop is exited.]

Performance of Quicksort

The running time of Quicksort depends on the partitioning of the subarrays:

• If the subarrays are balanced, then Quicksort can run as fast as mergesort.

• If they are unbalanced, then Quicksort can run as slowly as insertion sort.

Worst case

• Occurs when the subarrays are completely unbalanced.

• Have 0 elements in one subarray and $n - 1$ elements in the other subarray.

• Get the recurrence

$T(n) = T(n-1) + T(0) + \Theta(n)$

$= T(n-1) + \Theta(n)$

$= O(n2)$.

• Same running time as insertion sort.

• In fact, the worst-case running time occurs when Quicksort takes a sorted array as input, but insertion sort runs in *O(n)* time in this case.

Best case

• Occurs when the subarrays are completely balanced every time.

• Each subarray has ≤ $n/2$ elements.

• Get the recurrence

$T(n) = 2T(n/2) + \Theta(n) = O(n \lg n).$

Balanced partitioning

• QuickPort's average running time is much closer to the best case than to the worst case.

• Imagine that PARTITION always produces a 9-to-1 split.

• Get the recurrence

$T(n) \leq T(9n/10) + T(n/10) + \_ (n) = O(n \lg n).$

• Intuition: look at the recursion tree.

• It's like the one for $T(n) = T(n/3) + T(2n/3) + O(n).$

• Except that here the constants are different; we get log10 *n* full levels and log10⁄9 *n* levels that are nonempty.

• As long as it's a constant, the base of the log doesn't matter in asymptotic notation.

• Any split of constant proportionality will yield a recursion tree of depth $O(\lg n)$.

MODULE-2

Dynamic Programming

Matrix-chain multiplication:

Matrix Chain Multiplication is an algorithm that is applied to determine the lowest cost way for multiplying matrices. The actual multiplication is done using the standard way of multiplying the

matrices, i.e., it follows the basic rule that the number of rows in one matrix must be equal to the number of columns in another matrix.

MatrixChainOrder(p[], n)

1. Create a table m[1..n][1..n] to store minimum cost

2. For i = 1 to n:

m[i][i] = 0 // single matrix cost = 0

3. For L = 2 to n-1: // L = chain length

For i = 1 to n - L + 1:

j = i + L - 1

m[i][j] = ∞

For k = i to j - 1:

q = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j]

If (q < m[i][j])

m[i][j] = q

Return m[1][n-1]

Formula (Recursive Relation)

m[i][j]=0if i=jm[i][j] = 0 \quad \text{if } i = jm[i][j]=0if i=j
m[i][j]=min⁡i≤k<j{m[i][k]+m[k+1][j]+p[i−1]×p[k]×p[j]}m[i][j] = \min_{i \leq k < j} \{ m[i][k] + m[k+1][j] + p[i-1] \times p[k] \times p[j] \}m[i][j]=i≤k<jmin{m[i][k]+m[k+1][j]+p[i−1]×p[k]×p[j]}

Here:

- m[i][j] → minimum number of multiplications needed to compute $A_i \ldots A_j$
- p[] → array of matrix dimensions

Example:

Matrix	**Dimensions**
A_1	10 × 30
A_2	30 × 5
A_3	5 × 60

So,
p[] = {10, 30, 5, 60}
Number of matrices n = 3

Step 1: Initialization

m[1][1] = 0
m[2][2] = 0
m[3][3] = 0

Step 2: Chain length = 2

- (A_1A_2): cost = 10×30×5 = 1500
- (A_2A_3): cost = 30×5×60 = 9000

m[1][2] = 1500
m[2][3] = 9000

Step 3: Chain length = 3

Only one way: ($A_1A_2A_3$)

We can split between:

1. k = 1 → (A_1)(A_2A_3)
   - cost = 0 + 9000 + 10×30×60 = **27000**
2. k = 2 → (A_1A_2)(A_3)
   - cost = 1500 + 0 + 10×5×60 = **4500**

Minimum = 4500

Result

- Minimum number of multiplications = **4500**
- Optimal order = **(A_1A_2)A_3**

Elements of dynamic programming:

Dynamic Programming (DP) is a method used to solve **complex problems** by breaking them into **smaller overlapping subproblems** and solving each subproblem **only once**.
It stores the results to avoid repeated computation.

No.	Element	Description
1	**Optimal Substructure**	A problem has an optimal substructure if an **optimal solution** of the problem can be formed from **optimal solutions of its subproblems**. *Example:* In **Shortest Path** problems (like Dijkstra's), the shortest path from A to C via B is the sum of shortest paths A→B and B→C.
2	**Overlapping Subproblems**	The problem can be broken into **smaller subproblems** which are **reused multiple times**. *Example:* In **Fibonacci series**, F(5) requires F(4) and F(3), but F(4) again requires F(3) — subproblems repeat.
3	**Memoization (Top-Down Approach)**	Store the results of solved subproblems in a **table (or array)** so that they can be reused instead of recalculating. Uses recursion + storage.

No.	Element	Description
4	**Tabulation (Bottom-Up Approach)**	Build the solution **iteratively** from the smallest subproblems up to the final answer.□ Uses loops + tables (no recursion).
5	**State Definition**	Clearly define what each subproblem (state) represents in terms of the original problem.□ Example: In Matrix Chain Multiplication, m[i][j] represents the minimum cost to multiply matrices from i to j.
6	**Recurrence Relation**	Define a mathematical formula that relates a subproblem to smaller subproblems.□ Example: F(n) = F(n-1) + F(n-2) in Fibonacci.
7	**Reconstruction of Solution**	After computing the minimum/maximum value, sometimes we need to **trace back** to find the **actual solution or path** used to reach that result.□ Example: Tracing the order of matrix multiplication in MCM.
No.	**Element**	Description
1	**Optimal Substructure**	A problem has an optimal substructure if an optimal solution of the problem can be formed from optimal solutions of its subproblems.□ Example: In Shortest Path problems (like Dijkstra's), the shortest path from A to C via B is the sum of shortest paths A→B and B→C.
2	**Overlapping Subproblems**	The problem can be broken into smaller subproblems which are reused multiple times.□ Example: In Fibonacci series, F(5) requires F(4) and F(3), but F(4) again requires F(3) — subproblems repeat.
3	**Memoization (Top-Down Approach)**	Store the results of solved subproblems in a table (or array) so that they can be reused instead of recalculating.□ Uses recursion + storage.
4	**Tabulation (Bottom-Up Approach)**	Build the solution iteratively from the smallest subproblems up to the final answer.□ Uses loops + tables (no recursion).
5	**State Definition**	Clearly define what each subproblem (state) represents in terms of the original problem.□ Example: In Matrix Chain Multiplication, m[i][j] represents the minimum cost to multiply matrices from i to j.
6	**Recurrence Relation**	Define a mathematical formula that relates a subproblem to smaller subproblems.□ Example: F(n) = F(n-1) + F(n-2) in Fibonacci.
7	**Reconstruction of Solution**	After computing the minimum/maximum value, sometimes we need to trace back to find the actual solution or path used to reach that result.□ Example: Tracing the order of matrix multiplication in MCM.

Examples

- Fibonacci Sequence
- Knapsack Problem
- Matrix Chain Multiplication
- Shortest Path (Floyd–Warshall)
- Longest Common Subsequence

Longest Common Subsequence (LCS)

The **Longest Common Subsequence (LCS)** problem is a **Dynamic Programming** problem that finds the **longest sequence of characters** that appears **in the same order** in both given strings (not necessarily contiguous).

We define LCS(X, Y) as the length of the longest subsequence common to both strings.

If:

- X[m-1] == Y[n-1], then
  LCS(X, Y) = 1 + LCS(X-1, Y-1)
- Else
  LCS(X, Y) = max(LCS(X-1, Y), LCS(X, Y-1))

Recursive Formula

$$L[i][j] = \begin{cases} 0, & \text{if } i = 0 \text{ or } j = 0 \\ L[i-1][j-1] + 1, & \text{if } X[i-1] = Y[j-1] \\ \max(L[i-1][j], L[i][j-1]), & \text{if } X[i-1] \neq Y[j-1] \end{cases}$$

Algorithm

LCS(X, Y, m, n)

1. Create a table L[0..m][0..n]

2. For i = 0 to m:

L[i][0] = 0

For j = 0 to n:

L[0][j] = 0

3. For i = 1 to m:

For j = 1 to n:

If X[i-1] == Y[j-1]:

L[i][j] = L[i-1][j-1] + 1

Else:

L[i][j] = max(L[i-1][j], L[i][j-1])

Return L[m][n]

Example

Let's take:
X = **"ABCBDAB"**, Y = **"BDCABA"**

		B	**D**	**C**	**A**	**B**	**A**
	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
B	0	1	1	2	2	3	3
D	0	1	2	2	2	3	3
A	0	1	2	2	3	3	4
B	0	1	2	2	3	4	4

Length of LCS = 4
LCS = "BCBA" or "BDAB"

Time Complexity
O(m × n)

Greedy Algorithms

The **Activity Selection Problem** is a **Greedy Algorithm** problem that aims to **select the maximum number of activities** that can be performed by a single person or machine, assuming **only one activity can be performed at a time**.

Each activity has:

- A **start time**
- A **finish time**

The goal is to choose the **maximum set of non-overlapping activities**.

Algorithm

ActivitySelection(s[], f[], n)

1. Sort activities by their finish time f[i]

2. Print the first activity (index = 0)

3. Let j = 0 // index of last selected activity

4. For i = 1 to n-1:

if s[i] >= f[j]:

select activity i

j = i

Elements of Greedy Strategy

A **Greedy Strategy** is an algorithmic approach where a problem is solved by making a **series of choices**, each of which looks **best at the moment** (locally optimal), with the hope that these choices will lead to a **global optimum** solution.

No.	Element	Description
1	**Greedy Choice Property**	A problem has this property if a **global optimal solution** can be obtained by choosing **local optimum solutions** at each step. Example: In the **Activity Selection Problem**, selecting the activity that finishes earliest leads to the best overall solution.
2	**Optimal Substructure**	A problem has optimal substructure if an **optimal solution to the problem contains optimal solutions to its subproblems**. Example: In **Shortest Path Problems (Dijkstra's Algorithm)**, the shortest path between two vertices includes the shortest paths within it.
3	**Feasibility Check**	After making a choice, we must ensure that the **current solution remains valid** (feasible). Example: In **Knapsack Problem**, an item can only be added if it doesn't exceed the total capacity.
4	**Solution Space**	Represents all possible solutions of the problem. The algorithm moves through this space making greedy choices to find the optimal one.
5	**Objective Function**	A function used to evaluate the total benefit or cost of a solution. The greedy choice is made based on **maximizing or minimizing** this objective function.

Fractional Knapsack Problem

The **Fractional Knapsack Problem** is a **Greedy Algorithm** problem in which:

- You are given a set of items, each with a **weight (w_i)** and a **value (v_i)**.
- You have a knapsack (bag) with a maximum capacity **W**.
- You can take **fractions of items** (i.e., split items) to maximize the **total value** in the knapsack.

Objective

To maximize

$$\text{Total Value} = \sum(\text{fraction of item} \times \text{value of item})$$

subject to

$$\sum(\text{fraction of item} \times \text{weight of item}) \leq W$$

Algorithm

FractionalKnapsack(value[], weight[], n, W)
1. Calculate ratio[i] = value[i] / weight[i] for each item
2. Sort items in descending order of ratio[i]
3. Initialize totalValue = 0
4. For i = 0 to n-1:
if weight[i] <= W:
W = W - weight[i]

totalValue += value[i]
else:
totalValue += value[i] * (W / weight[i])
break

5. **Return totalValue**

Example

Item	Value (v_i)	Weight (w_i)	v_i / w_i
1	60	10	6
2	100	20	5
3	120	30	4

Knapsack capacity (W) = 50

Step 1: Sort items by v_i / w_i

Order → Item 1 (6), Item 2 (5), Item 3 (4)

Step 2: Pick items greedily

- Take **Item 1** → weight = 10, remaining W = 40
  Total value = 60
- Take **Item 2** → weight = 20, remaining W = 20
  Total value = 60 + 100 = 160
- Take **2/3 of Item 3** → (20/30 = 0.66 fraction)
  Added value = 120 × 0.66 = 80
  Total value = 160 + 80 = **240**

Maximum value = 240

Huffman codes

Huffman Coding is a **greedy algorithm** used for **lossless data compression**. It reduces the average number of bits used to represent characters in a file by assigning **shorter codes** to **frequent characters** and **longer codes** to **less frequent characters**.

Algorithm

HuffmanCoding(characters[], frequency[])

1. Create a leaf node for each character and insert all nodes into a min-priority queue.

2. While there is more than one node in the queue:

a. Remove the two nodes with the smallest frequencies.

b. Create a new internal node with frequency = sum of the two.

c. Set the two removed nodes as left and right children.

d. Insert the new node back into the queue.

3. The remaining node in the queue is the root of the Huffman Tree.

4. Traverse the tree:

- Assign '0' for left edge and '1' for right edge.

- Record the code for each character.

Example

Character	Frequency
A	5
B	9
C	12
D	13
E	16
F	45

Step 1: Arrange in ascending order by frequency

A(5), B(9), C(12), D(13), E(16), F(45)

Step 2: Combine two smallest each time

1 Combine A(5) + B(9) → Node1(14)
→ Remaining: C(12), D(13), E(16), F(45), Node1(14)

2 Combine C(12) + D(13) → Node2(25)
→ Remaining: E(16), F(45), Node1(14), Node2(25)

3 Combine Node1(14) + E(16) → Node3(30)
→ Remaining: Node2(25), F(45), Node3(30)

4 Combine Node2(25) + Node3(30) → Node4(55)
→ Remaining: F(45), Node4(55)

5 Combine F(45) + Node4(55) → Root(100)

Step 3: Generate Codes

Character	Huffman Code
F	0
C	100
D	101
A	1100
B	1101
E	111

Result

Character	Frequency	Code	Bits Used
A	5	1100	20
B	9	1101	36
C	12	100	36
D	13	101	39
E	16	111	48
F	45	0	45
Total	**100**		**224 bits**

If fixed-length coding (3 bits per character) were used:
→ **100 × 3 = 300 bits**

Module-3

Data structures for Disjoint Sets

Disjoint set operations

Disjoint Set / Union-Find

A **Disjoint Set** (or Union-Find) is a **data structure** that keeps track of a **collection of disjoint (non-overlapping) sets**.
It supports **efficient operations** to **union sets** and **find which set an element belongs to**.

Disjoint sets are widely used in:

- **Kruskal's Minimum Spanning Tree (MST) algorithm**
- **Network connectivity**
- **Image processing**

Operations on Disjoint Sets

1. MakeSet(x)

- Create a **new set** containing the single element x.
- Each element initially is the **parent of itself**.

MakeSet(x):

parent[x] = x

rank[x] = 0 // optional, for optimization

2.Find(x)

- Find the **representative (or parent)** of the set containing x.
- Determines whether two elements are in the **same set**.

Find(x):
if parent[x] != x:
parent[x] = Find(parent[x]) // Path compression
return parent[x]

Path Compression: Makes all nodes on the path point directly to the root.
Improves time complexity to nearly O(1) per operation.

3. Union(x, y)

- Combine the sets containing x and y into a **single set**.

Union(x, y):

rootX = Find(x)

rootY = Find(y)

if rootX == rootY:

return // already in same set

// Union by rank

if rank[rootX] < rank[rootY]:

parent[rootX] = rootY

else if rank[rootX] > rank[rootY]:

parent[rootY] = rootX

else:

parent[rootY] = rootX

rank[rootX] += 1

Example

Suppose we have 5 elements: {1, 2, 3, 4, 5}

1. **MakeSet for each element:**
    - Sets: {1}, {2}, {3}, {4}, {5}
2. **Union(1, 2)** → {1,2}, {3}, {4}, {5}
3. **Union(3, 4)** → {1,2}, {3,4}, {5}
4. **Union(2, 3)** → {1,2,3,4}, {5}
5. **Find(4)** → returns representative of set → 1 (or 3 depending on union)

Now we can check **if two elements are in the same set**:

- Find(1) == Find(4) → True
- Find(1) == Find(5) → False

Linked-list representation of disjoint sets

Instead of using an array for parent pointers, **disjoint sets** can be represented using **linked lists**, where:

- Each **set** is a **linked list of elements**.
- Each node contains:
    - **Data/element value**
    - **Pointer to next element**
- Each list has a **head pointer**, which acts as the **representative of the set**.

Example

Initial sets: {1}, {2}, {3}, {4}

1. **MakeSet** for each:

- 1 → 1 → NULL (rep=1)
- 2 → 2 → NULL (rep=2)
- 3 → 3 → NULL (rep=3)
- 4 → 4 → NULL (rep=4)

2. **Union(1,2)** → Merge 2 into 1:

- 1 → 2 → NULL (rep=1 for both)

3. **Union(3,4)** → Merge 4 into 3:

- 3 → 4 → NULL (rep=3 for both)

4. **Union(2,3)** → Merge 3→4 into 1→2:

- 1 → 2 → 3 → 4 → NULL (rep=1 for all)

Find(4) → returns 1 (representative of the set).

Disjoint-set forests. Graph Algorithms:

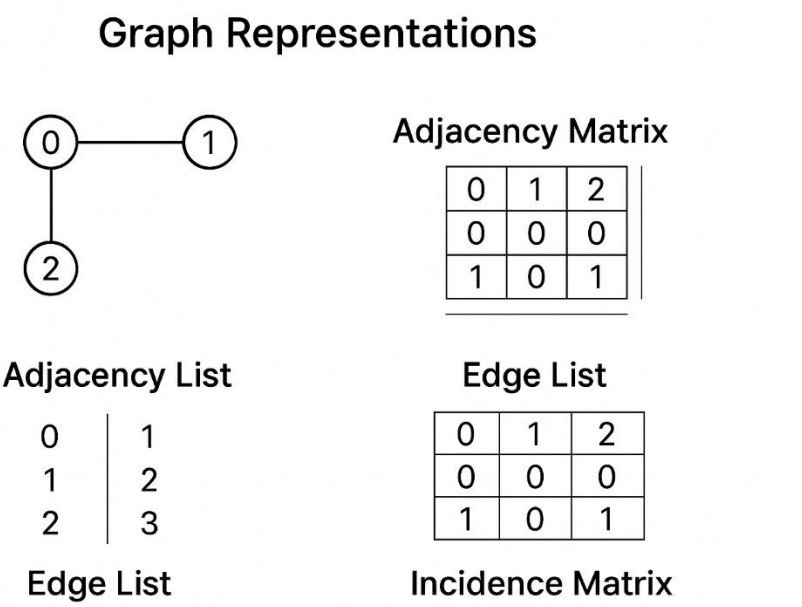
Elementary Graph Algorithms

Representations of graphs:

A **graph** is a set of **vertices (nodes)** and **edges (connections)**.

Graph representations define how we **store graphs in memory** for efficient computation.

- **Adjacency Matrix:**
- This representation uses a 2D array (matrix) where both rows and columns represent the vertices of the graph.
- An entry matrix[i][j] indicates the presence or absence of an edge between vertex i and vertex j.
- For unweighted graphs, a 1 typically denotes an edge and 0 denotes no edge.
- For weighted graphs, matrix[i][j] stores the weight of the edge between i and j, or 0 (or infinity) if no edge exists.
- **Advantages:** Fast checking for the existence of an edge between two specific vertices.
- **Disadvantages:** Requires O(V^2) space, where V is the number of vertices, which can be inefficient for sparse graphs (graphs with few edges).
- **Adjacency List:**
- This representation uses an array of lists (or linked lists).
- Each index in the array corresponds to a vertex, and the list at that index contains all the vertices adjacent to it.
- For weighted graphs, the list elements can be pairs of (adjacent vertex, edge weight).
- **Advantages:** Space-efficient for sparse graphs, as it only stores existing edges.
- **Disadvantages:** Checking for the existence of an edge between two specific vertices might require traversing a list, which can be slower than an adjacency matrix.
- **Edge List:**
- This representation simply stores a list of all edges in the graph.
- Each element in the list represents an edge and typically contains the two vertices it connects (and its weight, if applicable).
- **Advantages:** Simple to implement and useful for algorithms that primarily process edges.
- **Disadvantages:** Finding all neighbors of a specific vertex can be inefficient, requiring iteration through the entire edge list.



Breadth-First Search (BFS)

Breadth-First Search is a **graph traversal algorithm** that explores all the **neighbors of a vertex before moving to the next level** of vertices.

It uses a **queue** data structure (FIFO order).

Algorithm: Breadth-First Search (BFS)

Input: A graph G(V, E) and a starting vertex s
Output: Vertices visited in BFS order

```
BFS(Graph G, Vertex s)
{
  create a queue Q
  mark s as visited
  enqueue s into Q

  while Q is not empty do
  {
    u = dequeue(Q)
    print(u)
    for each vertex v adjacent to u do
    {
      if v is not visited then
      {
        mark v as visited
        enqueue v into Q
      }
    }
  }
   }
```

Step	Action	Queue	Visited Order	
1	Enqueue A	[A]	A	**Final BFS**
2	Dequeue A → enqueue B, C	[B, C]	A	
3	Dequeue B → enqueue D, E	[C, D, E]	A, B	
4	Dequeue C → enqueue F	[D, E, F]	A, B, C	
5	Dequeue D	[E, F]	A, B, C, D	
6	Dequeue E	[F]	A, B, C, D, E	
7	Dequeue F	[]	A, B, C, D, E, F	

Traversal Order:
A → B → C → D → E → F

Time Complexity

- **O(V + E)**
  where V = number of vertices, E = number of edges.

Depth First Search (DFS)

Depth First Search (DFS)** is a graph traversal algorithm** that explores as far as possible along each branch before backtracking.
It is similar to **preorder traversal in trees**.

DFS can be implemented using:

Recursion, or

Stack (explicit data structure)

Algorithm (Recursive Implementation)

```
DFS(G, v):
  mark v as visited
  for each vertex u adjacent to v:
    if u is not visited:
        DFS(G, u)
```

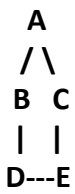
Algorithm (Stack-based Implementation)

```
DFS_Using_Stack(G, start):
  create an empty stack S
  push start vertex to S
  mark start as visited
```

```
while S is not empty:
   v = top element of S
   print v
   pop v from S

   for each adjacent vertex u of v:
      if u is not visited:
         mark u as visited
            push u to S
```

Example:



Graph Representation (Adjacency List):

Vertex	**Adjacent Vertices**
A	B, C
B	A, D
C	A, E
D	B, E
E	C, D

Step-by-Step DFS Traversal (starting from A)

Step 1: Start at A

Visited = {A}

Step 2: Go to adjacent B

Visited = {A, B}

Step 3: From B, go to D

Visited = {A, B, D}

Step 4: From D, go to E

Visited = {A, B, D, E}

Step 5: From E, go to C
Visited = {A, B, D, E, C}
Step 6: All vertices visited, DFS complete.

DFS Traversal Order:
A → B → D → E → C

Time Complexity

- **Using adjacency list:** O(V + E)
  (V = number of vertices, E = number of edges)
- **Using adjacency matrix:** O(V^2)

Minimum Spanning Trees

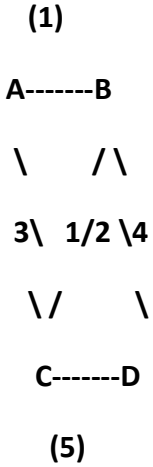
Kruskal Algorithm:

Kruskal's algorithm is a **greedy algorithm** used to find the **Minimum Spanning Tree (MST)** of a **connected, weighted, undirected graph**.

An **MST** connects all the vertices with the **minimum possible total edge weight** and **without any cycles**.

Algorithm
KRUSKAL(G):
1. **A = Ø**
2. **for each vertex v in G:**
   MAKE-SET(v)
3. **sort all edges of G in non-decreasing order by weight**
4. **for each edge (u, v) in sorted edges:**
   if FIND-SET(u) ≠ FIND-SET(v):
   A = A ∪ {(u, v)}
   UNION(u, v)
5. **return A**

Example

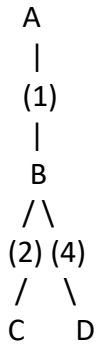


Step 1: Represent Edges and Weights

Edge	Weight
A–B	1
B–C	2
A–C	3
B–D	4
C–D	5

Step 2: Sort edges in non-decreasing order
(A-B) = 1
(B-C) = 2
(A-C) = 3
(B-D) = 4
(C-D) = 5
MST Edges: (A-B), (B-C), (B-D)

Minimum Spanning Tree



Total Minimum Cost = 1 + 2 + 4 = 7

Time Complexity- **O(E log V)**

Prim's Algorithm

Prim's Algorithm is a **greedy algorithm** that finds the **Minimum Spanning Tree (MST)** for a **weighted, connected, undirected graph**.

It starts from a **single vertex** and **grows** the MST one edge at a time by **adding the smallest possible edge** that connects a **vertex inside the tree** to a **vertex outside the tree**.

Algorithm

PRIM(G):

1. Choose any vertex as the starting vertex

2. Initialize MST = {start}

3. While (MST has less than V vertices):

Find the edge (u, v) with minimum weight

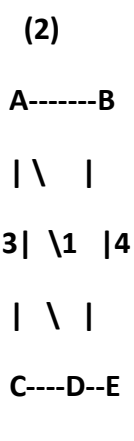
such that u ∈ MST and v ∉ MST

Add edge (u, v) to MST

4. **Return MST**

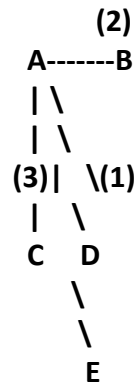
Example

Consider the following graph:



Graph Representation (Adjacency List):

Edge	Weight
A–B	2
A–C	3
A–D	1
B–E	4
D–E	5
C–D	6



MST Edges:
(A–D), (A–B), (A–C), (B–E)

Total Minimum Cost = 1 + 2 + 3 + 4 = 10

Single Source Shortest Paths
The Bellman-Ford Algorithm:
The **Bellman–Ford algorithm** is a **shortest path algorithm** used to find the **shortest distance** from a **single source vertex** to **all other vertices** in a **weighted graph**.

Unlike Dijkstra's algorithm, **Bellman–Ford can handle negative edge weights**.

if dist[u]+w(u,v)<dist[v], then dist[v]=dist[u]+w(u,v)

Algorithm

BELLFORD(G, source):

1. for each vertex v in G:

dist[v] = ∞

dist[source] = 0

2. repeat (V - 1) times:

for each edge (u, v) with weight w:

if dist[u] + w < dist[v]:

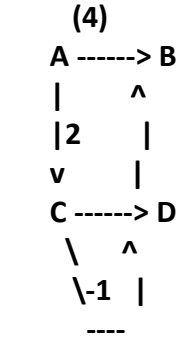
dist[v] = dist[u] + w

3. for each edge (u, v) with weight w:

if dist[u] + w < dist[v]:

print "Graph contains a negative weight cycle"

Example



Edges and Weights:

Edge	**Weight**
A → B	4
A → C	2
C → D	-1
D → B	3

Final Shortest Distances from Source A:

Vertex	**Distance**
A	0
B	4
C	2
D	1

Time and Space Complexity

O(V × E)

Space O(V)

Dijkstra's algorithm:

Dijkstra's algorithm is a **greedy algorithm** used to find the **shortest path** from a **single source vertex** to **all other vertices** in a **weighted graph** (with **non-negative edge weights**).

Algorithm.

Algorithm

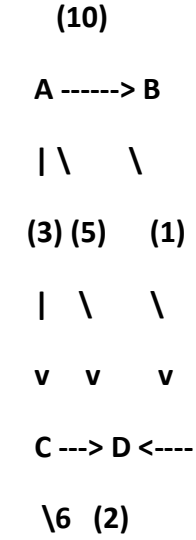
DIJKSTRA(G, source):

1. for each vertex v in G:
dist[v] = ∞
visited[v] = false
dist[source] = 0

2. for i = 1 to V - 1:
u = vertex with minimum dist[u] among unvisited vertices
visited[u] = true
for each neighbor v of u:
if (!visited[v] && dist[u] + weight(u,v) < dist[v]):
dist[v] = dist[u] + weight(u,v)

3. print dist[]

Example



Shortest Paths from A:

- A–C = 3
- A–C–B = 4
- A–D = 5
- A–C–B–E = 5

Time Complexity:

O((V + E) log V)

All-Pairs Shortest Paths

The Floyd-Warshall Algorithm:

The **Floyd–Warshall algorithm** is a **dynamic programming algorithm** used to find the **shortest paths between all pairs of vertices** in a **weighted graph**.

It works for **directed** as well as **undirected graphs**, and it can also handle **negative edge weights**, provided **no negative cycles** exist.

The algorithm considers all possible **intermediate vertices** one by one and tries to improve the shortest distance between every pair of vertices (i, j).

If the shortest path from vertex i to vertex j **passes through** an intermediate vertex k, then:

dist[i][j]=min⁡(dist[i][j], dist[i][k]+dist[k][j])dist[i][j] = \min(dist[i][j],\; dist[i][k] + dist[k][j])dist[i][j]=min(dist[i][j],dist[i][k]+dist[k][j])

Algorithm

```
FLOYD_WARSHALL (dist[][]):
for k = 1 to V:
   for i = 1 to V:
      for j = 1 to V:
         if dist[i][k] + dist[k][j] < dist[i][j]:
            dist[i][j] = dist[i][k] + dist[k][j]
```

Module-4
Maximum Flow

Flow Networks:

A **flow network** is a directed graph **G = (V, E)** where:

- Each edge **(u, v) ∈ E** has a **non-negative capacity c(u, v) ≥ 0**
- There are two special vertices:
    - **Source (s)** — where flow originates
    - **Sink (t)** — where flow terminates
- Flow **f(u, v)** represents how much material moves along edge **(u, v)**

Residual Graph

A **residual graph** represents the remaining capacity of each edge in the network.

For every edge (u, v):

cf(u,v)=c(u,v)−f(u,v)c_f(u, v) = c(u, v) - f(u, v)cf(u,v)=c(u,v)−f(u,v)

and

cf(v,u)=f(u,v)c_f(v, u) = f(u, v)cf(v,u)=f(u,v)

Ford–Fulkerson Algorithm

The Ford–Fulkerson algorithm is one of the most widely used techniques for solving the maximum flow problem in a flow network. The goal of this problem is to determine the largest possible amount of flow **that can be sent from a** source node (s) **to a** sink node (t) **in a** directed, weighted graph**,** while ensuring that the flow through each edge does not exceed its assigned capacity.

The core idea of the algorithm is based on **iterative improvement**. In each iteration, it searches for an **augmenting path**—a path from the source to the sink in the **residual graph**. The residual graph represents the remaining capacity of each edge after considering the current flow. Once such a path is found, the algorithm augments (increases) the flow along this path by the **minimum residual capacity** of the edges that form the path (often referred to as the **bottleneck capacity**).

This process continues until **no more augmenting paths** exist in the residual graph, meaning that no additional flow can be pushed from the source to the sink. The total flow in the network at this point represents the **maximum flow**.

Problem Statement

Given a **directed graph** that models a **flow network**, where each edge has an associated **capacity**, and two specific vertices — a **source (s)** and a **sink (t)** — the objective is to determine the **maximum flow** possible from **s** to **t** under the following conditions:

1. The **flow on any edge** must not exceed the **capacity** assigned to that edge.
2. For every vertex except the **source (s)** and **sink (t)**, the **total incoming flow** must be **equal to the total outgoing flow** (flow conservation property).

FORD-FULKERSON(G, s, t)
1. Initialize f(u, v) = 0 for all edges
2. While there exists an augmenting path P from s to t:
c_f(P) = min{ c_f(u, v) | (u, v) in P }
for each edge (u, v) in P:
f(u, v) = f(u, v) + c_f(P)
f(v, u) = -f(u, v)
4. Return total flow = sum of f(s, v) for all v

Ford-Fulkerson Algorithm
The following is simple idea of Ford-Fulkerson algorithm:

1. Start with initial flow as 0.
2. While there exists an augmenting path from the source to the sink:
   - Find an augmenting path using any path-finding algorithm, such as breadth-first search or depth-first search.
   - Determine the amount of flow that can be sent along the augmenting path, which is the minimum residual capacity along the edges of the path.
   - Increase the flow along the augmenting path by the determined amount.
3. Return the maximum flow.

Polynomials

A **polynomial** is a mathematical expression consisting of variables (also called indeterminates) and coefficients, combined using addition, subtraction, and multiplication**.**
A polynomial can have one or several terms. Each term is a product of a constant and a variable raised to an exponent.

$$P(x)=anxn+an−1xn−1+\cdots+a1x+a0$$

Where:

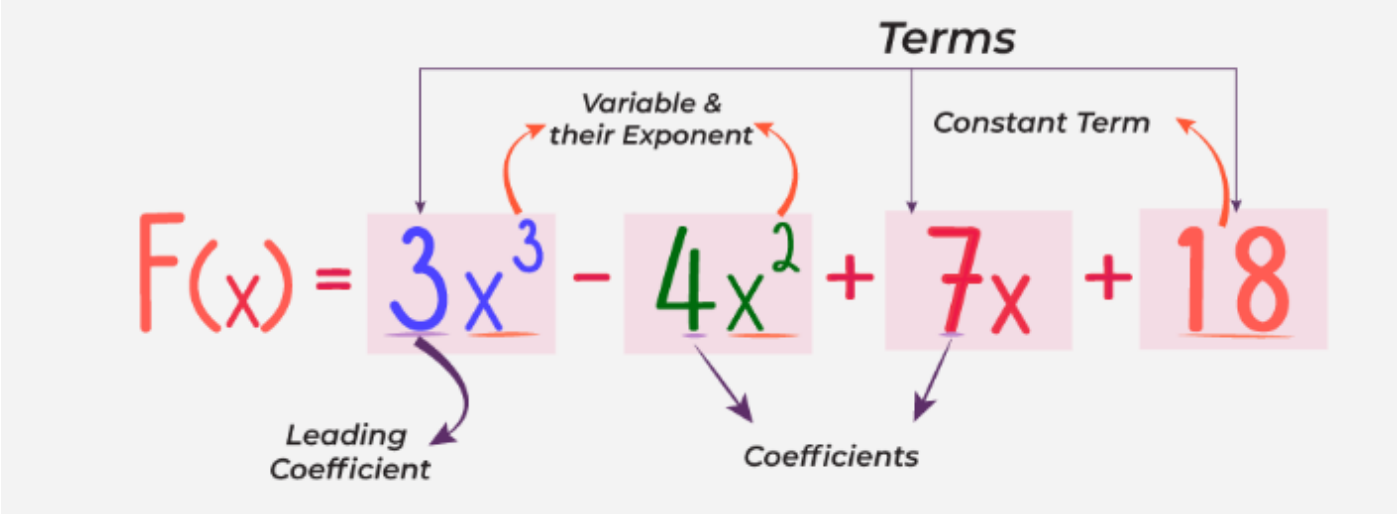
- x is the **variable**
- a0,a1,…,an are **coefficients** (real or complex numbers)
- n is a **non-negative integer**, called the **degree** of the polynomial

Degree of a Polynomial

- The **degree** is the **highest power of the variable** in the polynomial with a non-zero coefficient.
- Example:

$$P(x)=5x4+3x2-7x+2$$

Degree = 4 (highest exponent)



Types of Polynomials

1. **Based on Degree**

0	Constant polynomial	777
1	Linear polynomial	2x+32x + 32x+3
2	Quadratic polynomial	x2+5x+6x^2 + 5x + 6x2+5x+6
3	Cubic polynomial	x3−2x2+x−5x^3 - 2x^2 + x - 5x3−2x2+x−5
n	n-th degree polynomial	xn+2xn−1+⋯+1x^n + 2x^{n-1} + \dots + 1xn+2xn−1+⋯+1

2. **Based on Number of Terms**

Terms	Name	Example
1	Monomial	5x35x^35x3
2	Binomial	x2+3xx^2 + 3xx2+3x
3	Trinomial	x3+2x2−5xx^3 + 2x^2 - 5xx3+2x2−5x
>3	Multinomial	x4+3x3−2x2+x−1x^4 + 3x^3 - 2x^2 + x - 1x4+3x3−2x2+x−1

NP-Completeness and reducibility:

In computational theory, problems are classified based on how efficiently they can be solved.
Two important complexity classes are **P** and **NP**.

- **P (Polynomial Time):**
  Problems that can be solved efficiently (in polynomial time) by a deterministic algorithm.
- **NP (Nondeterministic Polynomial Time):**
  Problems for which a *proposed solution* can be **verified** efficiently (in polynomial time), even if finding the solution may not be efficient.

NP Problems

A problem is in **NP** if:

- A proposed solution can be verified in polynomial time.
- Alternatively, there exists a *nondeterministic* algorithm that can find a solution in polynomial time.

Example:
Given a set of numbers, check whether there exists a subset that adds up to a particular sum (Subset Sum Problem).
We can easily verify a given subset's sum in polynomial time.

NP-Hard Problems

- A problem is said to be **NP-Hard** if every problem in NP can be **reduced** to it in polynomial time.
- NP-Hard problems are at least as hard as NP problems.
- They **do not have to be in NP** (i.e., they might not even have efficiently verifiable solutions)

NP-Complete Problems

A problem is **NP-Complete (NPC)** if it satisfies two conditions:

1. It belongs to **NP**.
2. Every problem in NP can be **polynomially reduced** to it.

Reducibility

Reducibility is the process of converting one problem into another problem in polynomial time.

- If problem A can be reduced to problem B (denoted as **A ≤p B**),
  then if B can be solved efficiently, A can also be solved efficiently.
- Polynomial-time reducibility is used to **prove NP-Completeness**.

Example:
If we can reduce **3-SAT** to **Clique** in polynomial time,
and since 3-SAT is NP-Complete, Clique is also NP-Complete.

Steps to Prove a Problem is NP-Complete

To show a problem **X** is NP-Complete:

1. **Show X ∈ NP**
   (i.e., we can verify a solution in polynomial time)
2. **Choose a known NP-Complete problem Y.**
3. **Reduce Y to X in polynomial time**
   (Y ≤p X)
4. Conclude that X is NP-Complete.

Example of NP-Complete Problems

Some well-known NP-Complete problems are:

- **Boolean Satisfiability Problem (SAT)** — The first problem proven NP-Complete.
- **3-SAT Problem**
- **Subset Sum Problem**
- **Hamiltonian Cycle Problem**
- **Travelling Salesman Problem (TSP)** (decision version)
- **Clique Problem**
- **Vertex Cover Problem**
- **Graph Coloring Problem**

Approximation Algorithms:

The Vertex Cover Problem:

The **Vertex Cover Problem** is one of the fundamental **NP-Complete** problems in graph theory and computer science.
It plays an important role in **network design**, **resource allocation**, and **computational complexity** studies.

Given an **undirected graph** G=(V,E)G = (V, E)G=(V,E), a **vertex cover** is a subset of vertices V′⊆VV' \subseteq VV′⊆V such that **every edge (u, v) ∈ E** has at least one endpoint in V′V'V′.

Example:

V = {A, B, C, D}

E = {(A,B), (A,C), (B,C), (C,D)}

Possible Vertex Covers:

- {A, C}
- {B, C}
- {A, B, D}

Minimum Vertex Cover:
The smallest vertex cover is {B, C}, with size = 2.

The Travelling salesman problem:

“Given a list of cities and the distances between each pair, what is the shortest route that visits each city exactly once and returns to the starting city?"

Example with 4 Cities

Let's say a salesman has to visit **4 cities**: A, B, C, and D.
The distances between them are as follows:

From → To	A	B	C	D
A	0	10	15	20
B	10	0	35	25
C	15	35	0	30
D	20	25	30	0

Goal:

Find the **shortest route** that:

1. Starts at one city (say A),
2. Visits all other cities exactly once,
3. Returns back to A.

All Possible Routes (Starting at A)

1. A → B → C → D → A = 10 + 35 + 30 + 20 = **95**
2. A → B → D → C → A = 10 + 25 + 30 + 15 = **80**
3. A → C → B → D → A = 15 + 35 + 25 + 20 = **95**
4. A → C → D → B → A = 15 + 30 + 25 + 10 = **80**
5. A → D → B → C → A = 20 + 25 + 35 + 15 = **95**
6. A → D → C → B → A = 20 + 30 + 35 + 10 = **95**

All Possible Routes (Starting at A)

1. A → B → C → D → A = 10 + 35 + 30 + 20 = **95**
2. A → B → D → C → A = 10 + 25 + 30 + 15 = **80**
3. A → C → B → D → A = 15 + 35 + 25 + 20 = **95**
4. A → C → D → B → A = 15 + 30 + 25 + 10 = **80**
5. A → D → B → C → A = 20 + 25 + 35 + 15 = **95**
6. A → D → C → B → A = 20 + 30 + 35 + 10 = **95**

Best Route

- Routes 2 and 4 both have the shortest distance: **80**
- Example route: **A → B → D → C → A**

Set Covering Problem

Given a **universal set** of elements and a collection of **subsets**, the goal is to **select the smallest number of subsets** such that every element in the universal set is **covered** (i.e., included in at least one chosen subset).

Suppose you need to cover a universal set:

U = {1, 2, 3, 4, 5}

You have the following subsets:

- **S1 = {1, 2, 3}**
- **S2 = {2, 4}**
- **S3 = {3, 4}**
- **S4 = {4, 5}**
- **S5 = {5}**

Suppose you need to cover a universal set:

One possible minimal set cover:

- **S1** (covers 1, 2, 3)
- **S4** (covers 4, 5)

S1 ∪ S4 = {1, 2, 3, 4, 5} → All elements covered
Only **2 subsets used** — which is minimal in this case

Subset Sum Problem

Given a **set of integers** and a **target sum**, determine whether there is a **subset** of the given set whose **sum equals the target**.

Suppose you have:

- **Set** = {3, 34, 4, 12, 5, 2}
- **Target Sum** = 9

One valid subset is:
{4, 5} → 4 + 5 = 9