Algorithms

Reference Books:

1. CLR Introduction to Algorithms - 3rd Edition, By Thomas H. Cormen

This PDF contains the notes from the standards books and are only meant for GATE CSE aspirants.

Notes Compiled By-
Manu Thakur
Mtech CSE, IIT Delhi
worstguymanu@gmail.com
https://www.facebook.com/Worstguymanu

Algorithms

An **algorithm** is any well-defined computational procedure that takes some value, or set of values, as **input** and produces some value, or set of values, as **output**. An algorithm is thus a sequence of computational steps that transform the input into the output.

A **data structure** is a way to store and organize data in order to facilitate access and modifications. **No single data structure works well for all purposes**, and so it is important to know the strengths And limitations of several of them.

The running-time of an algorithm on a particular input is the number of primitive operations or steps executed.

Worst Case T.C the worst-case running time of an algorithm gives us an upper bound on the running time for any input. Knowing it provides a guarantee that the algorithm will never take any longer.

Advantage of Sorting:-

1. Unsorted array search = O(n)
2. Sorted array search = O(logn)
3. Finding Median, sorted array = O(1)
4. Finding Median, unsorted array = O(n)
5. Building a frequency table
6. Checking duplicates

Insertion Sort

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

Output: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

The numbers that we wish to sort are also known as the **keys**.

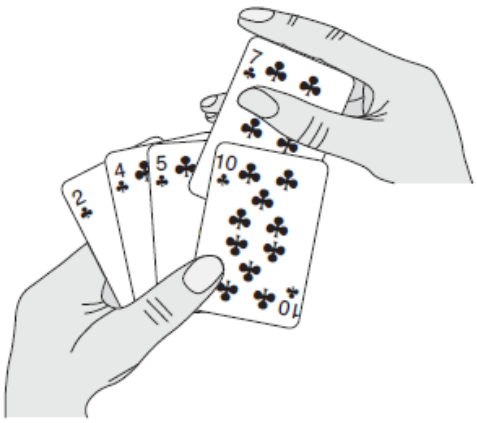
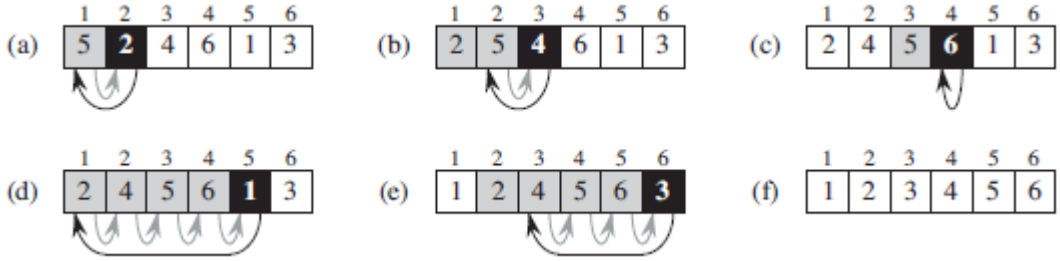


Figure 2.1 Sorting a hand of cards using insertion sort.

Insertion sort works the way we sort a hand of playing cards. We start with an empty left hand and the cards face down on the table. We then remove one card at a time from the table and **insert it into the correct position in the left hand**. To find the correct position for a card, we compare it with

each of the cards already in the hand, **from right to left**. At all times, t**he cards held in the left hand are sorted**, and these cards were originally the top cards of the pile on the table.




```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

1. **Array starting with index 1.**
2. Starting from index 2 upto length of array n
3. A[j] will be copied into key.
4. From i=0 to j-1 array is already sorted.
5. While i > 0 && key < A[i]
6. Shifting elements to right A[i+1] = A[i]
7. Decreasing index i.
8. Storing key at correct position.

Loop Invariant:

Initialization: It is true prior to the first iteration of the loop.

Maintenance: If it is true before an iteration of the loop, it remains true before the next iteration.

Termination: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

Note:-

1. Insertion sort, takes time roughly equal to **c1*n^2** to sort n items, where c1 is a constant that does not depend on n.
2. Merge sort, takes time roughly equal to **c2*logn** (base2), and c2 is another constant that also does not depend on n.
3. Insertion sort can take different amounts of time to sort two input sequences of the same size depending on how nearly sorted they already are.

Analysis of insertion sort:

Insertion sort typically has a smaller constant factor than merge sort, so that c1 < c2 Hence, **Insertion sort is better than merge sort for small input**.

Worst-case and average-case analysis

The running time of the algorithm is the sum of running times for each statement executed;

The **best case** occurs if the array is already sorted. Line 5 will execute (n-1) times and while loop will 0 times. **Best T.C will be O(n)**

The worst case occurs if the array is sorted in **reverse order**. There will be **n(n-1)/2 comparisons** and **n(n-1)/2 shifts**. Overall worst case T.C will be **O(n^2).**

Note:-

1. If we replace linear search with Binary Search in sorted part, Insertion sort will still take O(n^2) T.C due to n(n-1)/2 shifts.
2. If we use Linked List in place of array, Insertion Sort will still take O(n^2) because of n(n-1)/2 comparisons.

Properties:-

1. Insertion sort is an efficient algorithm to sort small number of elements
2. The algorithm sorts the input numbers **in place**. Input array A contains the sorted output sequence when the INSERTION-SORT is finished.
3. Insertion sort is **stable** sorting. A sorting algorithm is said to be **stable** if **two objects with equal keys** appear in the same order in sorted output as they appear in the input unsorted array.
4. Best case O(n) when array is already sorted
5. Worst case O(n^2) when array is sorted in reverse order.
6. Insertion sort is an **Adaptive** sorting.

Insertion Sort (Recursive Solution)

Sorted part | Unsorted Part

a. Assume A[0…….. i-1] is already sorted
b. A[I ….. n-1] yet to be sorted
c. Insert A[i] into A[0…. I-1]
d. Recursively sort A[i+1 …… n-1]
e. Base case = i=n-1

Recurrence relation

Since it takes theta(n) time in worst case to insert A[n] in the sorted array A[I …… n-1] we get the recurrence :

T(n) = T(n-1) + n if n>1 and O(1) if n=1

Inversions:-

Let A[1..n] be an array of n distinct numbers. **If i<j and A[i]>A[j]**, then the pair (i,j) is called an **inversion** of A.

Suppose A={2,3,8,6,1}, there are 5 inversions ⟨2,1⟩ , ⟨3,1⟩, ⟨8,6⟩, ⟨8,1⟩ and ⟨6,1⟩. If array is sorted in reverse order then there will be **maximum number** of inversions (n-1) + n(-2) + …. 1 = **n(n-1)/2**

Average number of inversions will be (0 + n(n-1)/2)/2 = **n(n-1)/4**

Relationship with insertion sort:
Insertion sort performs the body of the inner loop **once for each inversion**. Thus the **running time of insertion sort is Theeta(n + d) where d is the number of inversions**.

a. In best case there will be no inversion.
b. In worst case, there will be n(n-1)/2 inversions.

Time Complexity

1. Merge procedure can be modified to count number of inversions in **Theeta(nlogn)** time.
2. For each element, count number of elements which are on right side of it and are smaller than it. It requires **O(n^2)** time.

Note: - among O(n^2) sorts, Insertion sort is usually better than selection sort and both are better than Bubble Sort.

Selection Sort:-

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

1. The subarray which is already sorted.
2. Remaining subarray which is unsorted.

In every iteration of selection sort, the minimum element from the unsorted subarray is picked and moved to the sorted subarray.

Algorithm 3 Selection Sort

```
1: for i = 1 to n − 1 do
2:     min = i
3:     for j = i + 1 to n do
4:         // Find the index of the i^th smallest element
5:         if A[j] < A[min] then
6:             min = j
7:         end if
8:     end for
9:     Swap A[min] and A[i]
10: end for
```

1. For loop will run for n-1 times, as left nth element will be at its correct position.
2. Initially, we have considered first element as smallest element

3. Inner for loop will start from index 2 and go up to n
4. If any element is found smaller than our minimum element
5. Set min = index of smaller element
6. At the end swap them

This yields a running time of

$$\sum_{i=1}^{n-1} n-i = n(n-1) - \sum_{i=1}^{n-1} i = n^2 - n - \frac{n^2-n}{2} = \frac{n^2-n}{2} = \Theta(n^2).$$

T(n) = (n-1) + (n-2) + …… + 2 + 1 = n(n-1)/2 = O(n^2)

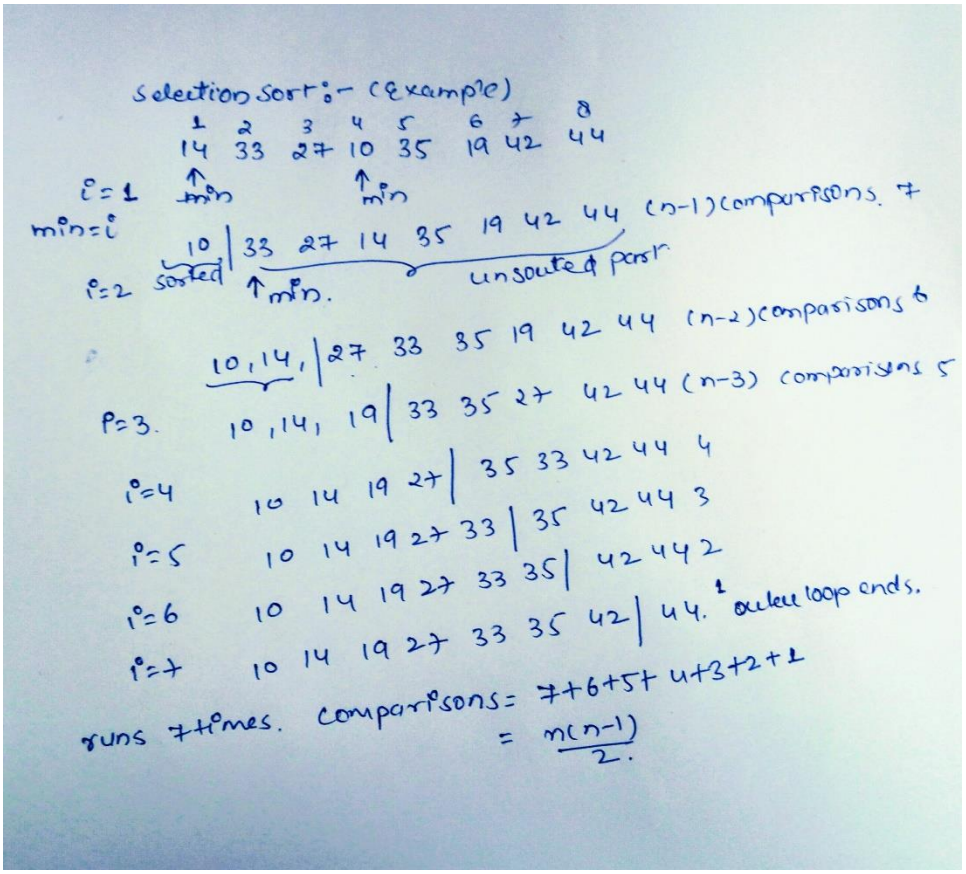
Properties:-

1. There is no best case. Best, worst, and average all are O(n^2)
2. Selection sort is not an **Adaptive** sorting algorithm.
3. Selection sort is **in-place** algorithm.
4. Selection Sort is **not stable** sorting. Given [2, 2, 1], the '2' will be replaced with 1.

Adaptive Sorting Algorithm:

A sorting algorithm falls into the adaptive sort family if it takes advantage of existing order in its input. It benefits from the pre-sortedness in the input sequence.
Example- Insertion sort.

Note: - we can have adaptive sorting versions of Heap Sort and Merge Sort.



Selection Sort Example

Selection Sort Recursive Solution:-

a. To sort A[i….. n-1] find minimum value in array and move to A[i]
b. Apply selection sort on A[i+1, n-1]
c. Base Case: do nothing if i=n-1, only on element is left in array

SelectionSort(A, i, n) // Array Name, starting index, total size

1. If(i>= n-1)
2. Return;
3. Lowest_key = i;
4. For(j=i+1 to n-1)
5. If (A[i] > A[j]) then
6. Lowest_key=j;
7. Swap(A[Lowest_key], A[i])
8. SelectionSort(A, i+1, n);

Analysis: n steps to find minimum and move to first position
T(n) = t(n-1) + n
T(n) = T(n-2) + (n-1) + n
.
.
.
= 1 + 2 + ….. + n = n(n+1)/2 = O(n^2)

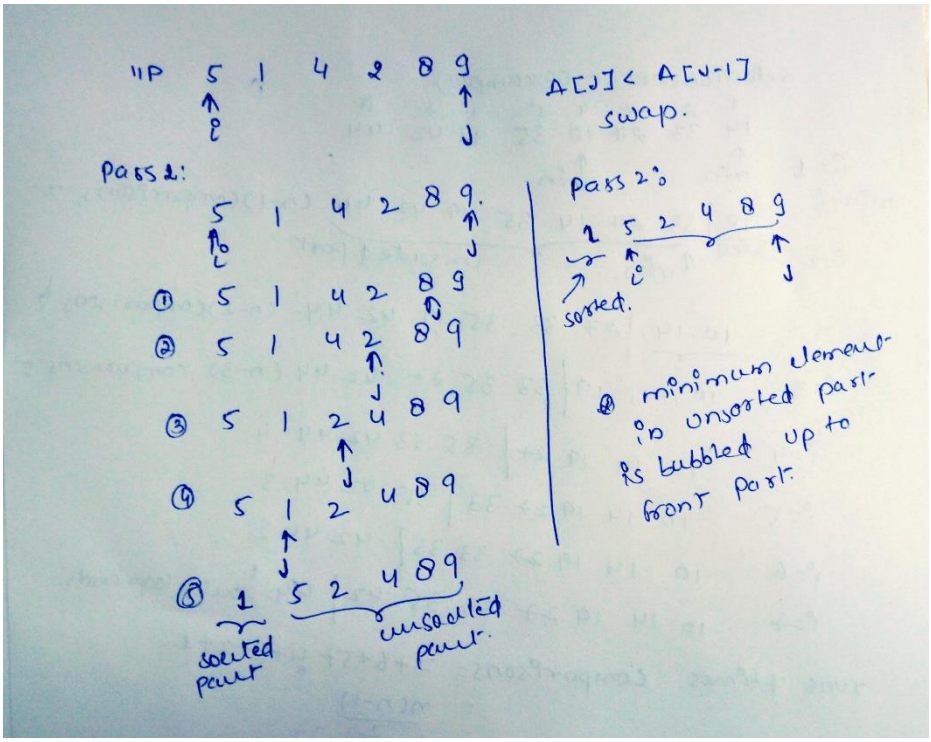
Bubble Sort:-

A simple sorting algorithm compares each pair of adjacent items and swaps them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted. It can be practical if the input is usually in sorted order but may occasionally have some out-of-order elements nearly in position.

```
BUBBLESORT(A)
1  for i = 1 to A.length − 1
2      for j = A.length downto i + 1
3          if A[j] < A[j − 1]
4              exchange A[j] with A[j − 1]
```

1. For loop will execute n-1 times from 1 to n-1
2. It will start from RHS upto i+1
3. If RHS element is less than LHS element
4. Will bubble up smallest element in unsorted part to the front.

Note: - If no swap happens in any pass we can immediately stop. We can use a flag to check this property. If Array is already sorted we can stop after 1 iteration only i.e O(n)



Properties:-

1. Worst case T.C is **O(n^2)**.
2. Best Case when array is already sorted **O(n)**.
3. Bubble sort is **in-place** sorting.
4. Bubble sort is a **stable** sort.

The divide-and-conquer approach

Recursive algorithms typically follow a **divide-and-conquer** approach: they break the problem into several sub-problems that are similar to the original problem but smaller in size, solve the sub-problems recursively, and then combine these solutions to create a solution to the original problem.

Divide the problem into a number of subproblems that are smaller instances of the same problem.

Conquer the sub-problems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.

Combine the solutions to the subproblems into the solution for the original problem.

When the subproblems are large enough to solve recursively, we call that the **recursive case**. Once the subproblems become small enough that we no longer recurse, we have gotten down to the **base Case**.

Recurrences

A **recurrence** is an equation or inequality that describes a function in terms of its Value on smaller inputs.

worst-case running time $T(n)$ of the MERGE-SORT procedure by the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 , \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 , \end{cases} \tag{4.1}$$

whose solution we claimed to be $T(n) = \Theta(n \lg n)$.

Recurrences can take many forms. For example, a recursive algorithm might divide subproblems into unequal sizes, such as a 2/3-to-1/3 split. If the divide and combine steps take linear time, such an algorithm would give rise to the recurrence $T(n) = T(2n/3) + T(n/3) + \Theta(n)$.

Master Method:-

The master method provides bounds for recurrences of the form:

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

Where a>=1 and b>1, and f(n) is a given function.
A recurrence of the form in above equation characterizes a **divide and conquer algorithm** that creates a subproblems, each of which is 1/b the size of the original problem, and in which **the divide and combine steps together take f(n) time**.

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

Extended Master Theorem

If the recurrence is of the form $T(n) = aT(\frac{n}{b}) + \Theta(n^k log^p n) where, a \geq 1, b > 1, k \geq 0$ and p is a real number.

1. if $a > b^k$, then T(n) = $\Theta(n^{log_b^a})$

2. if $a = b^k$
   a. If p > -1 then $T(n) = \Theta(n^{log_b^a} log^{p+1} n)$
   b. If p = -1 then $T(n) = \Theta(n^{log_b^a} loglogn)$
   c. If p < -1, then $T(n) = \Theta(n^{log_b^a})$

3. If $a < b^k$
   a. If $p \geq 0$, then $T(n) = \Theta(n^k log^p n)$
   b. If p < 0, then $T(n) = O(n^k)$

Occasionally, we shall see recurrences that are not **equalities** but rather **inequalities**,

ities, such as $T(n) \leq 2T(n/2) + \Theta(n)$. Because such a recurrence states only an upper bound on $T(n)$, we will couch its solution using O-notation rather than Θ-notation. Similarly, if the inequality were reversed to $T(n) \geq 2T(n/2) + \Theta(n)$, then because the recurrence gives only a lower bound on $T(n)$, we would use Ω-notation in its solution.

Strassen's algorithm for matrix multiplication

```
SQUARE-MATRIX-MULTIPLY(A, B)
1  n = A.rows
2  let C be a new n × n matrix
3  for i = 1 to n
4      for j = 1 to n
5          c_ij = 0
6          for k = 1 to n
7              c_ij = c_ij + a_ik · b_kj
8  return C
```

The SQUARE-MATRIX-MULTIPLY procedure takes **Θ(n^3)** time. There are three nested for loops.

A simple divide-and-conquer algorithm:

When we use a divide-and-conquer algorithm to compute the matrix product **C = A*B**, we assume that n is an exact power of 2 in each of the **n x n matrices**. We make this assumption because in each divide step, we will divide n x n matrices into four **n/2 x n/2** matrices, and by assuming that n is an Exact power of 2, we are guaranteed that as long as n >=2, the dimension n/2 is an integer.

Suppose that we partition each of A, B, and C into four $n/2 \times n/2$ matrices

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}, \tag{4.9}$$

so that we rewrite the equation $C = A \cdot B$ as

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}. \tag{4.10}$$

Equation (4.10) corresponds to the four equations

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}, \tag{4.11}$$
$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}, \tag{4.12}$$
$$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}, \tag{4.13}$$
$$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}. \tag{4.14}$$

Each of these four equations specifies two multiplications of n/2 x n/2 matrices and the addition of their n/2 x n/2 products.

Let T(n) be the time to multiply two nxn matrices using this procedure. In the base case, when n=1, we perform just the one scalar multiplication in. **T (1) = Θ (1)**

The recurrence for the running time of SQUARE-MATRIX-MULTIPLY-RECURSIVE is:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 8T(n/2) + \Theta(n^2) & \text{if } n > 1. \end{cases}$$

T(n) = Θ(n^3) Thus, this simple divide-and-conquer approach is no faster than the straightforward SQUARE-MATRIX-MULTIPLY procedure.

Strassen's method

Instead of performing eight recursive multiplications of n/2xn/2 matrices, **it performs only seven**. The cost of eliminating one matrix multiplication will be several new additions of n/2xn/2 matrices, but still only a constant number of additions.

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 7T(n/2) + \Theta(n^2) & \text{if } n > 1. \end{cases}$$

Thus, this solution has **time complexity of T(n) = Θ(n^log7)**. Or that is **T(n) = Θ(n^2.81)**.

Note:

1. **The best available algorithm to multiply two nxn matrices is O(n^2.37)**
2. **To multiply two matrices of order mxn and nxp orders, we need m*n*p multiplications and m*(n-1)*p additions.**

Changing variables:-

Sometimes, a little algebraic manipulation can make an unknown recurrence similar to one we have seen before.

$$T(n) = 2T\left(\lfloor \sqrt{n} \rfloor\right) + \lg n \,,$$

n=2^m and m=logn

$$T(2^m) = 2T(2^{m/2}) + m \,.$$

We can now rename $S(m) = T(2^m)$ to produce the new recurren

$$S(m) = 2S(m/2) + m \,,$$

The new recurrence has the same solution: **S(m)= O(mlogm)** changing back from S(m)to T(n) we obtain

$$T(n) = T(2^m) = S(m) = O(m \lg m) = O(\lg n \lg \lg n)$$

Que (CLR): Solve the recurrence $T(n) = 3T(\sqrt{n}) + \log n$ by making a change of variables.
Solution:

$$\begin{aligned} T(n) &= 3T(\sqrt{n}) + \lg n \qquad \text{rename } m = \lg n \\ T(2^m) &= 3T(2^{m/2}) + m \\ S(m) &= 3S(m/2) + m \\ S(m) &= \Theta(m^{\lg 3}) \\ T(n) &= \Theta(\lg^{\lg 3} n) \end{aligned}$$

T(n)=Θ(logn)^log3

The recursion-tree method for solving recurrences

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

For example, let us see how a recursion tree would provide a good guess for the recurrence

T (n) = 3T(n/4) + Θ(n^2)

Below figure shows how we derive the recursion tree **for T(n) =3T(n/4) + c.n^2** For convenience, we assume that n is an exact power of 4 so that all subproblem sizes are integers.

1. Part (a) of the figure shows T(n), which we expand in part (b) into an equivalent tree representing the recurrence.
2. The cn^2 term at the root represents the cost at the top level of recursion, and the three subtrees of the root represent the costs incurred by the subproblems of size n/4.
3. Part (c) shows this process carried one step further by expanding each node with cost T(n/4) from part (b).
4. The cost for each of the three children of the root is c.(n/4)^2.
5. We continue expanding each node in the tree by breaking it into its constituent parts as determined by the recurrence.

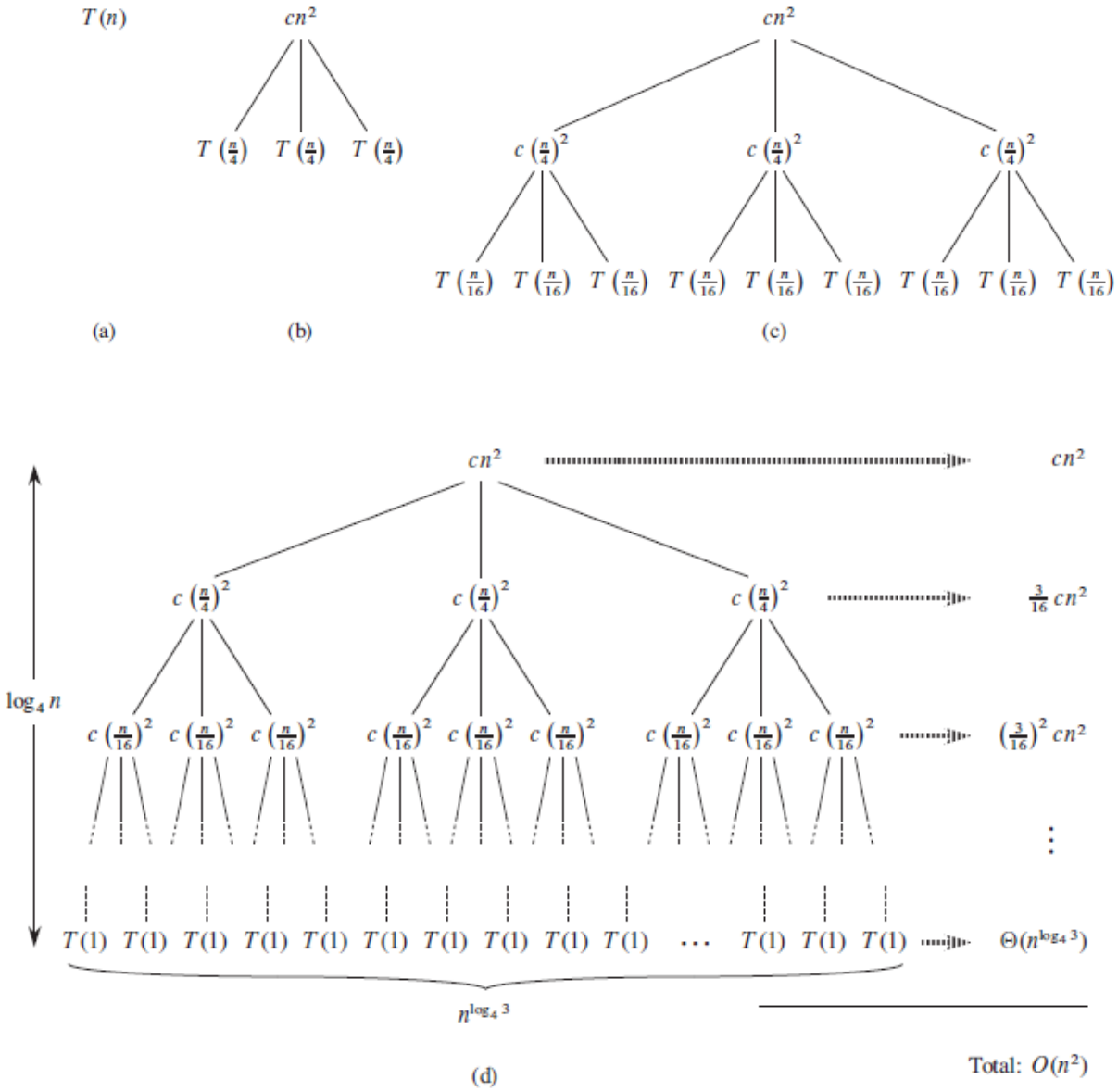


Figure 4.5 Constructing a recursion tree for the recurrence $T(n) = 3T(n/4) + cn^2$. Part (a) shows $T(n)$, which progressively expands in (b)–(d) to form the recursion tree. The fully expanded tree in part (d) has height $\log_4 n$ (it has $\log_4 n + 1$ levels).

In another example, Figure 4.6 shows the recursion tree for **T(n)= T(n/3) + T(2n/3) + O(n)**

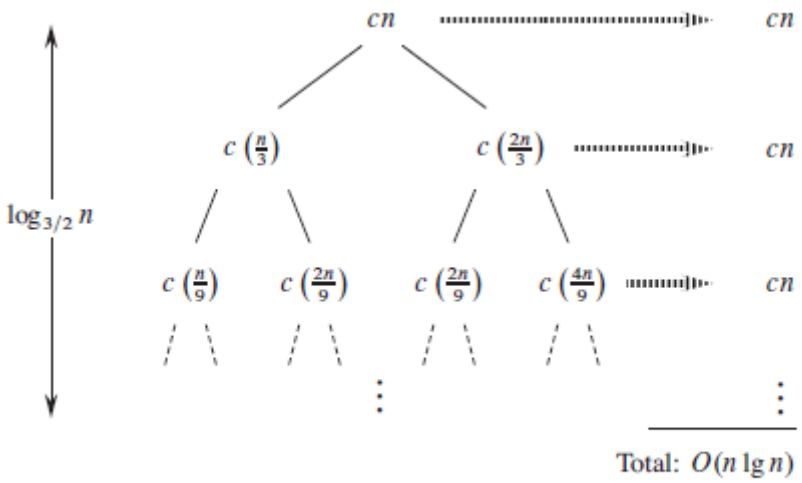


Figure 4.6 A recursion tree for the recurrence $T(n) = T(n/3) + T(2n/3) + cn$.

The longest simple path from the root to a leaf is $n \to (2/3)n \to (2/3)^2 n \to \cdots \to 1$. Since $(2/3)^k n = 1$ when $k = \log_{3/2} n$, the height of the tree is $\log_{3/2} n$.

Intuitively, we expect the solution to the recurrence to be at most the number of levels times the cost of each level, or **O(c*n)*logn base(3/2) = O(nlogn)**

=> SUBSTITUTION METHOD

$T(n) = 3T(n/4) + cn^2$ using $n^{\log_4 3} < n^2 = \Theta(n^2)$

$T(n/4) = 3T\left(\frac{n}{16}\right) + \left(\frac{n}{4}\right)^2$

$T\left(\frac{n}{16}\right) = 3T\left(\frac{n}{64}\right) + \left(\frac{n}{16}\right)^2$

$T(n) = 3\left(3T\left(\frac{n}{16}\right) + \left(\frac{n}{4}\right)^2\right) + n^2$

$= 3^2 T\left(\frac{n}{4^2}\right) + 3*\left(\frac{n}{4}\right)^2 + n^2$

$T(n) = 3^2\left[3T\left(\frac{n}{4^3}\right) + \left(\frac{n}{4^2}\right)^2\right] + 3*\frac{n}{4}$

$T(n) = 3^3 T\left(\frac{n}{4^3}\right) + 3^2\left(\frac{n}{4^2}\right)^2 + 3\left(\frac{n}{4}\right)^2 + 3^0\left(\frac{n}{4^0}\right)^2$

$T(n) = 3^4 T\left(\frac{n}{4^4}\right) + 3^3\left(\frac{n}{4^3}\right)^2 + 3^2\left(\frac{n}{4^2}\right)^2 + 3\left(\frac{n}{4}\right)^2 + 3^0\left(\frac{n}{4^0}\right)^2$

k^{th} term $\hookrightarrow = 3^k T\left(\frac{n}{4^k}\right) + 3^{k-1}\left(\frac{n}{4^{k-1}}\right)^2 + 3^{k-2}\left(\frac{n}{4^{k-2}}\right)^2 + \cdots$

$\cdots + 3^2\left(\frac{n}{4^2}\right)^2 + 3\left(\frac{n}{4}\right)^2 + 3^0\left(\frac{n}{4^0}\right)^2$

$\frac{n}{4^k} = 1$

$n = 4^k$

$\log n = k \log 4$

$k = \log n / \log 4$

$k = \log_4 n$

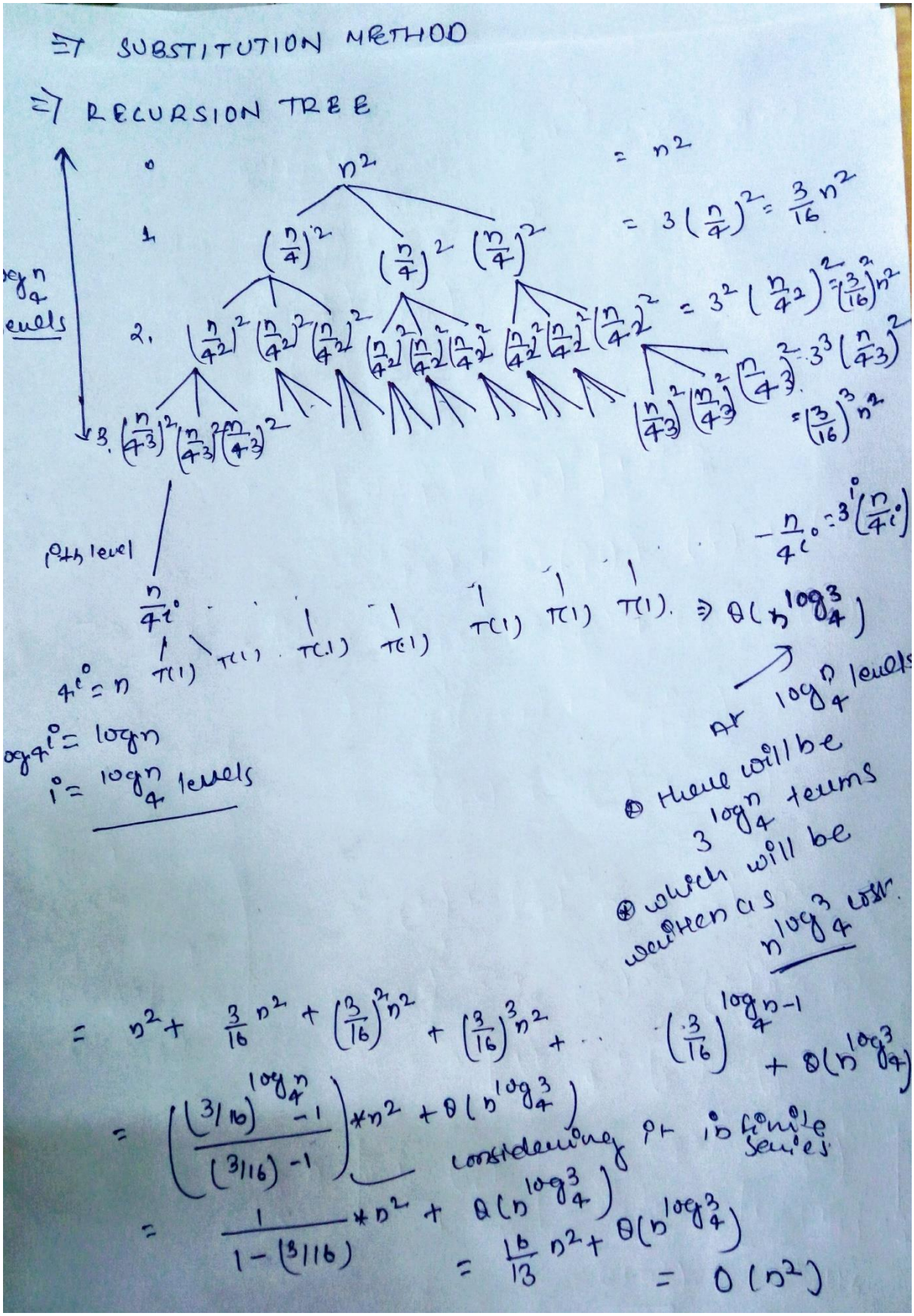
$= 3^k T(1) + \left(\frac{3}{4}\right)^{k-1} n^2 + \left(\frac{3}{4^2}\right)^{k-2} * n^2$

$+ \left(\frac{3}{4^2}\right)^{k-3} n^2 + \cdots \left(\frac{3}{4^2}\right)^2 n^2 + \left(\frac{3}{4^2}\right)^1 n^2 + n^2$

$= 3^{\log_4 n} + n^2\left[1 + \frac{3}{4^2} + \left(\frac{3}{4^2}\right)^2 + \left(\frac{3}{4^2}\right)^3 + \cdots \left(\frac{3}{4^2}\right)^{k-2} + \left(\frac{3}{4^2}\right)^{k-1}\right]$

$= n^{\log_4 3} + n^2\left[\frac{1 - (3/16)^k}{1 - 3/16}\right] = n^{\log_4 3} + n^2\left[\frac{1 - (3/16)^{\log_4 n}}{13/16}\right]$

$= O(n^2)$


⇒ SUBSTITUTION METHOD
⇒ RECURSION TREE
n^2
$= n^2$
$= 3\left(\frac{n}{4}\right)^2 = \frac{3}{16}n^2$
$= 3^2\left(\frac{n}{4^2}\right)^2 = \left(\frac{3}{16}\right)^2 n^2$
$= 3^3\left(\frac{n}{4^3}\right)^2 = \left(\frac{3}{16}\right)^3 n^2$
$\log_4 n$ levels
i^{th} level
$\frac{n}{4^i} = 3^i\left(\frac{n}{4^i}\right)$
$\frac{n}{4^i}$ T(1) T(1) T(1) T(1) T(1) T(1) $\Rightarrow \Theta(n^{\log_4 3})$
$4^i = n$
$\log_4 i = \log n$
$i = \log_4 n$ levels
At $\log_4 n$ levels
• There will be $3^{\log_4 n}$ terms
• which will be calculated as $n^{\log_4 3}$ (approx.)
$= n^2 + \frac{3}{16}n^2 + \left(\frac{3}{16}\right)^2 n^2 + \left(\frac{3}{16}\right)^3 n^2 + \cdots \left(\frac{3}{16}\right)^{\log_4 n - 1} + \Theta(n^{\log_4 3})$
$= \left(\frac{(3/16)^{\log_4 n} - 1}{(3/16) - 1}\right) * n^2 + \Theta(n^{\log_4 3})$ considering it is finite series
$= \frac{1}{1 - (3/16)} * n^2 + \Theta(n^{\log_4 3})$
$= \frac{16}{13}n^2 + \Theta(n^{\log_4 3})$
$= O(n^2)$

Que: - Use the master method to give tight asymptotic bounds for the following recurrences.

a. $T(n) = 2T(n/4) + 1.$

b. $T(n) = 2T(n/4) + \sqrt{n}.$

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

d. $T(n) = 2T(n/4) + n^2.$

a. n^log2 base(4) = n^log4^(1/2) base(4) = **theta(n^1/2)**
b. **theta(n^1/2 *logn)**
c. **theta(n)** as n>n^log2 (base4)
d. **theta(n^2)**

4-1 Recurrence examples

Give asymptotic upper and lower bounds for $T(n)$ in each of the following recurrences. Assume that $T(n)$ is constant for $n \leq 2$. Make your bounds as tight as possible, and justify your answers.

a. $T(n) = 2T(n/2) + n^4.$

b. $T(n) = T(7n/10) + n.$

c. $T(n) = 16T(n/4) + n^2.$

d. $T(n) = 7T(n/3) + n^2.$

e. $T(n) = 7T(n/2) + n^2.$

f. $T(n) = 2T(n/4) + \sqrt{n}.$

g. $T(n) = T(n-2) + n^2.$

Solution:-

a. n^loga base(b), hence **theta(n^4)**
b. n^log7 base(10) = n^0.85 < n hence **theta(n)**
c. **theta(n^2logn)**
d. n^log7 base(3) < n^2 hence, **theta(n^2)**
e. n^log7 (base2) > n^2 hence **theta(n^log7 (base2))**
f. n^log2 (base4) = **theta(n^1/2*logn)**
g. **theta(n^3)**

T(n) = T(n-4) + (n-2)^2 + n^2
T(n) = T(n-6) + (n-4)^2 + (n-2)^2 + n^2|
T(n) = T(n-8) + (n-6)^2 + (n-4)^2 + (n-2)^2 + n^2
.
.kth term
.
T(n) = T(n-2k) + (n-(2k-2))^2 + (n-(2k-4))^2 + ... (n-4)^2 +(n-2)^2 + n^2

Suppose 2k = n, k=n/2
=T(0) + 2^2 + 4^2 + 6^2 + 8^2 + ...(n-2)^2 + n^2
=1 + 2^2[1 + 2^2 + 3^2 + 4^2 + ... k^2] (there are n/2 total terms)
=1 + 4[k(k+1)(2k+1)/6] = k^3 or **theta(n^3)** as k=n/2

4-3 More recurrence examples

Give asymptotic upper and lower bounds for $T(n)$ in each of the following recurrences. Assume that $T(n)$ is constant for sufficiently small n. Make your bounds as tight as possible, and justify your answers.

a. $T(n) = 4T(n/3) + n \lg n$.

b. $T(n) = 3T(n/3) + n/\lg n$.

c. $T(n) = 4T(n/2) + n^2\sqrt{n}$.

d. $T(n) = 3T(n/3 - 2) + n/2$.

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

f. $T(n) = T(n/2) + T(n/4) + T(n/8) + n$.

g. $T(n) = T(n-1) + 1/n$.

h. $T(n) = T(n-1) + \lg n$.

i. $T(n) = T(n-2) + 1/\lg n$.

j. $T(n) = \sqrt{n}T(\sqrt{n}) + n$.

a. $4T(n/3) + n\log n$

$a=4 \quad b=3 \quad k=1 \quad p=1$

$4 > 3^1 \quad (a > b^k) = \theta(n^{\log_b^a})$ T.C: $\underline{\theta(n^{\log_3^4})}$

b. $T(n) = 3T(n/3) + \frac{n}{\log n}$

$T(n) = 3T(n/3) + n*(\log n)^{-1}$

$a=3 \quad b=3 \quad k=1 \quad p=-1$

$a = b^k \quad 3 = 3^1 =$

$p=-1 \quad = \theta(n^{\log_b^a} \log\log n)$

$= n^{\log_3^3} \log\log n$

$= \underline{\theta(n\log\log n)}$

c. $T(n) = 4T(n/2) + n^2\sqrt{n} \quad = 4T(n/2) + n^2 * n^{1/2}$

$= 4T(n/2) + n^{5/2}$

$a=4 \quad b=2$

$n^{\log_2^4} = n^2 < n^{2.5} = \underline{\theta(n^{2.5})}$

d. $T(n) = 3T(n/3-2) + \frac{n}{2}$

ignore constant-

$T(n) = 3T(n/3) + n$

$= \underline{\theta(n\log n)}$

e. $T(n) = 2T(n/2) + n/\log n$

$= 2T(n/2) + n(\log n)^{-1}$

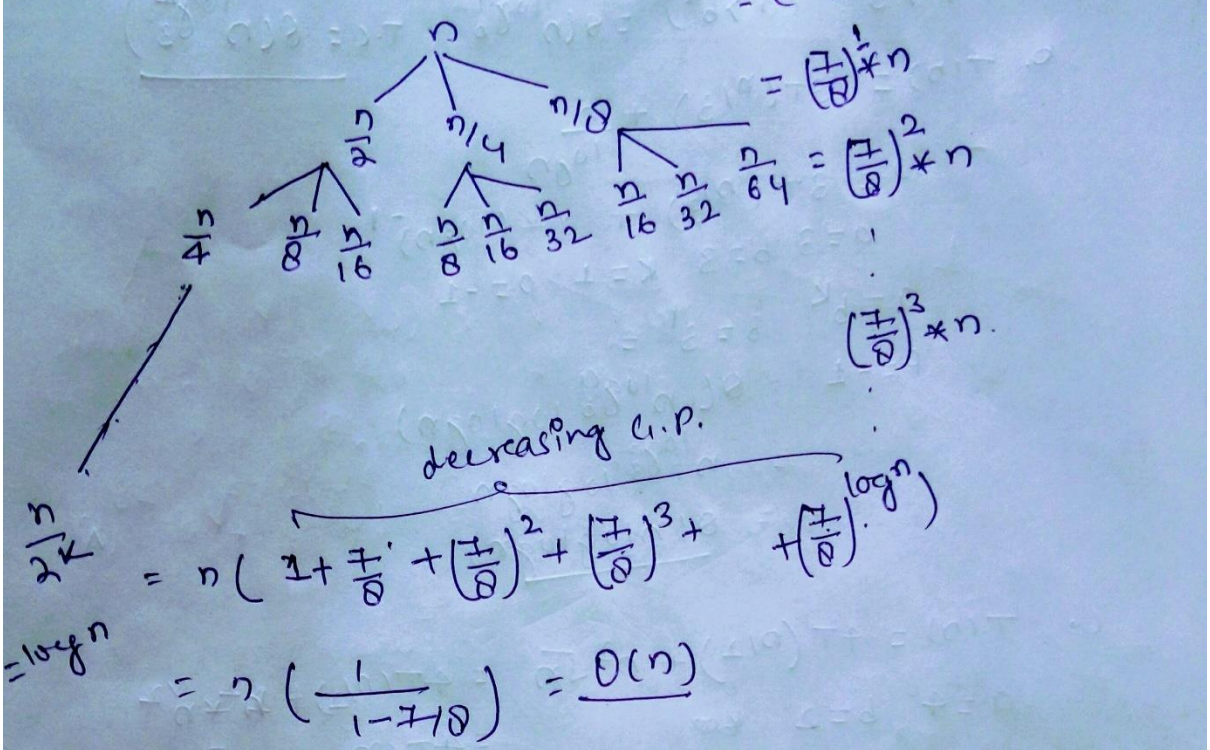
$a=1, b=2, k=1, p=-1$

$a = b^k$

$p=-1 \quad (n^{\log_b^a} \log\log n)$

$= \underline{\theta(n\log\log n)}$

f. $T(n) = T(n/2) + T(n/4) + T(n/8) + n$



$= (7/8)^0 * n$

$= (7/8)^1 * n$

$= (7/8)^2 * n$

$(7/8)^3 * n$

decreasing G.P.

$= n\left(1 + \frac{7}{8} + \left(\frac{7}{8}\right)^2 + \left(\frac{7}{8}\right)^3 + \cdots + \left(\frac{7}{8}\right)^{\log n}\right)$

$= n\left(\frac{1}{1 - 7/8}\right) = \underline{O(n)}$

g. $T(n) = T(n-1) + \frac{1}{n}$

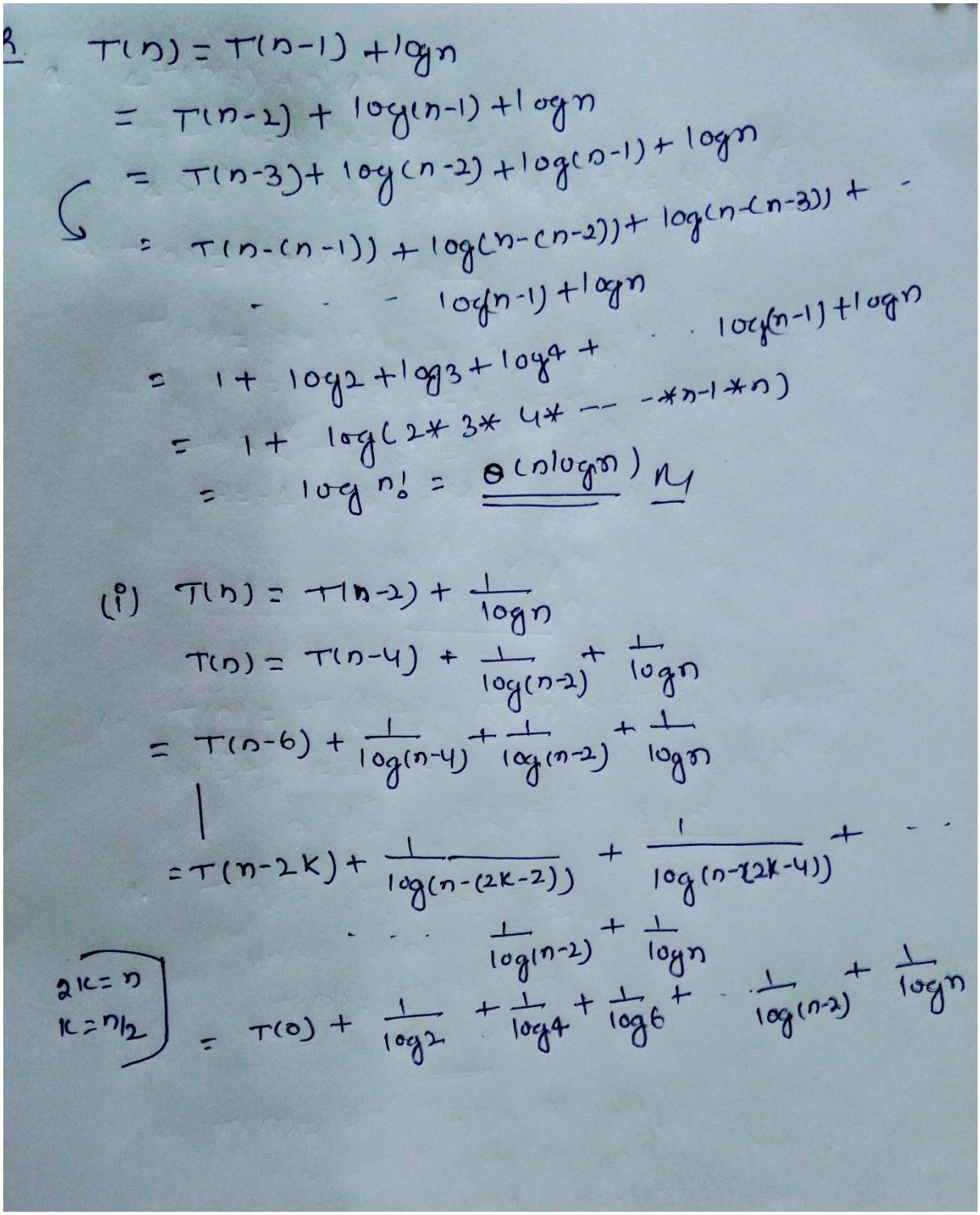
$= T(n-2) + \frac{1}{n-1} + \frac{1}{n}$

$= T(n-3) + \frac{1}{(n-2)} + \frac{1}{n-1} + \frac{1}{n}$

$= T(n-(n-1)) + \frac{1}{(n-(n-2))} + \frac{1}{(n-(n-3))} + \cdots \frac{1}{n-1} + \frac{1}{n}$

$= T(1) + \underbrace{\frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n-1} + \frac{1}{n}}_{\log n \text{ series}}$

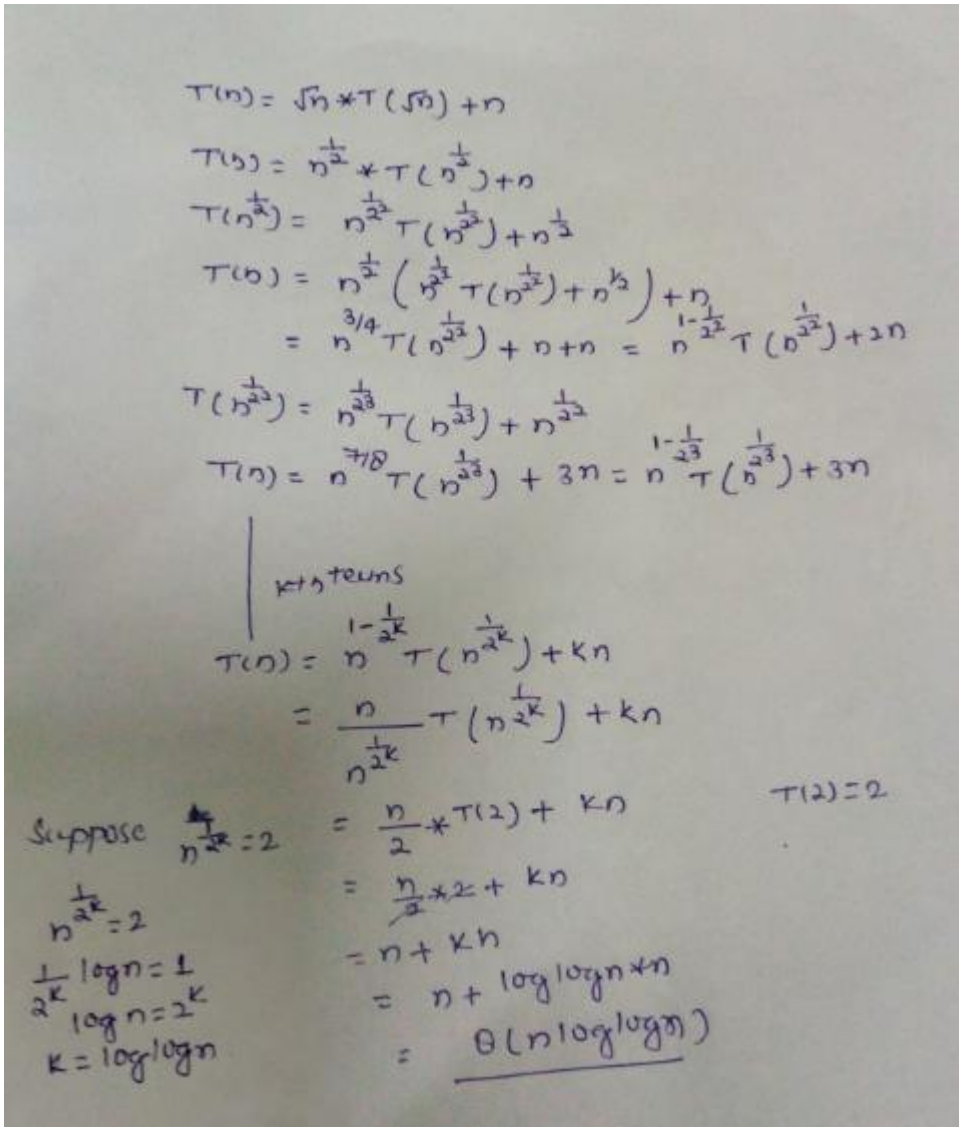
$= \underline{\text{T.C. } O(\log n)}$ Ans



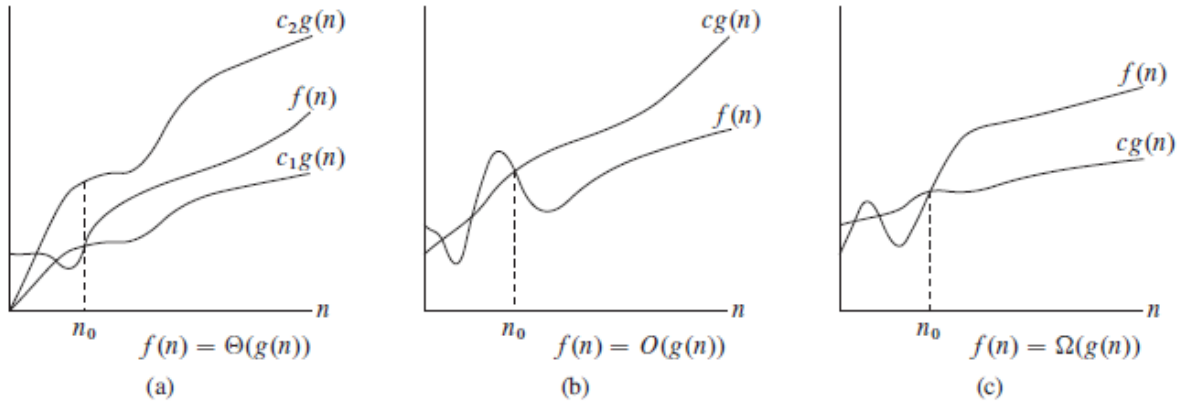
1/log6 = 1/log2*3 = 1/(log2 + log3) = log2 can be ignored
T(n) = 1/log2 + 1/log3 + 1/log4 + 1/log5 + ...1/log(n/2)

suppose we have been given a series
S=3+4+7+9+16+17.....100. (Assume there are 75 terms in this series)
In worst case if we can't find any pattern, we can at least say that
3*75 <= S <= 100*75

Hence, n*(1/logn) <= T(n) <= n*(1/log2) **So TC will be theta(n/logn) or O(n)**


T(n) = √n * T(√n) + n
T(n) = n^{1/2} * T(n^{1/2}) + n
T(n^{1/2}) = n^{1/2^2} T(n^{1/2^2}) + n^{1/2}
T(n) = n^{1/2} (n^{1/2^2} T(n^{1/2^2}) + n^{1/2}) + n
= n^{3/4} T(n^{1/2^2}) + n + n = n^{1-1/2^2} T(n^{1/2^2}) + 2n
T(n^{1/2^2}) = n^{1/2^3} T(n^{1/2^3}) + n^{1/2^2}
T(n) = n^{7/8} T(n^{1/2^3}) + 3n = n^{1-1/2^3} T(n^{1/2^3}) + 3n
k times
T(n) = n^{1-1/2^k} T(n^{1/2^k}) + kn
= n / n^{1/2^k} T(n^{1/2^k}) + kn
Suppose n^{1/2^k} = 2
= n/2 * T(2) + kn
T(2) = 2
= n/2 * 2 + kn
= n + kn
= n + log log n * n
= θ(n log log n)
n^{1/2^k} = 2
1/2^k log n = 1
log n = 2^k
k = log log n

Growth of Functions



O-notation (Big – oh notation)

When we have only an asymptotic upper bound, we use O-notation. For a given function g(n), we denote by O(g(n)) the set of functions
O(g(n)) = { f(n): there contains positive c and n0 such that

$$0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\} .$$

We use O-notation to give an upper bound on a function, to within a constant factor.

Ω-notation

Just as O-notation provides an asymptotic *upper* bound on a function, Ω-notation provides an ***asymptotic lower bound***. For a given function $g(n)$, we denote by $\Omega(g(n))$ (pronounced "big-omega of g of n" or sometimes just "omega of g of n") the set of functions

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

Θ-notation

In Chapter 2, we found that the worst-case running time of insertion sort is $T(n) = \Theta(n^2)$. Let us define what this notation means. For a given function $g(n)$, we denote by $\Theta(g(n))$ the *set of functions*

$$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1g(n) \le f(n) \le c_2g(n) \text{ for all } n \ge n_0\} .^{1}$$

Example:-

nition to show that $\frac{1}{2}n^2 - 3n = \Theta(n^2)$. To do so, we must determine positive constants c_1, c_2, and n_0 such that

$$c_1 n^2 \le \frac{1}{2}n^2 - 3n \le c_2 n^2$$

for all $n \ge n_0$. Dividing by n^2 yields

$$c_1 \le \frac{1}{2} - \frac{3}{n} \le c_2 .$$

We can make the right-hand inequality hold for any value of $n \ge 1$ by choosing any constant $c_2 \ge 1/2$. Likewise, we can make the left-hand inequality hold for any value of $n \ge 7$ by choosing any constant $c_1 \le 1/14$. Thus, by choosing $c_1 = 1/14$, $c_2 = 1/2$, and $n_0 = 7$, we can verify that $\frac{1}{2}n^2 - 3n = \Theta(n^2)$. Certainly, other choices for the constants exist, but the important thing is that *some* choice exists. Note that these constants depend on the function $\frac{1}{2}n^2 - 3n$; a different function belonging to $\Theta(n^2)$ would usually require different constants.

o-notation

The asymptotic upper bound provided by O-notation may or may not be asymptotically tight. The bound $2n^2 = O(n^2)$ is asymptotically tight, but the bound $2n = O(n^2)$ is not. We use o-notation to denote an upper bound that is not asymptotically tight. We formally define $o(g(n))$ ("little-oh of g of n") as the set

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

For example, $2n = o(n^2)$, but $2n^2 \ne o(n^2)$.

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

(small – oh condition)

ω-notation

By analogy, ω-notation is to Ω-notation as o-notation is to O-notation. We use ω-notation to denote a lower bound that is not asymptotically tight. One way to define it is by

$f(n) \in \omega(g(n))$ if and only if $g(n) \in o(f(n))$.

Formally, however, we define $\omega(g(n))$ ("little-omega of g of n") as the set

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

For example, $n^2/2 = \omega(n)$, but $n^2/2 \ne \omega(n^2)$. The relation $f(n) = \omega(g(n))$ implies that

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

(Small – omega)

Comparing functions:-

1. **Transitivity**

$$\begin{aligned}
f(n) &= \Theta(g(n)) \text{ and } g(n) = \Theta(h(n)) \quad \text{imply} \quad f(n) = \Theta(h(n)), \\
f(n) &= O(g(n)) \text{ and } g(n) = O(h(n)) \quad \text{imply} \quad f(n) = O(h(n)), \\
f(n) &= \Omega(g(n)) \text{ and } g(n) = \Omega(h(n)) \quad \text{imply} \quad f(n) = \Omega(h(n)), \\
f(n) &= o(g(n)) \text{ and } g(n) = o(h(n)) \quad \text{imply} \quad f(n) = o(h(n)), \\
f(n) &= \omega(g(n)) \text{ and } g(n) = \omega(h(n)) \quad \text{imply} \quad f(n) = \omega(h(n)).
\end{aligned}$$

2. **Reflexivity**

$$\begin{aligned}
f(n) &= \Theta(f(n)), \\
f(n) &= O(f(n)), \\
f(n) &= \Omega(f(n)).
\end{aligned}$$

3. **Symmetry**

$$f(n) = \Theta(g(n)) \text{ if and only if } g(n) = \Theta(f(n)).$$

4. **Transpose symmetry:**

$$\begin{aligned}
f(n) &= O(g(n)) \text{ if and only if } g(n) = \Omega(f(n)) \\
f(n) &= o(g(n)) \text{ if and only if } g(n) = \omega(f(n))
\end{aligned}$$

Because these properties hold for asymptotic notations, we can draw an analogy between the asymptotic comparison of two functions f and g and the comparison of two real numbers a and b:

$$\begin{aligned}
f(n) = O(g(n)) &\quad \text{is like} \quad a \le b \\
f(n) = \Omega(g(n)) &\quad \text{is like} \quad a \ge b \\
f(n) = \Theta(g(n)) &\quad \text{is like} \quad a = b \\
f(n) = o(g(n)) &\quad \text{is like} \quad a < b \\
f(n) = \omega(g(n)) &\quad \text{is like} \quad a > b
\end{aligned}$$

Note:-

1. We say that $f(n)$ is ***asymptotically smaller*** than $g(n)$ if $f(n) = o(g(n))$, and $f(n)$ is ***asymptotically larger*** than $g(n)$ if $f(n) = \omega(g(n))$.
2. **Not all functions are asymptotically comparable**, That is, for two functions f(n) and g(n), it may be the case that neither f(n)=O(g(n)) nor f(n)=omega(g(n)) holds.

Example: we can compare the function n and n^{1+sin n} since the value of sin n will oscillate b/w 0 and 1, and exponent of n^{1+sin n} will oscillate b/w 0 and 2.

Properties:-

1. Let f(n) and g(n) be asymptotically nonnegative functions, then
   $\max(f(n), g(n)) = \Theta(f(n) + g(n))$
2. For any real constants a and b, where b > 0
   $(n + a)^b = \Theta(n^b)$

3. Following property holds

   Prove that $o(g(n)) \cap \omega(g(n))$ is the empty set.

Example: Is 2^(n+1) = O(2^n)? Is 2^2n = O(2^n)?

$2^{n+1} \geq 2 \cdot 2^n$ for all $n \geq 0$, so $2^{n+1} = O(2^n)$. However, 2^{2n} is not $O(2^n)$. If it were, there would exist n_0 and c such that $n \geq n_0$ implies $2^n \cdot 2^n = 2^{2n} \leq c2^n$, so $2^n \leq c$ for $n \geq n_0$ which is clearly impossible since c is a constant.

Monotonicity:-

A function $f(n)$ is ***monotonically increasing*** if $m \leq n$ implies $f(m) \leq f(n)$. Similarly, it is ***monotonically decreasing*** if $m \leq n$ implies $f(m) \geq f(n)$. A function $f(n)$ is ***strictly increasing*** if $m < n$ implies $f(m) < f(n)$ and ***strictly decreasing*** if $m < n$ implies $f(m) > f(n)$.

Exponentials

1. For all real a > 0, m, and n, we have the following identities:

$$\begin{aligned} a^0 &= 1 , \\ a^1 &= a , \\ a^{-1} &= 1/a , \\ (a^m)^n &= a^{mn} , \\ (a^m)^n &= (a^n)^m , \\ a^m a^n &= a^{m+n} . \end{aligned}$$

2. $$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots = \sum_{i=0}^{\infty} \frac{x^i}{i!} ,$$

Logarithms:-

1. We shall use the following notations:

$$\begin{aligned} \lg n &= \log_2 n \quad \text{(binary logarithm)} , \\ \ln n &= \log_e n \quad \text{(natural logarithm)} , \\ \lg^k n &= (\lg n)^k \quad \text{(exponentiation)} , \\ \lg \lg n &= \lg(\lg n) \quad \text{(composition)} . \end{aligned}$$

2. For all real a > 0, b > 0, c > 0, and n,

$$
\begin{aligned}
a &= b^{\log_b a}, \\
\log_c(ab) &= \log_c a + \log_c b, \\
\log_b a^n &= n \log_b a, \\
\log_b a &= \frac{\log_c a}{\log_c b}, \\
\log_b(1/a) &= -\log_b a, \\
\log_b a &= \frac{1}{\log_a b}, \\
a^{\log_b c} &= c^{\log_b a},
\end{aligned}
$$

Note: - Where, in each equation above, logarithm bases are not 1.

3. There is a simple series expansion for **log(1 + x) when |x| < 1**:

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} - \cdots$$

4. Following properties hold true

$$
\begin{aligned}
n! &= o(n^n), \\
n! &= \omega(2^n), \\
\lg(n!) &= \Theta(n \lg n)
\end{aligned}
$$

The iterated logarithm function

We define the iterated logarithm function as:

$$\lg^* n = \min\{i \geq 0 : \lg^{(i)} n \leq 1\}$$

The iterated logarithm is a very slowly growing function:

$$
\begin{aligned}
\lg^* 2 &= 1 \\
\lg^* 4 &= 2 \\
\lg^* 16 &= 3 \\
\lg^* 65536 &= 4 \\
\lg^*(2^{65536}) &= 5
\end{aligned}
$$

Note:-

If f(n) and g(n) are **monotonically increasing functions**, then so are the functions f(n) + g(n) and f(g(n)), and if f(n) and g(n) are in addition nonnegative, then f(n)*g(n) is monotonically increasing.

Example:-

Which is asymptotically larger: $\lg(\lg^* n)$ or $\lg^*(\lg n)$?

Solution:
Suppose n = 2^128
log*(2^65536) = 5
log(log*n) = log(5) = 2 (approx)

log*(log2^65536) = log*(2^16)= 4

Therefore, log*(logn) grows more quickly.

Relative asymptotic growths

Que Coreman 3.2

Indicate, for each pair of expressions (A, B) in the table below, whether A is O, o, Ω, ω, or Θ of B. Assume that $k \geq 1$, $\epsilon > 0$, and $c > 1$ are constants. Your answer

Solution:-

A	B	O	o	Ω	ω	Θ
$\lg^k n$	n^ϵ	yes	yes	no	no	no
n^k	c^n	yes	yes	no	no	no
$\sqrt{n}$	$n^{\sin n}$	no	no	no	no	no
2^n	$2^{n/2}$	no	no	yes	yes	no
$n^{\log c}$	$c^{\log n}$	yes	no	yes	no	yes
$\log(n!)$	$\log(n^n)$	yes	no	yes	no	yes

a.

A= $\log^k n$, k>=1

B= n^p, p>0

log A= k*log(logn)

log B= p*logn

now,

log B > log A, for large values of n and also log is an increasing function, so we can say that

B > A, for large values of n, hence A=O(B)

b. n^k is a polynomial function, while c^n is an exponential function
hence n^k = o(c^n) or O(c^n)
or take log both sides
k*logn = n*logc k and c are constant factors
logn = o(n) (small-oh)

c. 2^n = 2^(n/2)
2^n > Root(2^n)

d. n^logc and c^logn
both equal functions as n^logc can be written as c^logn which equal to RHS
hence A = O(B), A=Omega(B) hence A=Theeta(B)

e. It's a known property log(n!) = Theeta(nlogn)

f. 1 = o(1/n) (small-oh)

Que Coreman 3.3

Write the following functions in increasing order. And mention those functions which are asymptotically equal to each other.

$\lg(\lg^* n)$	$2^{\lg^* n}$	$(\sqrt{2})^{\lg n}$	n^2	$n!$	$(\lg n)!$
$(\frac{3}{2})^n$	n^3	$\lg^2 n$	$\lg(n!)$	2^{2^n}	$n^{1/\lg n}$
$\ln\ln n$	$\lg^* n$	$n \cdot 2^n$	$n^{\lg\lg n}$	$\ln n$	1
$2^{\lg n}$	$(\lg n)^{\lg n}$	e^n	$4^{\lg n}$	$(n+1)!$	$\sqrt{\lg n}$
$\lg^*(\lg n)$	$2^{\sqrt{2\lg n}}$	n	2^n	$n \lg n$	$2^{2^{n+1}}$

Solution-

$$2^{2^{n+1}} > 2^{2^n} > (n+1)! > n! > e^n >$$

$$n \cdot 2^n > 2^n > (\tfrac{3}{2})^n > \frac{n^{\lg\lg n}}{(\lg n)^{\lg n}} > (\lg n)! >$$

$$n^3 > \frac{n^2}{4^{\lg n}} > \frac{n \lg n}{\lg(n!)} > \frac{n}{2^{\lg n}} > (\sqrt{2})^{\lg n} >$$

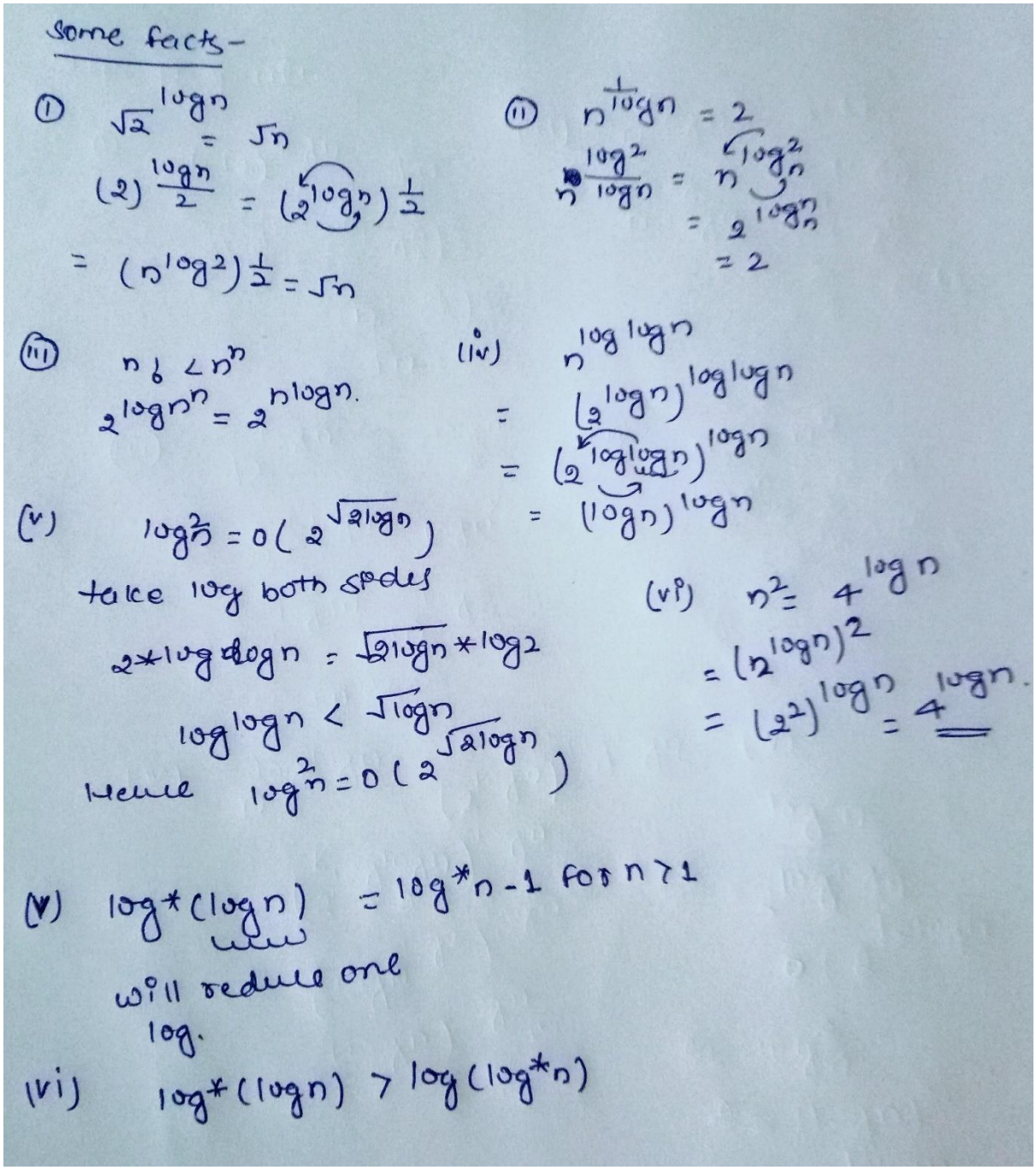
$$2^{\sqrt{2\lg n}} > \lg^2 n > \ln n > \sqrt{\lg n} > \ln\ln n >$$

$$2^{\lg^* n} > \frac{\lg^*(\lg n)}{\lg^* n} > \lg(\lg^* n) > \frac{1}{n^{1/\lg n}}$$

(b) **[2 points]** Do problem 3-3(b) on page 58 of CLRS.
Give an example of a single nonnegative function $f(n)$ such that for all functions $g_i(n)$ in part (a), $f(n)$ is neither $O(g_i(n))$ nor $\Omega(g_i(n))$.
Answer: $f(n) = (1 + \sin n) \cdot 2^{2n+2}$.

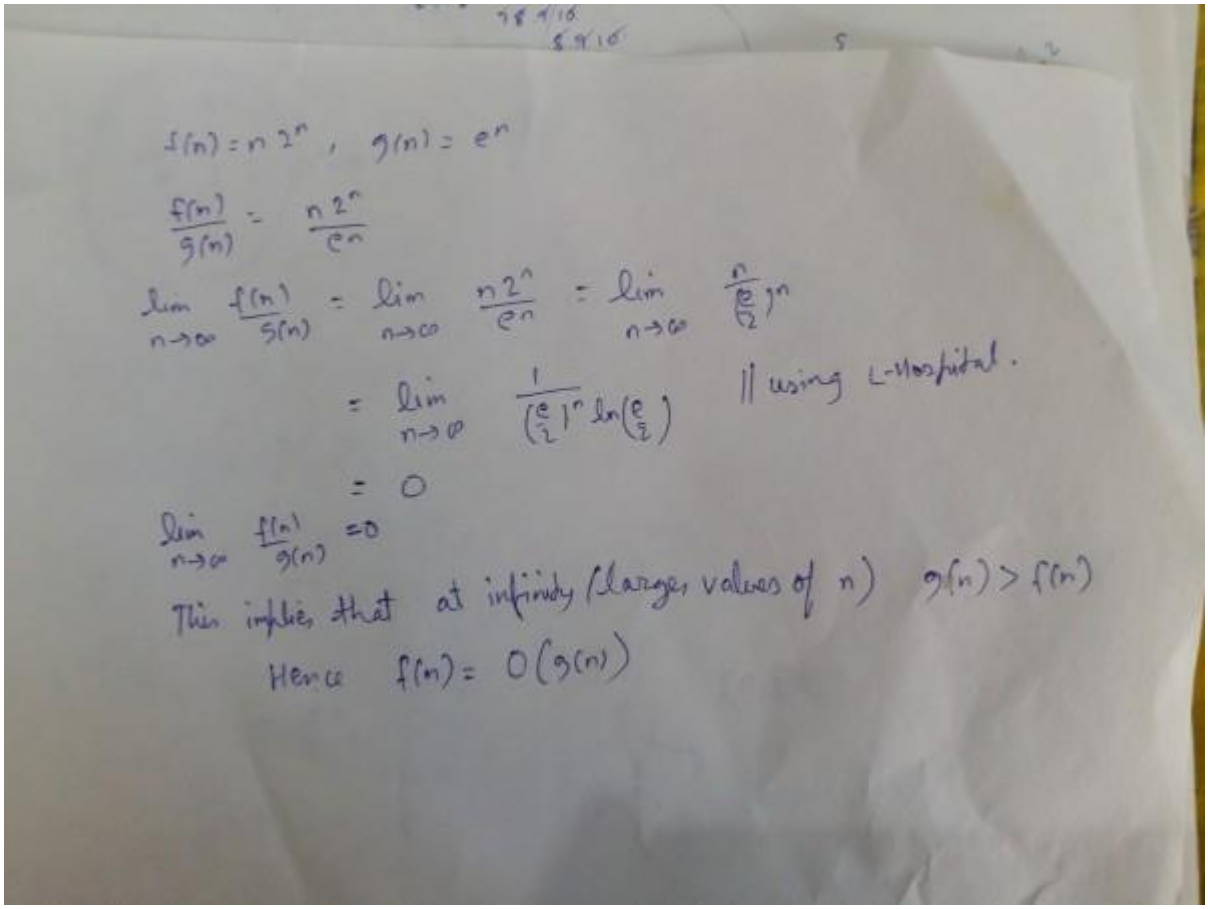
(c) **[2 points]** Give an example of a single nonnegative function $f(n)$ such that for all functions $g_i(n)$ in part (a), $f(n) = o(g_i(n))$.
Answer: $f(n) = 1/n$.

(d) **[2 points]** Give an example of a single nonnegative function $f(n)$ such that for all functions $g_i(n)$ in part (a), $f(n) = \omega(g_i(n))$.
Answer: $f(n) = 2^{2^{2^n}}$.



Important Properties:-

1. $f(n) = O(g(n))$ iff $\text{limit}_{n \to \infty} \frac{f(n)}{g(n)} \leq C$
2. $f(n) = \Omega(g(n))$ iff $\text{limit}_{n \to \infty} \frac{f(n)}{g(n)} \geq C$
3. $f(n) = \Theta \geq (g(n))$ iff $\text{limit}_{n \to \infty} \frac{f(n)}{g(n)} = C$
4. $f(n) = o(g(n))$ iff $\text{limit}_{n \to \infty} \frac{f(n)}{g(n)} = o$
5. $f(n) = \omega(g(n))$ iff $\text{limit}_{n \to \infty} \frac{f(n)}{g(n)} = \infty$



e^n > n*2^n

Ques 3-4 Coreman:

Let f(n) and g(n) be asymptotically positive functions. Prove or disprove each of the following conjecture.

a. $f(n) = O(g(n))$ implies $g(n) = O(f(n))$.

b. $f(n) + g(n) = \Theta(\min(f(n), g(n)))$.

c. $f(n) = O(g(n))$ implies $\lg(f(n)) = O(\lg(g(n)))$, where $\lg(g(n)) \geq 1$ and $f(n) \geq 1$ for all sufficiently large n.

d. $f(n) = O(g(n))$ implies $2^{f(n)} = O\left(2^{g(n)}\right)$.

e. $f(n) = O\left((f(n))^2\right)$.

f. $f(n) = O(g(n))$ implies $g(n) = \Omega(f(n))$.

g. $f(n) = \Theta(f(n/2))$.

h. $f(n) + o(f(n)) = \Theta(f(n))$.

Solution:

a. **False**. Counterexample: n = O(n^2) but n^2 != O(n).
b. **False**. Counterexample: n + n^2 != O(n).

c. **True**

True. Since $f(n) = O(g(n))$ there exist c and n_0 sucht hat $n \geq n_0$ implies $f(n) \leq cg(n)$ and $f(n) \geq 1$. This means that $\log(f(n)) \leq \log(cg(n)) = \log(c) + \log(g(n))$. Note that the inequality is preserved after taking logs because $f(n) \geq 1$. Now we need to find d such that $f(n) \leq d\log(g(n))$. It will suffice to make $\log(c) + \log(g(n)) \leq d\log(g(n))$, which is achieved by taking $d = \log(c) + 1$, since $\log(g(n)) \geq 1$.

d. False. Counterexample: $2n = O(n)$ but $2^{2n} \neq 2^n$

e. **False**, counter example is f(n) = 1/n as 1/n != O(1/n^2)

f. **True**

True. Since $f(n) = O(g(n))$ there exist c and n_0 such that $n \geq n_0$ implies $f(n) \leq cg(n)$. Thus $g(n) \geq \frac{1}{c}f(n)$, so $g(n) = \Omega(f(n))$.

g. **False.** Let f(n) = 2^2n then f(n/2) = 2^n and 2^2n != O(2^n)

h. **True**.

True. Let g be any function such that $g(n) = o(f(n))$. Since g is asymptotically positive let n_0 be such that $n \geq n_0$ implies $g(n) \geq 0$. Then $f(n) + g(n) \geq f(n)$ so $f(n) + o(f(n)) = \Omega(f(n))$. Next, choose n_1 such that $n \geq n_1$ implies $g(n) \leq f(n)$. Then $f(n) + g(n) \leq f(n) + f(n) = 2f(n)$ so $f(n) + o(f(n)) = O(f(n))$. By Theorem 3.1, this implies $f(n) + o(f(n)) = \Theta(f(n))$.

Que (GO)

Which of these functions grows fastest with n?

A. e^n/n.
B. $e^{n-0.9\log n}$.
C. 2^n.
D. $(\log n)^{n-1}$.
E. None of the above.

Assuming that the base of the log in the question is e.

Let us try to rewrite each of these functions in the form $e^{\text{something}}$, to make the comparison easier.

a.	e^n/n	$= \frac{e^n}{e^{(\ln n)}}$	$= e^{(n - \ln n)}$
b.	$e^{n-0.9\ln n}$		$= e^{(n - 0.9\ln n)}$
c.	2^n	$= \left(e^{\ln 2}\right)^n$	$= e^{(n \ln 2)}$
d.	$(\ln n)^{n-1}$	$= \left(e^{\ln\ln n}\right)^{n-1}$	$= e^{(n\ln\ln n - \ln\ln n)}$

Now, if we just compare the exponents of all, we can clearly see that $(n\ln\ln n - \ln\ln n)$ grows faster than the rest. Note that in option c. the multiplicative $\ln 2$ is a constant, and hence grows slower than the multiplicative $\ln\ln n$ from option d.

This implies that $e^{(n\ln\ln n - \ln\ln n)}$ grows the fastest, and hence, $(\ln n)^{n-1}$ grows the fastest.

Thus, option d. is the correct answer.

Note:- C is not growing because e^(n*constant).

Sort a nearly sorted Array (K-Sorted Array)

We have an unsorted array in which any element is misplaced **no more than k position** from its correct position.

1. **Insertion Sort** the inner while loop won't run more than k times hence T.C will be **O(n*k)**

2. **Using Heap Data Structure**
   a. Take k+1 elements from unsorted array and create **min-heap** in **O(k)** time. We are sure that this min heap of (k+1) elements will have **smallest** element.
   b. Remove root element from min-heap **O(logk)** time and keep it in some temporary array called as **resulted array.**
   c. Next, insert another from unsorted array which is left with **(n-k-1)** elements into min-heap in O(logk) time.
   d. Now, min-heap will have second smallest element at the root, remove it, put it into resulted array.
   e. Continue until there is no element left in unsorted array

Time Complexity O(k) + O(n-k)*logk = O(nlogk)

Part II Sorting and Order Statistics

Algorithm	Worst-case running time	Average-case/expected running time
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Heapsort	$O(n \lg n)$	—
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)$ (expected)
Counting sort	$\Theta(k + n)$	$\Theta(k + n)$
Radix sort	$\Theta(d(n + k))$	$\Theta(d(n + k))$
Bucket sort	$\Theta(n^2)$	$\Theta(n)$ (average-case)

Merge Sort

The merge sort algorithm closely follows the **divide-and-conquer** paradigm. Intuitively, it operates as follows.
Divide: Divide the n-element sequence to be sorted into two subsequences of n/2 elements each.
Conquer: Sort the two subsequences recursively using merge sort.
Combine: Merge the two sorted subsequences to produce the sorted answer.

The recursion "**bottoms out**" **when the sequence to be sorted has length 1**, since every sequence of length 1 is already in sorted order.
The key operation of the merge sort algorithm is the merging of two sorted sequences in the "**combine**" step. We merge by calling an auxiliary procedure **MERGE (A, p, q, r)**, where A is an array and p, q, and r are indices into the array such that p <=q < r. The procedure assumes that the sub-arrays **A[p…q]** and **A[q+1….r]** are in sorted order. It merges them to form a single sorted subarray that replaces the current subarray **A[p….r]**.

Note: - MERGE procedure takes time **Θ(n)**, where **n= r-p + 1** is the total number of elements being merged.

Merge procedure consists steps of choosing the smaller of the two top elements of two sorted list. Removing it from its list (which exposes a new top card), and placing this element into the output list.
We repeat this step until one input sorted list is empty, and we insert remaining elements into output list. Each basic step takes constant time.

The following pseudocode implements the merge procedure but with additional logic that avoids having to check whether either list is empty in each basic step. We place on the bottom of each list a **sentinel** element. We use the **infinite** as the sentential value. Since we know in advance that exactly **r - p + 1** cards will be placed onto the output pile, we can stop once we have performed that many basic steps.

1. Line computes the length n1 of the sub array A[p……q]
2. Line 2 computes the length n2 of the subarray A[q+1 ……..r]
3. We create two array L and R (Left & Right) of lengths n1+1 and n2+1 respectively. Extra position in each array will hold **sentential** element.
4. The for loops of line 4&5 copies the sub array A[p…..q] into L[1 …… n1]
5. The for loops of line 6&7 copies the sub array A[q+1…..r] into R[1 …… n2]
6. Line 8 and 9 put the sentinels at the ends of the arrays L and R.

```
MERGE(A, p, q, r)
 1  n1 = q − p + 1
 2  n2 = r − q
 3  let L[1..n1 + 1] and R[1..n2 + 1] be new arrays
 4  for i = 1 to n1
 5      L[i] = A[p + i − 1]
 6  for j = 1 to n2
 7      R[j] = A[q + j]
 8  L[n1 + 1] = ∞
 9  R[n2 + 1] = ∞
10  i = 1
11  j = 1
12  for k = p to r
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
```

From lines 12 to 17, a loop runs from p to r, it selects minimum of two elements on the top of list and insert into **original list A**

We can now use the **MERGE** procedure as a subroutine in the **merge sort** algorithm.

1. The procedure **MERGE-SORT(A, p, r)** sorts the elements in the subarray A[p…..r].
2. If p>=r, the sub array has at most one element and is therefore already sorted.
3. Otherwise, the divide step simply computes an index q that partitions A[p….r] into two subarrays: A[p….q], containing **ceil[n/2]** elements and A[q+1 ….r] containing **floor[n/2]** elements.

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

Note: - To sort the entire sequence A = <A[1], A[2], ……., A[n]>, we make the initial call **MERGE-SORT(A, 1, A.length)**, where once again A.length=n.

The algorithm consists of **merging pairs** of 1-item sequences to form sorted sequences of length 2, merging pairs of sequences of length 2 to form sorted sequences of length 4, and so on, until two sequences of length n/2 are merged to form the final sorted sequence of length n.

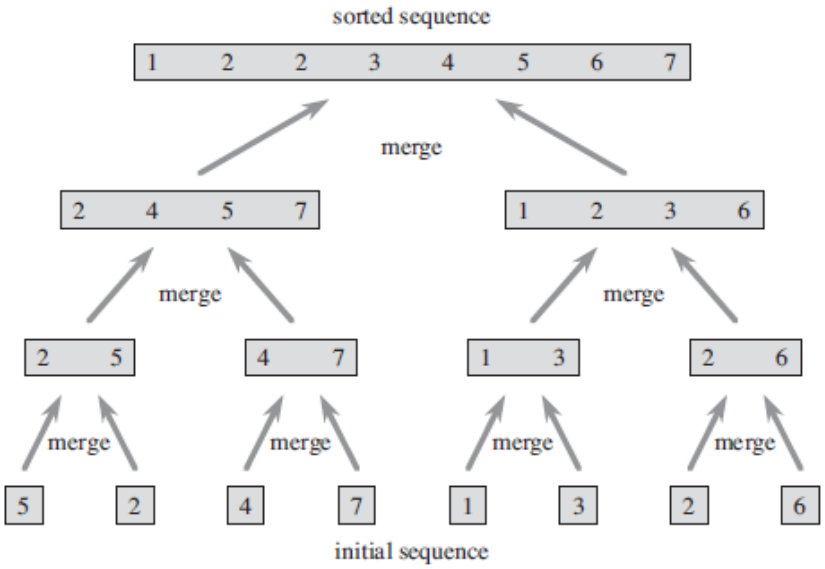
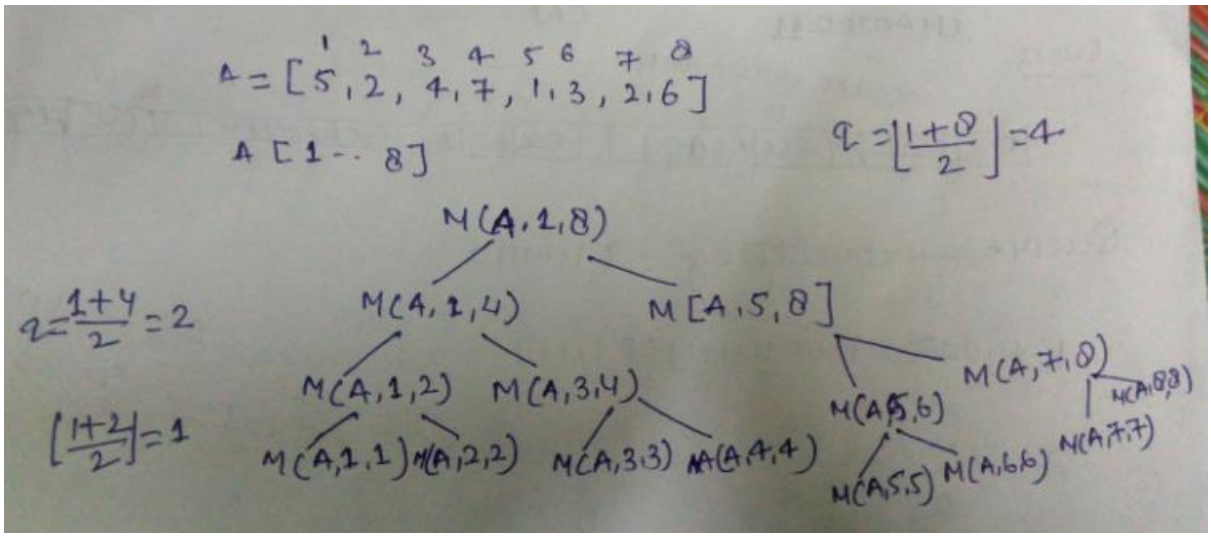


Figure 2.4 The operation of merge sort on the array $A = \langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$. The lengths of the sorted sequences being merged increase as the algorithm progresses from bottom to top.



Merge Procedure

Divide and Conquer recurrence relation Analysis:

Let T(n) be the running time on a problem of size n.

1. If the problem size is small enough, say **n <= c** for some constant c, the straightforward solution takes constant time, which we write as **Θ(1)**.
2. Suppose that division of the problem yields **a subproblems**, each of which is 1/b the size of the original.
3. It takes time **T(n/b)** to solve one subproblem of size **n/b**, and so it takes time **aT(n/b)** to solve **a of them**.
4. If we take **D(n)** time to divide the problem into subproblems and **C(n)** time to combine the solutions to the subproblems into the solution to the original problem. We get the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le c\,, \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

Analysis of merge sort

The worst-case running time of merge sort on n numbers. Merge sort on just one element takes constant time. When we have n > 1 elements, we break down the running time as follows.

Divide: The divide step just computes the middle of the subarray, which takes constant time. Thus, **D(n) = Θ(1)**.
Conquer: We recursively solve two subproblems, each of size n/2, which contributes 2T(n/2) to the running time.
Combine: We have already noted that the MERGE procedure on an n-element subarray takes time Θ(n), and so **C(n) = Θ(n)**

When we add the functions D(n) and C(n) for the merge sort analysis, we are adding a function that is Θ(n) and a function that is Θ(1). This sum is a linear function of n, **that is Θ(n)**.

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

T.C is Θ(nlogn)

Merge Sort Properties:-

1. Merge sort is **stable** and **not in-place**, **comparison-based** algorithm.
2. It's use for sorting **linked-list** in **O(nlogn)**.
3. It's useful in **external Sorting** such as database table don't fit in memory, need to sort on disk.
4. Useful in **Inversion Count** Problem.
5. sort requires **Ω(n)** auxiliary space.

Binary Search:-

The following recursive algorithm gives the desired result when called with a = 1 and b = n.

```
1: BinSearch(a,b,v)
2: if then a > b
3:     return NIL
4: end if
5: m = ⌊(a+b)/2⌋
6: if then m = v
7:     return m
8: end if
9: if then m < v
10:    return BinSearch(a,m,v)
11: end if
12: return BinSearch(m+1,b,v)
```

m is the middle element, v is the search key. If v is equal to middle element, return it. Else if v < m then apply BS from a to m else v>m then apply BS from m+1 to b.

$$\text{recurrence } T(n) = T(n/2) + c. \text{ So, } T(n) \in \Theta(\lg(n))$$

2.3-7 ★

Describe a $\Theta(n \lg n)$-time algorithm that, given a set S of n integers and another integer x, determines whether or not there exist two elements in S whose sum is exactly x.

```
1: Use Merge Sort to sort the array A in time Θ(n lg(n))
2: i = 1
3: j = n
4: while i < j do
5:     if A[i] + A[j] = S then
6:         return true
7:     end if
8:     if A[i] + A[j] < S then
9:         i = i + 1
10:    end if
11:    if A[i] + A[j] > S then
12:        j = j − 1
13:    end if
14: end while
15: return false
```

We can see that the while loop gets run at most O(n) times, as the quantity j – i starts at n -1 and decreases at each step.

Insertion sort on small arrays in merge sort

Although merge sort runs in **Θ(nlogn)** worst-case time and insertion sort runs in **Θ(n^2)** worst-case time, the constant factors in insertion sort can make it faster in practice for small problem sizes on many machines.
In merge sort, we can user insertion sort when subproblems become sufficiently small. Consider a modification to merge sort in which **n/k subsists** of length k are sorted using insertion sort and then merged using the standard merging mechanism, where k is a value to be determined.
Questions:

a. Show that insertion sort can sort the n/k sublists, each of length k, in $\Theta(nk)$ worst-case time.

b. Show how to merge the sublists in $\Theta(n \lg(n/k))$ worst-case time.

c. Given that the modified algorithm runs in $\Theta(nk + n \lg(n/k))$ worst-case time, what is the largest value of k as a function of n for which the modified algorithm has the same running time as standard merge sort, in terms of Θ-notation?

d. How should we choose k in practice?

Solutions:

a. The time for insertion sort to sort a single list of length k is $\Theta(k^2)$, so, n/k of them will take time $\Theta(\frac{n}{k}k^2) = \Theta(nk)$.

b. **Start merging it at the level in which each array has size at most k**. This means that the depth of the merge tree is **log(n) - log(k) = log(n/k)**. Each level of merging is still time **cn**, so putting it Together, the merging takes time **Θ (nlog(n/k))**.

c. If order for Θ(nk + nlog(n/k)) = Θ(nlogn), either nk=nlogn or nlog(n/k) = nlogn. From these two possibilities we know the largest asymptotic value for k is Θ(logn)

d. A constant choice of k is optimal.

Heap Sort

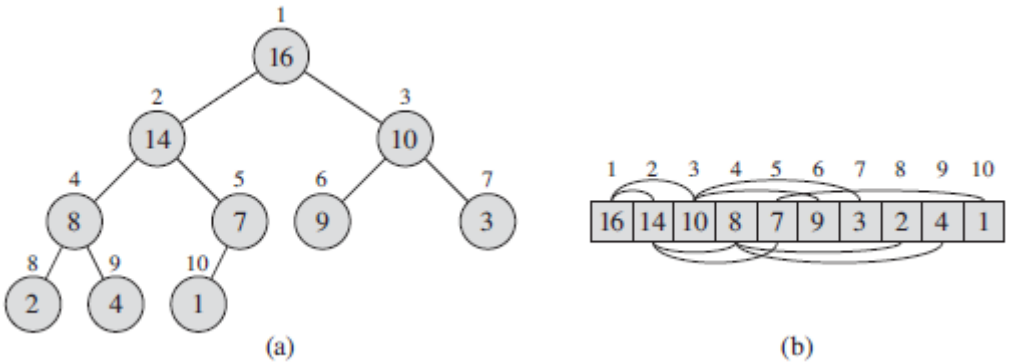
1. Like merge sort, but unlike insertion sort, heapsort's running time is **O(nlogn)**
2. Like insertion sort, but unlike merge sort, heapsort sorts **in place**. Only a constant number of array elements are stored outside the input array at any time.
3. Thus, **heapsort combines the better attributes of the two sorting algorithms.**

Heaps:

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**. An array A that represents a heap is an object with two attributes: **A.length**, which gives the number of elements in array and **A.heap-size**, which represents how many elements in the heap are stored within **array A**.

That is, although **A[1 . . A.length]** may contain elements, only the elements in **A[1 . . A.heap-size]**, where **0 <= A.heap-size <= A.length**, are valid elements of the heap. 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**.



A max-heap viewed as (a) a binary tree and (b) an array

```
PARENT(i)
1  return ⌊i/2⌋

LEFT(i)
1  return 2i

RIGHT(i)
1  return 2i + 1
```

There are two kinds of binary heaps: **max-heaps** and **min-heaps**. In both kinds, the values in the nodes satisfy a heap property.

Max-Heap: In a max-heap, the **max-heap** property is that for every node i other than the root,
$A[\text{PARENT}(i)] \geq A[i]$
That is, the value of a node is **at most the value of its parent**. 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.

Min-Heap: The min-heap property is that for every node i other than the root:
$A[\text{PARENT}(i)] \leq A[i]$ The smallest element is at the root.

Note: -

a. **Heapsort algorithm uses max-heaps**
b. **Min-heaps commonly implement priority queue.**

Height of a Node: number of edges on the longest simple downward path from the node to a leaf.
Height of Heap: height of root node.

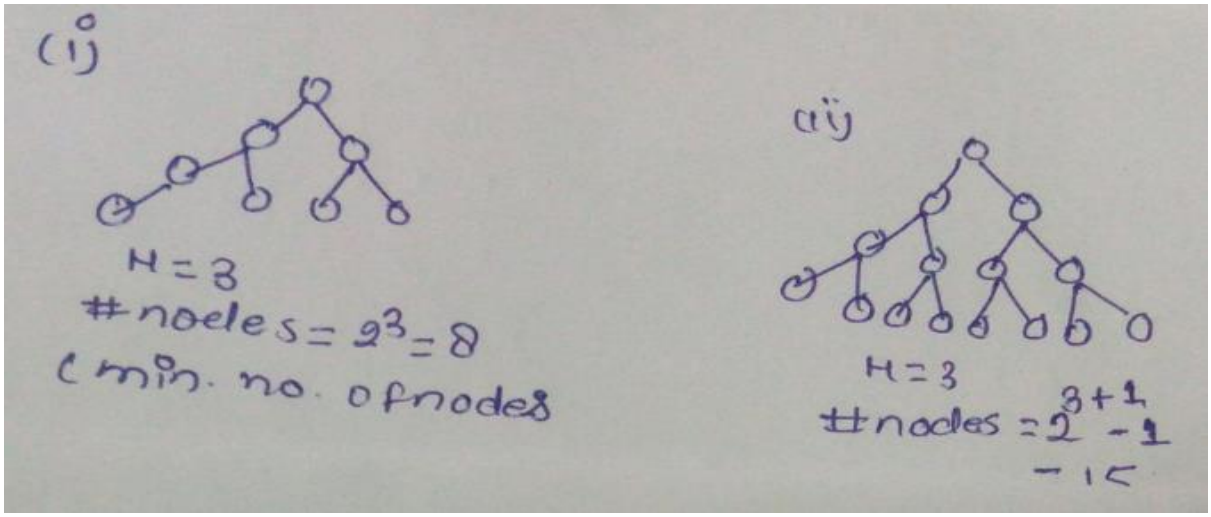
Note: -

1. Since a heap of n elements is based on a complete binary tree, its height is **Θ(logn).**
2. The basic operations on heaps run in time at most proportional to the height of the tree and thus take **Θ(logn)** time.

- The **MAX-HEAPIFY** procedure, which runs in **O(logn)** time, is the key to maintaining the max-heap property.
- The **BUILD-MAX-HEAP** procedure, which runs in linear time **O(n)**, produces a max-heap from an unordered input array.
- The **HEAPSORT** procedure, which runs in **O(nlogn)** time, sorts an array in place.
- The **MAX-HEAP-INSERT**, **HEAP-EXTRACT-MAX**, **HEAP-INCREASE-KEY**, and **HEAP-MAXIMUM** procedures, which run in **O(log n)** time, allow the heap data structure to implement a priority queue.

Note: - in a heap tree the leaves are the nodes indexed by $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n.$

Que 6.1-1 What are the minimum and maximum numbers of elements in a heap of height h?
Solution:
Maximum elements when tree is completely filled at all levels i.e. **2^(h+1) -1**
Minimum elements when there is only one node at last level i.e. **2^h**



Que 6.1-2 Show that an n-element heap has height $\lfloor \lg n \rfloor$

Solution: n = (2^h) – 1 + k, Then **the heap consists of a complete binary tree of height h -1**, along with k additional leaves along the bottom. The height of the root is the length of the longest simple path to one of these k leaves, which must have length h.

It is clear from the way we defined h that h = $\lfloor \lg n \rfloor$

Que 6.1-4 where in a max-heap might the smallest element reside, assuming that all elements are distinct?

Solution:

The smallest element must be a leaf node. Suppose that node x contains the smallest element and x is not a leaf. Let y denote a child node of x. By the max-heap property, the value of x is greater than or equal to the value of y. Since the elements of the heap are distinct, the inequality is strict.

Que 6.1-5 Is an array that is in sorted order a min-heap?

Yes, it is. The index of a child is always greater than the index of the parent, so the heap property is satisfied at each vertex.

Maintaining the heap property

In order to maintain the **max-heap property**, we call the procedure **MAX-HEAPIFY**. Its inputs are an array A and an index i into the array. When it is called, MAXHEAPIFY assumes that

1. The binary trees rooted at **LEFT[i]** and **RIGHT[i]** are maxheaps, but that **A[i]** might be smaller than its children, thus violating the max-heap property.
2. 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 l ≤ A.heap-size and A[l] > A[i]
4       largest = l
5  else largest = i
6  if r ≤ A.heap-size and A[r] > A[largest]
7       largest = r
8  if largest ≠ i
9       exchange A[i] with A[largest]
10      MAX-HEAPIFY(A, largest)
```

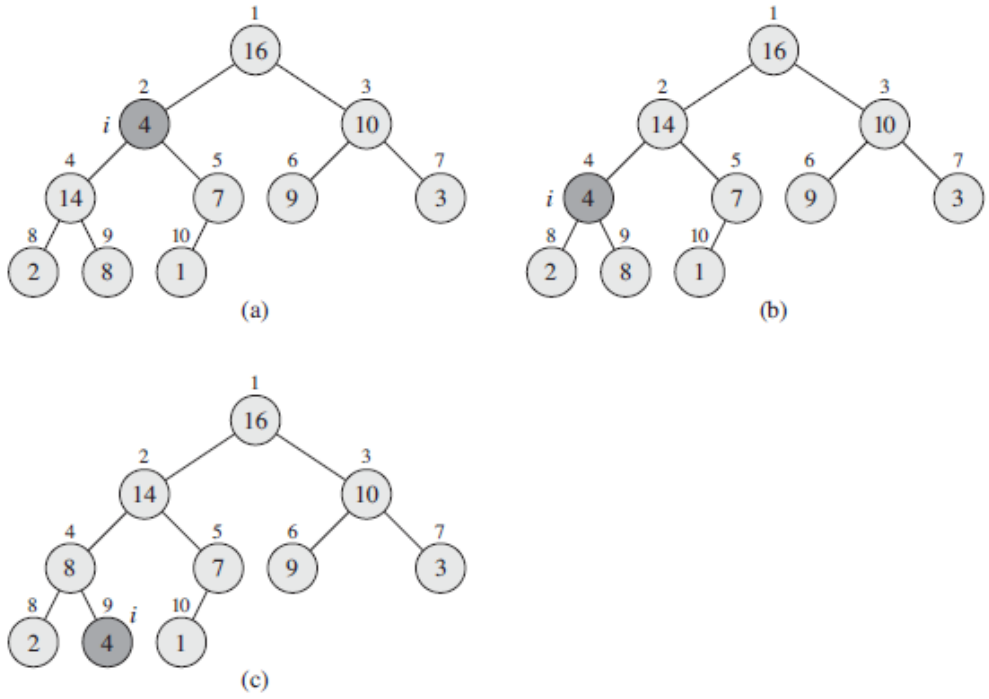


Figure 6.2 The action of MAX-HEAPIFY$(A, 2)$, where $A.heap\text{-}size = 10$. **(a)** The initial configuration, with $A[2]$ at node $i = 2$ violating the max-heap property since it is not larger than both children. The max-heap property is restored for node 2 in **(b)** by exchanging $A[2]$ with $A[4]$, 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.

Line 3 & 4: **index l and r must be less than equal to total size of heap**.

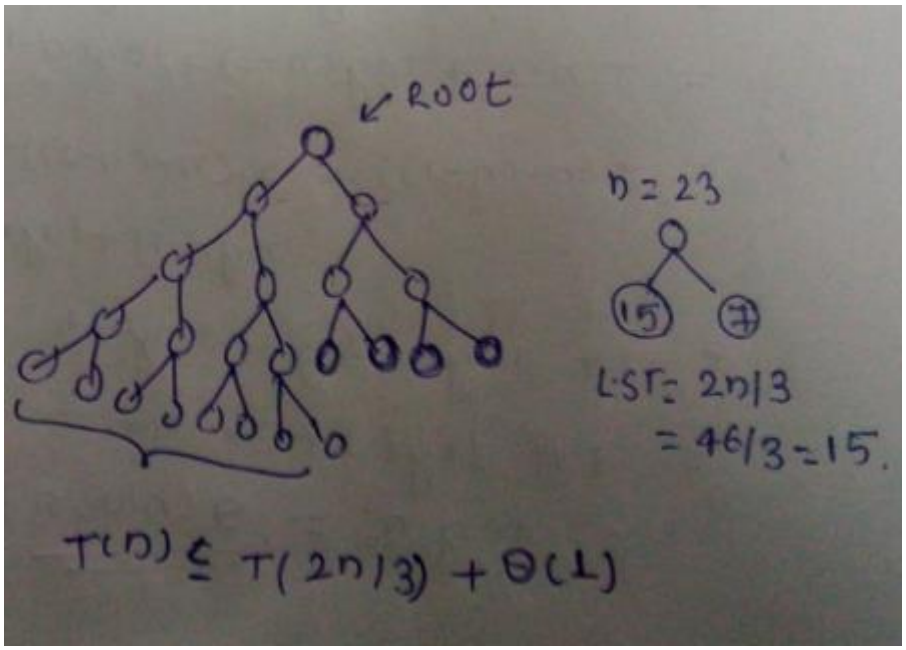
1. At each step, the largest of the elements A[i], A[left], and A[right] is determined, and its index is stored in largest.
2. If A[i] is largest, then the subtree rooted at node i is already a max-heap and the procedure terminates.
3. Otherwise, one of the two children has the largest element, and A[i] is swapped with A[largest], which causes node i and its children to satisfy the max-heap property.
4. Subtree rooted at largest might violate the max-heap property. Consequently, we call MAX-HEAPIFY recursively on that subtree.

The running time of MAX-HEAPIFY on a subtree of size n rooted at a given node i is the **Θ(1)** time to fix up the relationships among the elements A[i], A[LEFT], and A[RIGHT], plus the time to run MAX-HEAPIFY on a subtree rooted at one of the children of node i.
The children's subtrees each have size **at most 2n/3**—the worst case occurs when the bottom level of the tree is exactly half full and therefore we can describe the running time of MAX-HEAPIFY by the recurrence.

$$T(n) \leq T(2n/3) + \Theta(1)$$

a=1, b=3/2, n^(loga (baseb)) = log 1 base(3/2) hence n^0 = 1, thus **T.C Θ(logn)**

Alternatively, we can characterize the running time of MAXHEAPIFY on a node of height h as **O(h)**.



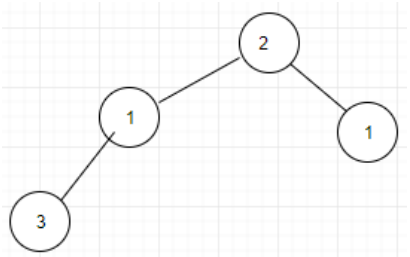
LST contains 2n/3 nodes

BUILD-MAX-HEAP

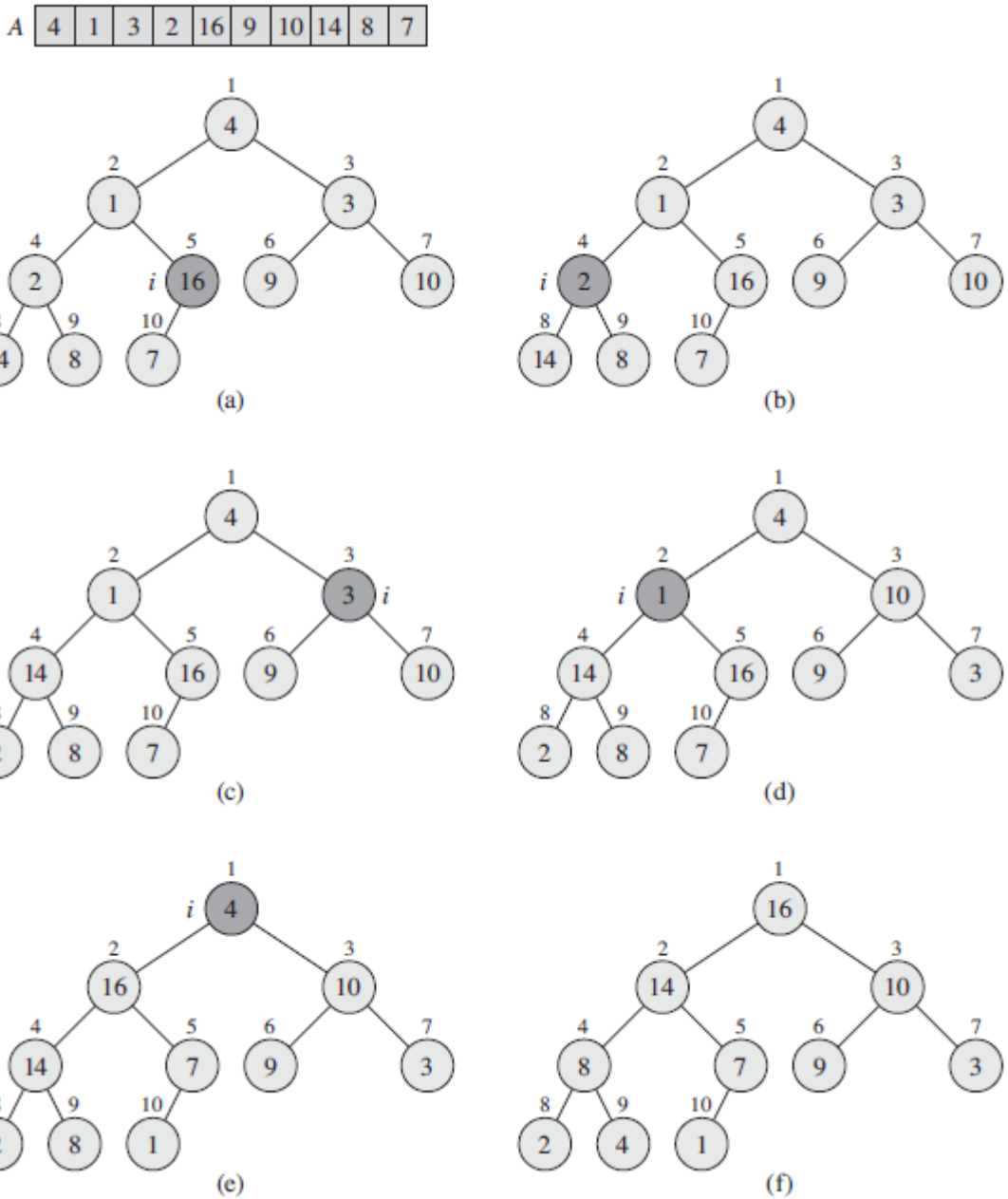
```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

6.3-2 why do we want the loop index i in line 2 of BUILD-MAX-HEAP to decrease from $\lfloor A.length/2 \rfloor$ to 1 rather than increase from 1 to $\lfloor A.length/2 \rfloor$?

Ans: If we had started at 1, we wouldn't be able to guarantee that the max-heap property is maintained.
For example, if the array A is given by **[2, 1, 1, 3]** then MAX-HEAPIFY **won't** exchange 2 with either of its children, both 1's. But, when MAX-HEAPIFY is called on the left child, 1, it will swap 1 with 3.



If we start at root, Max-heapify fails



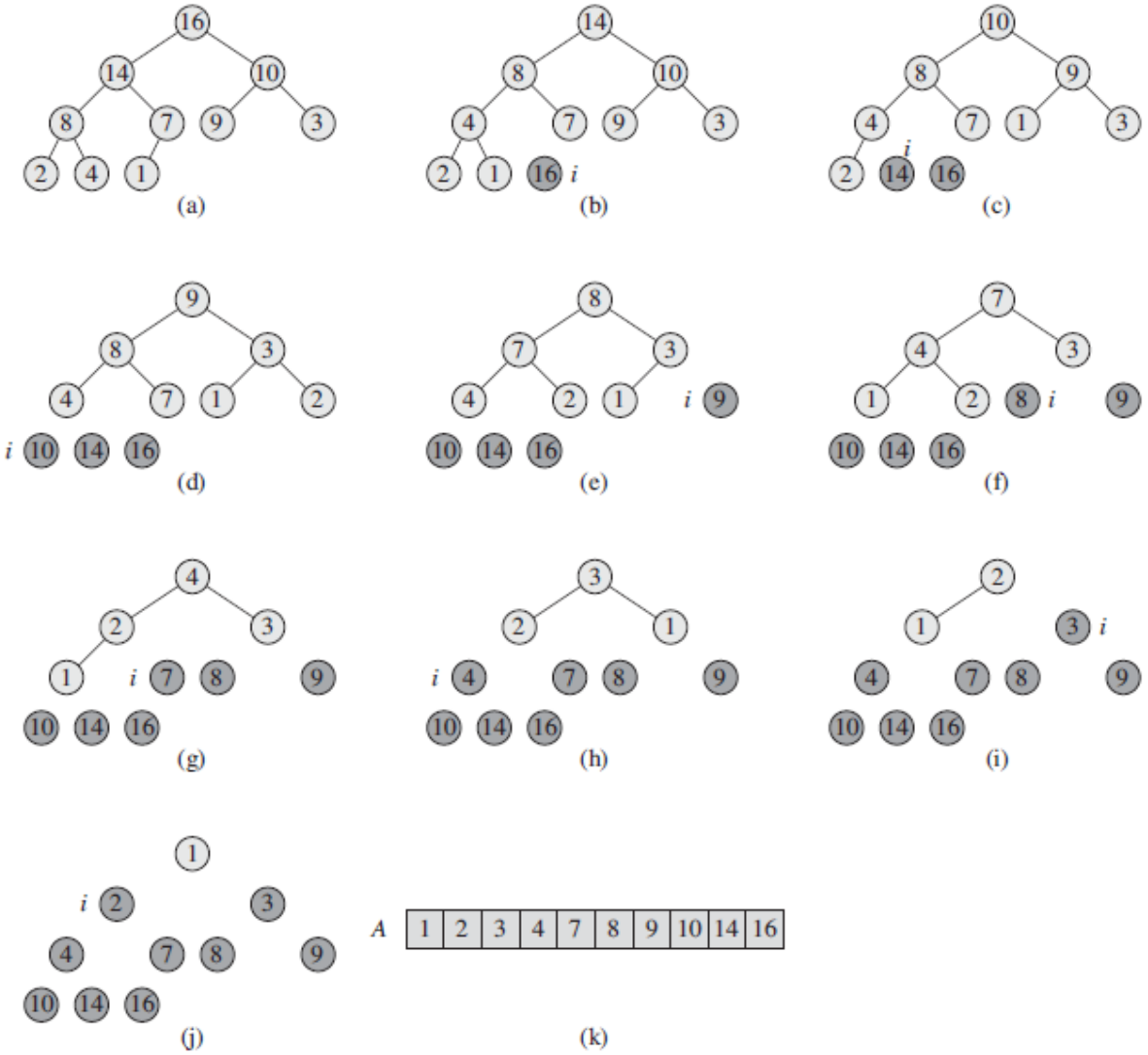
Build Max-Heap Procedure

The heapsort algorithm

The heapsort algorithm starts by using **BUILD-MAX-HEAP** to build a max-heap on the input array A[1…….n] where n=A.length.

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```

1. Since the maximum element of the array is stored at the root A[1], we can put it into its correct final position by exchanging it with A[n].
2. If we discard nth node from the heap, and we can do so by simply decrementing **A.heap-size – 1**;
3. We observe that the children of the root **remain** max-heaps, but the new root element might violate the max-heap property.
4. To restore the max-heap property, **MAX-HEAPIFY(A, 1)** is called, which leaves a max-heap in A[1……. N-1].
5. The heapsort algorithm then repeats this process for the max-heap of size n - 1 down to a heap of size 2. Following is example of **heapsort** algorithm.



The HEAPSORT procedure takes time O(nlogn), since the call to BUILD-MAXHEAP takes time O(n) and each of the n - 1 calls to MAX-HEAPIFY takes time O(logn).

Properties:-

1. Best, average and worst case T.C is O(nlogn)
2. Merge Sort outperforms heap sort in most of the practical cases.
3. Heapsort is **in-place** sorting algorithm
4. Heap Sort is **not stable**.
5. Heapsort can be adapted to operate on doubly linked lists with only **O(1)** extra space.

Que 6.4-3 What is the running time of **HEAPSORT** on an array A of length n that is **already sorted** in increasing order? What about **decreasing order**?

Solution:

The running time of HEAPSORT on an array of length n that is already sorted in increasing order is **Θ(nlogn)** because even though it is already sorted, it will be transformed back into a max-heap in O(n) time and then sorted.

Same goes for decreasing order. **BUILD-MAX-HEAP** will be faster (by a constant factor), but the computation time will be dominated by the loop in HEAPSORT, which is **Θ(nlogn)**.

Priority queues

As with heaps, priority queues come in two forms**: max-priority queues** and **min-priority queues**. We will focus here on how to implement max-priority queues, **which are based on maxheaps**.

A **priority queue** is a data structure for maintaining a set S of elements, each with an associated value called a **key**. A **max-priority** queue supports the following operations:

1. **INSERT(S, x)** inserts the element x into the set S, which is equivalent to the operation S = S U {x}.
2. **MAXIMUM(S)** returns the element of S with the largest key.
3. **EXTRACT-MAX(S)** removes and returns the element of S with the largest key.
4. **INCREASE-KEY(S, x, k)** increases the value of element x's key to the new value k, which is assumed to be at least as large as x's current key value.

Note: -

1. We can use max-priority queues to schedule jobs on a shared computer.
2. **Min-priority queue** supports the operations INSERT, MINIMUM, EXTRACT-MIN, & DECREASEKEY.
3. A min-priority queue can be used in an event-driven simulator. The items in the queue are events to be simulated, each with an associated time of occurrence that serves as its key. The events must be simulated in order of their time of occurrence.

Now we discuss how to implement the operations of a max-priority queue.

1. **Θ(1)** time, it returns root element.

```
HEAP-MAXIMUM(A)
1  return A[1]
```

2. **Θ(logn)** time. It extracts root element, replace with last element and apply max-heapify.

```
HEAP-EXTRACT-MAX(A)
1  if A.heap-size < 1
2       error "heap underflow"
3  max = A[1]
4  A[1] = A[A.heap-size]
5  A.heap-size = A.heap-size − 1
6  MAX-HEAPIFY(A, 1)
7  return max
```

3. It updates the key A[i] to its new value. Because increasing the key of A[i] might violate the max-heap property. HEAP-INCREASE-KEY on an n-element heap is **O(log n)**. It repeatedly compares an element to its parent, exchanging their keys and continuing if the element's key is larger.

```
HEAP-INCREASE-KEY(A, i, key)
1  if key < A[i]
2       error "new key is smaller than current key"
3  A[i] = key
4  while i > 1 and A[PARENT(i)] < A[i]
5       exchange A[i] with A[PARENT(i)]
6       i = PARENT(i)
```

4. It first expands the max-heap by adding to the tree a new leaf whose key is **- infinity**. Then it calls HEAP-INCREASE-KEY to set the key of this new node to its correct value and maintain the max-heap property. The running time of MAX-HEAP-INSERT on an n-element heap is **O(logn)**.

```
MAX-HEAP-INSERT(A, key)
1  A.heap-size = A.heap-size + 1
2  A[A.heap-size] = −∞
3  HEAP-INCREASE-KEY(A, A.heap-size, key)
```

Que 6.5-8
The operation HEAP-DELETE(A, i) deletes the item in node i from heap A. Give an implementation of HEAP-DELETE that runs in O(logn) time for an n-element max-heap.
Solution
Replace the node to be deleted by the last node of the heap. Update the size of the heap, then call MAX-HEAPIFY to move that node into its proper position. **O(logn)**

Algorithm 2 HEAP-DELETE(A,i)

```
1: A[i] = A[A.heap − size]
2: A.heap − size = A.heap − size - 1
3: MAX-HEAPIFY(A,i)
```

Que 6.5-9 Give an O(nlogk) time algorithm to merge k sorted lists into one sorted list, where n is the total number of elements in all the input lists.
Solution:
There are n elements, and k sorted list of n/k elements each.
1. Construct a min heap from the heads of each of the k lists.
2. To find the next element in the sorted array, extract the minimum element (in **Olog(k)** time)
3. Then, add to the heap the next element from the shorter list from which the extracted element originally came i.e **O(logk)**
4. Time to find the whole list, you need O**(nlog(k))** total steps.

Young tableaus:
An m x n Young tableau is an **m x n matrix** such that the entries of each row are in sorted order from left to right and the entries of each column are in sorted order from top to bottom. Some of the entries of a Young tableau may be **infinity**, which we treat as non-existent elements.

Properties:-
1. To insert a new element into a non-full m x n Young tableau in **O(m+n)** time.
2. To determine whether a given number is stored in a given m x n Young tableau is **O(m+n)**.
3. Using no other sorting method as a subroutine, an n x n Young tableau uses to sort **n^2 numbers in O(n^3) time**.

Quick Sort

Description of quicksort

Quicksort, like merge sort, applies the divide-and-conquer paradigm.

Divide: Partition (rearrange) the array A[p……r] into two (possibly empty) subarrays **A[p……q-1] and A[q+1…….r]** such that **each element of A[p……q-1] is less than or equal to A[q]**, which is, in turn, **less than or equal to each element of A[q+1…….r]**. Compute the index q as part of this partitioning procedure.

Conquer: Sort the two subarrays A[p……q-1] and A[q+1…….r] by recursive calls to quicksort.

Combine: Because the subarrays are already sorted, no work is needed to combine them: the entire array A[p……r] is now sorted.

The following procedure implements quicksort(ArrayName, StartIndex, length)

```
QUICKSORT(A, p, r)
1  if p < r
2      q = PARTITION(A, p, r)
3      QUICKSORT(A, p, q − 1)
4      QUICKSORT(A, q + 1, r)
```

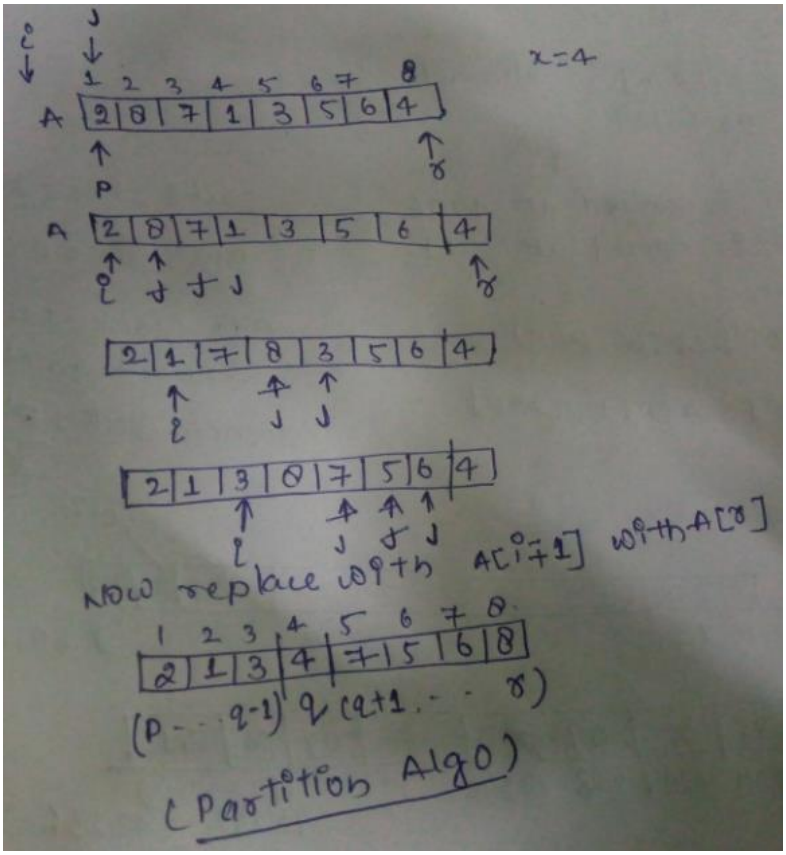
Note- To sort an entire array A, the initial call is **QUICKSORT(A, 1, A.length).**

Partitioning the array

The key to the algorithm is the PARTITION procedure, which rearranges the subarray A[p…r] in place.

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

1. We're choosing the last element of the array A[p…r] as **pivot** element.
2. i is assigned to 0 (p – 1, array starts with 1)
3. j for loop starts from p (first element) and goes till r – 1 (n-1th element)
4. If A[j] is **smaller or equal** to pivot element then increase i.
5. Exchange A[i] and A[j]
6. After the end of for loop, exchange A[i+1] and pivot element A[r]
7. Return i+1 as the position of pivot element.



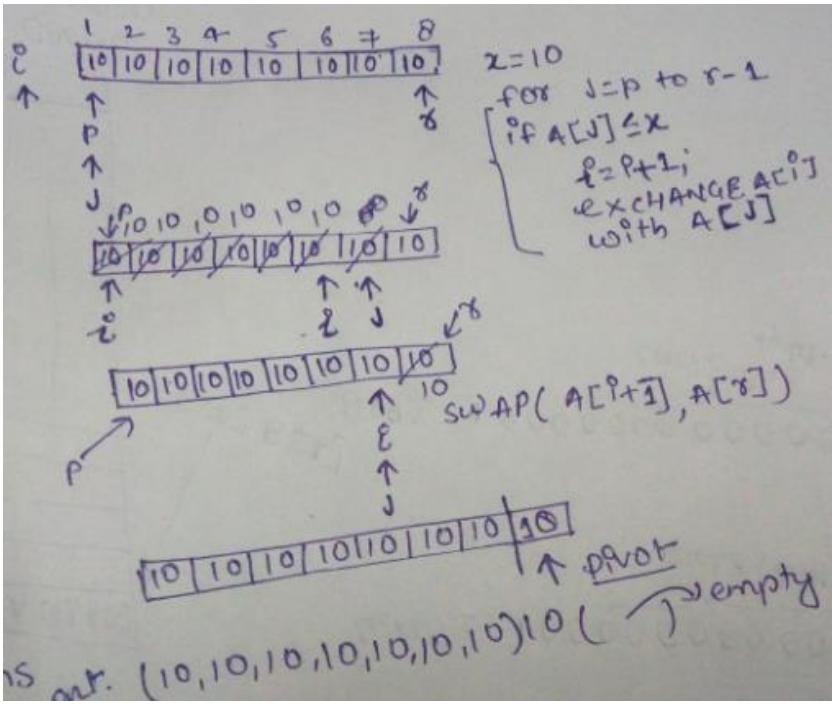
Partition Algorithm

Note: -The running time of **PARTITION** on the subarray A[p….r] is **Θ(n)** , where n= r – p + 1.

7.1-2

What value of q does PARTITION return when all elements in the array $A[p..r]$ have the same value? Modify PARTITION so that $q = \lfloor (p+r)/2 \rfloor$ when all elements in the array $A[p..r]$ have the same value.

Solution:



If all elements in the array have the same value, **PARTITION returns r**.

To make PARTITION return $q = \lfloor (p + r)/2 \rfloor$ (q should be middle of array) when all elements are same value modify line 4 of the partition algorithm to say this:

If A[j] <= x and j(mod2) = (p + 1)(mod2)

This causes the algorithm to treat half of the instances of the same value to count as less than, and the other half to count as greater than.

Performance of quicksort

The running time of quicksort depends on whether the partitioning is **balanced** or **unbalanced**, which in turn depends on which elements are used for partitioning.

1. If the partitioning is **balanced**, the algorithm runs asymptotically as fast as **merge sort**.
2. If the partitioning is **unbalanced**, the algorithm runs asymptotically as slow as **Insertion Sort**.

Worst-case partitioning [T(n) = Θ(n^2)]

The worst-case behavior for quicksort occurs when the partitioning routine produces one subproblem with n - 1 elements and one with 0 elements. Let us assume that this unbalanced partitioning arises in each recursive call. The partitioning costs **Θ(n)** time. The recurrence for the running time is

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) . \end{aligned}$$

Note: - If array is already sorted then Q.S takes Θ (n^2) and Insertion Sort takes O(n)

Best-case partitioning [Θ(nlogn)]

In the most even possible split, PARTITION produces two subproblems, each of size no more than $n/2$, since one is of size $\lfloor n/2 \rfloor$ and one of size $\lceil n/2 \rceil - 1$. In this case, quicksort runs much faster. The recurrence for the running time is then

$$T(n) = 2T(n/2) + \Theta(n) ,$$

Balanced partitioning the average-case running time of quicksort is much closer to the best case than to the worst case.

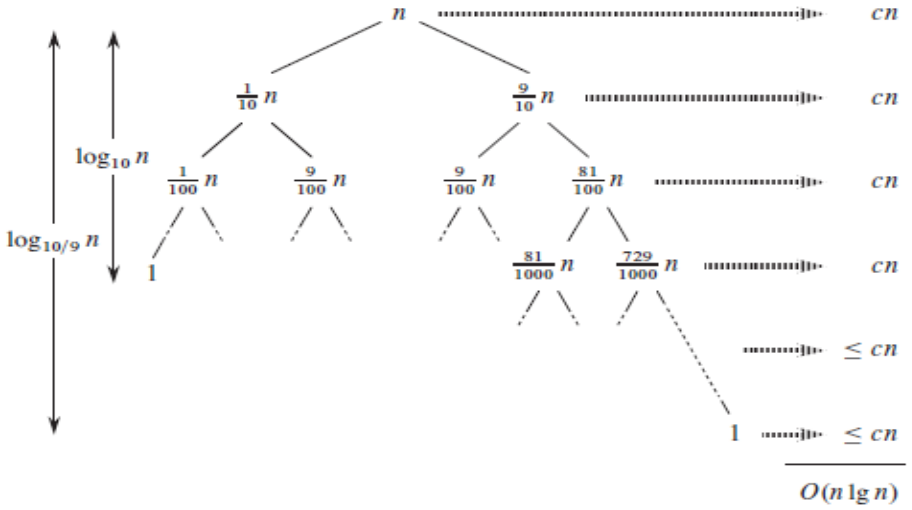
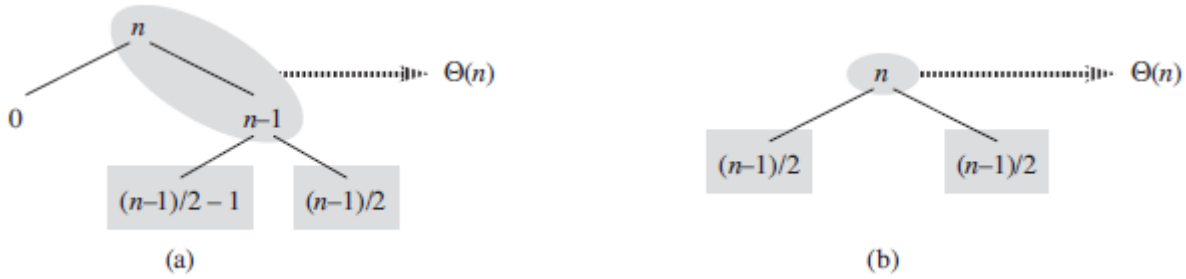


Figure 7.4 A recursion tree for QUICKSORT in which PARTITION always produces a 9-to-1 split, yielding a running time of $O(n \lg n)$. Nodes show subproblem sizes, with per-level costs on the right. The per-level costs include the constant c implicit in the $\Theta(n)$ term.

a. T(n) = T(9n/10) + T(n/10) + cn
b. T(n) = T(99n/100) + T(n/100) + cn

The running time is therefore **O(nlogn)** whenever the **split has constant proportionality**.

Intuition for the average case



Good split and bad split

In the average case, PARTITION produces a mix of "**good**" and "**bad**" splits. In a recursion tree for an average-case execution of PARTITION, the good and bad splits are distributed randomly throughout the tree.
At the root of the tree, cost is n for partitioning, and the subarrays produced have sizes **n - 1 and 0** the worst case. At the next level, the subarray of size n - 1 undergoes best-case partitioning into subarrays of size (n-1)/2 -1 and (n – 1)/2.

Que 7.2-2 What is the running time of QUICKSORT when all elements of array A have the same value?
Solution:
The running time of QUICKSORT on an array in which every element has the same value is **n^2**. This is because the partition will always occur at the last position of the array.

A randomized version of quicksort

Instead of always using A[r] as the pivot, we will select a randomly chosen element from the subarray A[p…r] We do so by first exchanging element A[r] with an element chosen at random from A[p…r] By randomly sampling the range p….r, we ensure that the pivot element x = A[r] is equally likely to be any of the r - p + 1 elements in the subarray. Because we randomly choose the pivot element, we expect the split of the input array to be reasonably well balanced on average.

```
RANDOMIZED-PARTITION(A, p, r)
1  i = RANDOM(p, r)
2  exchange A[r] with A[i]
3  return PARTITION(A, p, r)
```

The new quicksort calls RANDOMIZED-PARTITION in place of PARTITION:

```
RANDOMIZED-QUICKSORT(A, p, r)
1  if p < r
2      q = RANDOMIZED-PARTITION(A, p, r)
3      RANDOMIZED-QUICKSORT(A, p, q − 1)
4      RANDOMIZED-QUICKSORT(A, q + 1, r)
```

Analysis of Randomized Quick Sort:-

1. Expected T.C for Randomized Q.S is **Θ(nlogn)**
2. Worst case T.C is Still **O(n^2)** when either all elements are same or the randomized pivot selector happens to select smallest element in a row n time. Though this possibility is only **1/n!**

Note: T(n) = T(q) + T(n – q – 1) + Θ(n), T(n) = omega(n^2)

Comparison based Sorting Algoirthms

All the sorting algorithms introduced thus far are comparison sorts. The sorted order they determine is based only on comparisons between the input elements. We call such sorting algorithms **comparison sorts**.

Any comparison sort must make **omega(nlogn) comparisons** in the worst case to sort n elements.

Note: - Heapsort and merge sort are asymptotically **optimal comparison sorts**.

The decision-tree model:

A **decision tree** is a full binary tree that represents the comparisons between elements that are performed by a particular sorting algorithm operating on an input of a given size.

With n elements, there are total n! Permutations possible. Each leaf node in decision represents 1 of n! Permutations.

Number of comparisons to sort n elements: Ceil (logn!)

Sorting in Linear Time

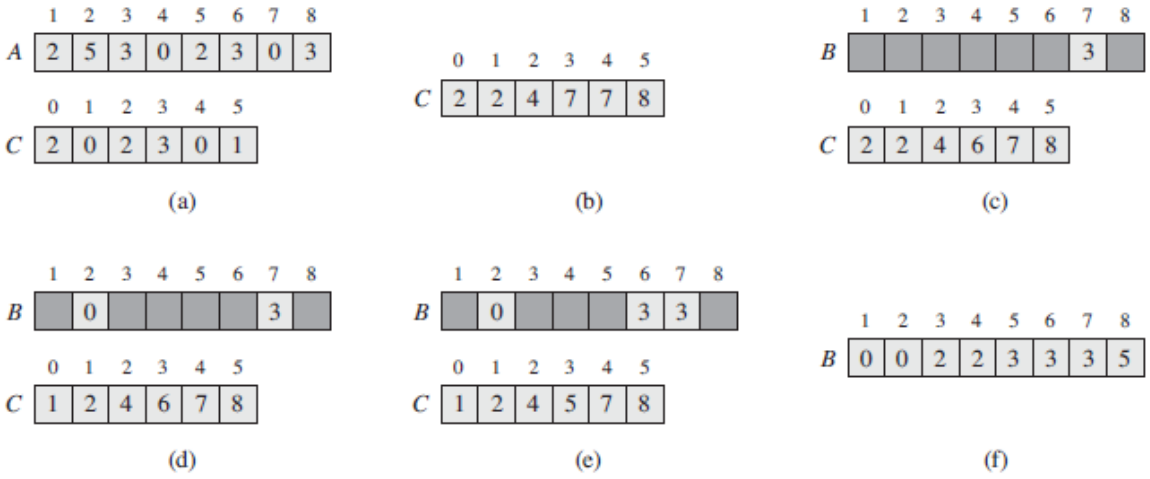
Counting Sort (Stable Sort)

Counting sort assumes that **each of the n input** elements is an integer in the range **0 to k**, for some integer k. When k = **O(n)**, the sort runs in **Θ(n)** time.

Counting sort determines, for each input element x, the number of elements less than x. It uses this information to place element x directly into its position in the output array.

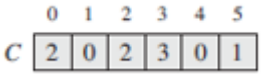
In the code for counting sort, we assume that

1. **input is an array A[1…n]**, and thus A.length = n.
2. We require two other arrays: the **array B[1…n] holds the sorted output**.
3. Array **C[0..k]** provides temporary working storage, size of this array is k+1.

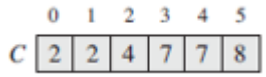


K = Max element in array

1. C[i]: How many times the element i appears in array A. For example 0 appeared two times in array A, hence C[0]=2, or 3 is three times in array A.



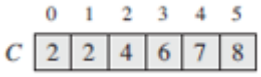
2. Start from second element in array C and add the value of that cell C[i] with the value of previous cell C[i-1].



Start from the last element in array A.

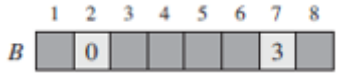
1. Last element in array A is 3, in array C, it is C[3]=7.
2. We will store 3 at position 7 in array B, and we will decrease value of C[3] by 1.

(3 is stored at 7th position)

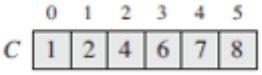


(C[3] is decreased by 1)

3. Repeat the procedure with second last element.
    a. Second last element is 0 in array A, C[0] = 2, hence 0 will be stored at B[2].

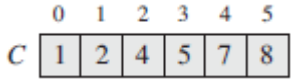


    b. C[0] is decremented by 1.

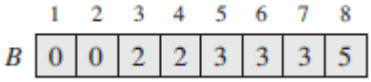


4. Repeat the procedure with 3rd last element
    a. 3rd last element in A is 3, C[3] = 6, hence 3 will be stored at B[6]

    b. C[3] will be decreased by 1



5. At the end, Array will have elements in sorted manner:



Overall time is **Θ(k+n).** In practice, we usually use counting sort when we have k = O(n).

Properties:-

1. Counting sort is that it is **stable**.
2. Counting sort is **not in-place**.
3. Counting sort is often used as a subroutine in radix sort.

Radix sort

Radix sort solves the problem by sorting on the least significant digit first. If LSB is same then write the no. first which appears first in the input array.
The process continues until the cards have been sorted **on all d digits**. To sort d digit numbers, d passes are required.

```
RADIX-SORT(A, d)
1  for i = 1 to d
2      use a stable sort to sort array A on digit i
```

The above procedure assumes that **each element in the n-element array A has d digits**, where digit 1 is the lowest-order digit and digit **d is the highest-order digit**.

Given n **d-digit numbers in which each digit can take on up to k possible values**, RADIX-SORT correctly sorts these numbers in **Θ(d(n+k))** time if the stable sort it uses takes **Θ(n+k)** time.

Note: - When d is constant and k = O(n) we can make radix sort run in linear time.

8.3-4

Show how to sort n integers in the range 0 to $n^3 - 1$ in $O(n)$ time.

First run through the list of integers and convert each one to base n, then radix sort them. Each number will have at most $\log_n(n^3) = 3$ digits so there will only need to be 3 passes. For each pass, there are n possible values which can be taken on, so we can use counting sort to sort each digit in $O(n)$ time.

Note:- We can sort an array of integers with range from 1 to n^c, if the numbers are represented in base n, each number will have c digits.

Elementary Data Structures

Stacks

We can implement a stack of at most n elements with an array S[1…n]. The array has an attribute **S.top** that indexes the most recently inserted element.

The stack consists of elements S[1… S.top], where **S[1] is the element at the bottom of the stack** and **S[Top] is the element at the top**.

When S.top = 0, the stack contains no elements and is empty.

We can implement each of the stack operations with just a few lines of code:

```
STACK-EMPTY(S)
1  if S.top == 0
2       return TRUE
3  else return FALSE

PUSH(S, x)
1  S.top = S.top + 1
2  S[S.top] = x

POP(S)
1  if STACK-EMPTY(S)
2       error "underflow"
3  else S.top = S.top − 1
4       return S[S.top + 1]
```

PUSH First increment TOP and then insert the TOP.
POP First copy the element and then decrement the TOP

Each of the three stack operations takes O(1) time.

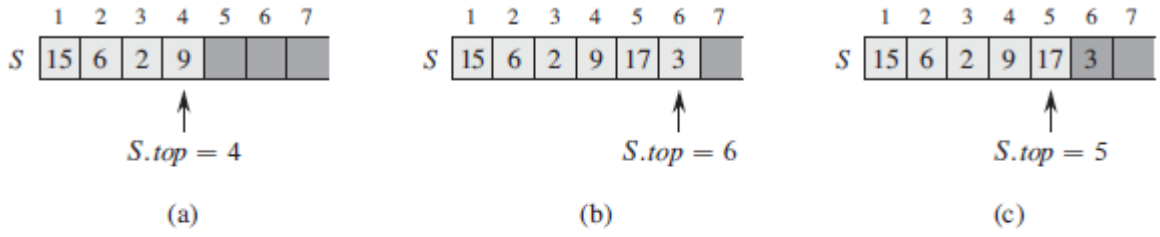


Figure 10.1 An array implementation of a stack S. Stack elements appear only in the lightly shaded positions. (a) Stack S has 4 elements. The top element is 9. (b) Stack S after the calls PUSH$(S, 17)$ and PUSH$(S, 3)$. (c) Stack S after the call POP(S) has returned the element 3, which is the one most recently pushed. Although element 3 still appears in the array, it is no longer in the stack; the top is element 17.

Queues

DEQUEUE takes no element argument. The **FIFO** property of a queue causes it to operate like a line of customers waiting to pay a cashier. **The queue has a head (front) and a tail (rear)**.

1. When an element is **enqueued**, it takes its place at the tail of the queue, just as a newly arriving customer takes a place at the end of the line.
2. The element **dequeued** is always the one at the head of the queue, like the customer at the head of the line who has waited the longest.

Figure 10.2 shows one way to implement a queue of at **most n - 1 elements** using an array Q[1..n]

1. The queue has an attribute **Q.head** points to, its head.
2. The attribute **Q.tail** indexes the next location at which a newly arriving element will be inserted into the queue
3. The elements in the queue reside in locations Q.head; Q.head+1; ….. Q.tail-1.
4. Location 1 immediately follows location n in a **circular order**.
5. ENQUEUE and DEQUEUE operations, Each operation takes **O(1)** time.

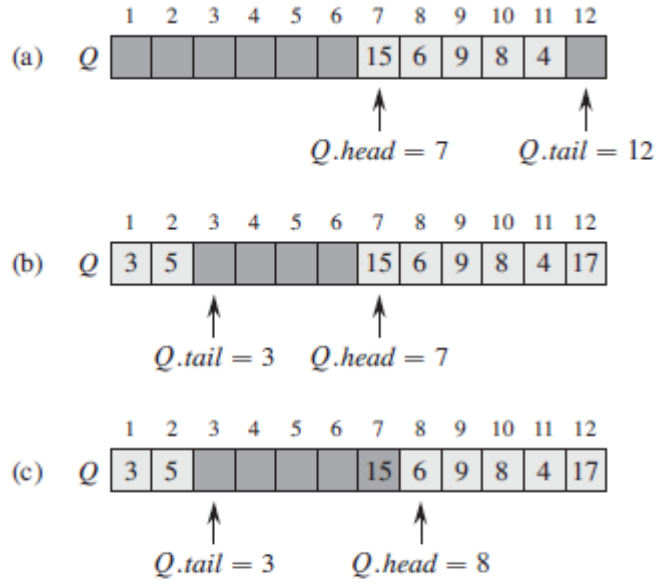


Figure 10.2 A queue implemented using an array $Q[1..12]$. Queue elements appear only in the lightly shaded positions. **(a)** The queue has 5 elements, in locations $Q[7..11]$. **(b)** The configuration of the queue after the calls ENQUEUE$(Q, 17)$, ENQUEUE$(Q, 3)$, and ENQUEUE$(Q, 5)$. **(c)** The configuration of the queue after the call DEQUEUE(Q) returns the key value 15 formerly at the head of the queue. The new head has key 6.

Note: -

1. **Q.head = Q.tail, the queue is empty.**
2. **Initially, we have Q.head = Q.tail = 1.**
3. **When Q.head = Q.tail +1, the queue is full.**

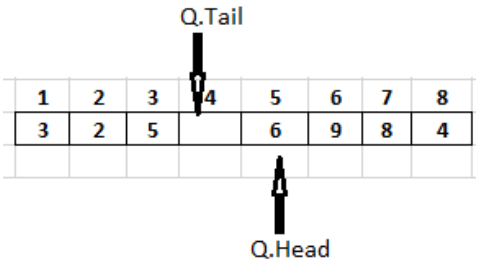
In our procedures ENQUEUE and DEQUEUE, we have omitted the error checking for underflow and overflow. The pseudocode assumes that **n = Q.length**.

```
ENQUEUE(Q, x)
1  Q[Q.tail] = x
2  if Q.tail == Q.length
3        Q.tail = 1
4  else Q.tail = Q.tail + 1
```

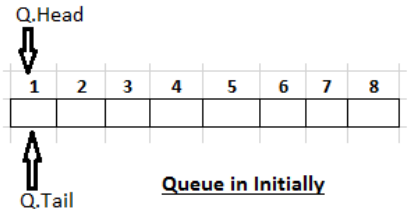
```
DEQUEUE(Q)
1  x = Q[Q.head]
2  if Q.head == Q.length
3        Q.head = 1
4  else Q.head = Q.head + 1
5  return x
```

Queues Underflow and Overflow Conditions:

Enqueue: Insert the element and then increment the pointer.
Dequeue: Copy the element and then increment the pointer.



If (**Q.head == Q.tail + 1** or **Q.head == 1 && Q.tail == Q.length**)
then "**error overflow**" (1 cell is wasted)



if **Q.tail == Q.head** then
error "**underflow**"

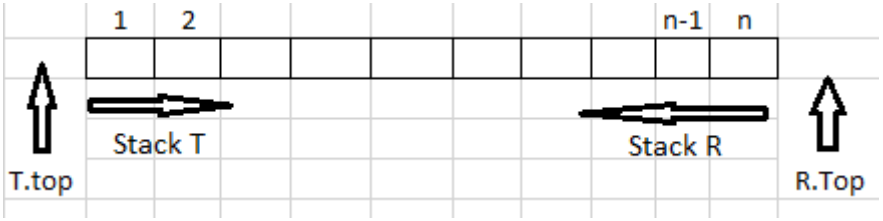
Que 10.1-2 Explain how to implement **two stacks** in one array **A[1…n]** in such a way that neither stack overflows unless the total number of elements in both stacks together is n. The PUSH and POP operations should run in **O(1)** time.
Solution

Array is A[1…n] starting from 1 to n.

We will call the stacks T and R. Initially, **set T.top = 0 and R.top = n + 1**.

1. Stack T uses the first part of the array and stack.
2. R uses the last part of the array.

3. In stack T, the top is the rightmost element of T.
4. In stack R, the top is the leftmost element of R.



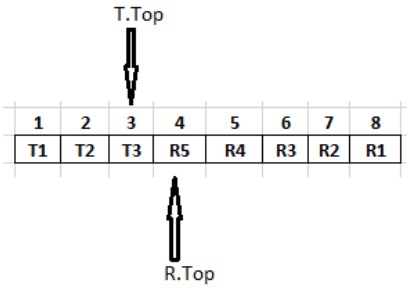
Initially position of T.Top and R.Top

Algorithm 1 PUSH(S,x)

```
1: if S == T then
2:     if T.top + 1 == R.top then
3:         error "overflow"
4:     else
5:         T.top = T.top + 1
6:         T[T.top] = x
7:     end if
8: end if
9: if S == R then
10:    if R.top − 1 == T.top then
11:        error "overflow"
12:    else
13:        R.top = R.top − 1
14:        T[T.top] = x
15:    end if
16: end if
```

Note- S is a stack pointer formal parameter to function PUSH.Stack name will be provided as actual parameter while calling the function PUSH

1. Overflow condition



If S = T then increment TOP and insert the element.

2. Overflow condition if T.Top = R.Top-1
   If S=R then decrement the TOP and insert the element.

Algorithm 2 POP(S)

```
if S == T then
    if T.top == 0 then
        error "underflow"
    else
        T.top = T.top − 1.
        return T[T.top + 1]
    end if
end if
if S == R then
    if R.top == n + 1 then
        error "underflow"
    else
        R.top = R.top + 1.
        return R[R.top − 1]
    end if
end if
```

Que 10.1-6 Show how to implement a **queue using two stacks**. Analyse the running time of the queue operations.
Solution:

1. The operation **enqueue** will be same as pushing an element on to **stack 1**. This operation is **O(1)**.
2. To **dequeue**, we pop an element from **stack 2**. If stack 2 is empty, for each element in stack 1 we pop it off, then push it on to stack 2. Finally, pop the top item from stack 2. This operation is **O(n)** in the worst case.

Que 10.1.7 Show how to implement a **stack using two queues**. Analyze the running time of the stack operations.
Solution:
The following is a way of implementing a stack **using two queues**, where

1. Pop takes linear time **O(n)**
2. Push takes constant time **O(1)**

Push and Pop are not fixed to any particular queue.
PUSH( )

1. Enqueue items in queue 1.

POP()

1. If queue contains more than 1 element, dequeue elements from Queue1 and place them on Queue2, except the last element. Return the single element left in Queue1.
2. Then switch the name of queues queue1 and queue2.

Linked lists:

A linked list is a data structure in which the objects are arranged in a linear order. Unlike an array, however, in which the linear order is determined by the array indices, the order in a linked list is determined by a pointer in each object.

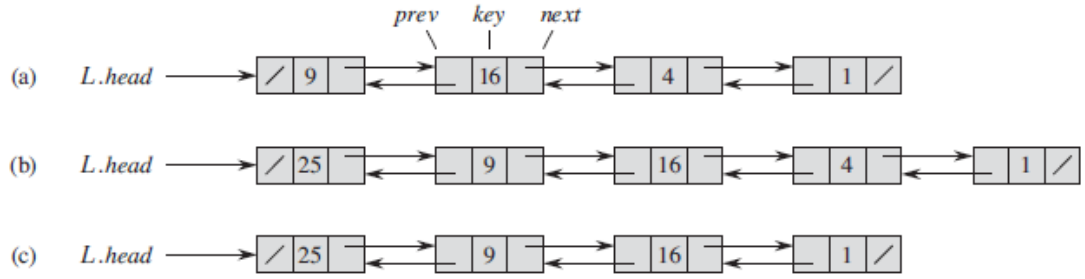
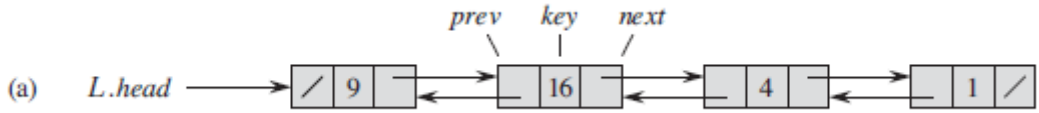


Figure 10.3 (a) A doubly linked list L representing the dynamic set $\{1, 4, 9, 16\}$. Each element in the list is an object with attributes for the key and pointers (shown by arrows) to the next and previous objects. The *next* attribute of the tail and the *prev* attribute of the head are NIL, indicated by a diagonal slash. The attribute $L.head$ points to the head. (b) Following the execution of LIST-INSERT(L, x), where $x.key = 25$, the linked list has a new object with key 25 as the new head. This new object points to the old head with key 9. (c) The result of the subsequent call LIST-DELETE(L, x), where x points to the object with key 4.

Doubly Linked List

L is an object with an attribute **key** and two other pointer attributes: **next** and **pre**. The object may also contain other satellite data.

1. Given an element x in the list, **x.next** points to its successor in the linked list, and **x.pre** points to its predecessor.
2. If **x.pre = NIL**, the element x has no predecessor and is therefore the first element, or head, of the list.
3. If **x.next = NIL**, the element x has no successor and is therefore the last element, or tail, of the list.
4. An attribute **L.head** points to the first element of the list. If **L.head = NIL**, the list is empty.



1. If a list is **single linked list** then we omit pre pointer in each element.
2. If a list is **sorted**, minimum element is the head and maximum element is the tail.
3. If a list is **unsorted**, the elements can appear in any order.
4. In a **circular** list, the **pre** pointer of the head of the list points to the **tail**, and the **next** pointer of the tail of the list points to the **head**.

Searching a linked list

The procedure **LIST-SEARCH(L, k)** finds the first element with key k in list L by a simple linear search, returning a pointer to this element. If no object with key k appears in the list, then the procedure returns NIL. The LIST-SEARCH procedure takes **Θ(n)** time in the worst case.

```
LIST-SEARCH(L, k)
1  x = L.head
2  while x ≠ NIL and x.key ≠ k
3      x = x.next
4  return x
```

Inserting into a linked list O(1)

Given an element x whose key attribute has already been set, the **LIST-INSERT** procedure insert x onto the front of the linked list in **O(1)** time.

X is the node to be inserted.

1. X.next will store the pointer, pointed by pointer head.
2. If the linked list is not nil, pre pointer of the node previously pointed by header, will store the address of x. New node will be pointing to the node previously pointed by header.
3. If Linked list was nil, then Head will point to node x, and pre pointer of x will contain null.

```
LIST-INSERT(L, x)
1  x.next = L.head
2  if L.head ≠ NIL
3      L.head.prev = x
4  L.head = x
5  x.prev = NIL
```

Deleting from a linked list

The procedure **LIST-DELETE** removes an element x from a linked list L. It must be given a pointer to x, and it then "splices" x out of the list by updating pointers. If we wish to delete an element with a given key, we must first call **LIST-SEARCH** to retrieve a pointer to the element.

```
LIST-DELETE(L, x)
1  if x.prev ≠ NIL
2      x.prev.next = x.next
3  else L.head = x.next
4  if x.next ≠ NIL
5      x.next.prev = x.prev
```

X is pointing to the node that needs to be deleted.

1. If x is not the first node, then last node to the node x will point to the next node.
2. If x is the first node in the linked list then head will be pointing to the next node.
3. If x is not the last node then pre pointer of the next node to x node will be pointing to the previous node to the x node.

Note:-

1. **O(1)** if pointer is given
2. **Θ(n)** if we wish to delete a node with given key.

Que 10.2-2 Implement a stack using a singly linked list L. The operations PUSH and POP should still take O(1) time.

Solution:
The element which is pushed at the end will be popped first. We insert element at front of the linked list hence we will pop element from the front of the linked list only.

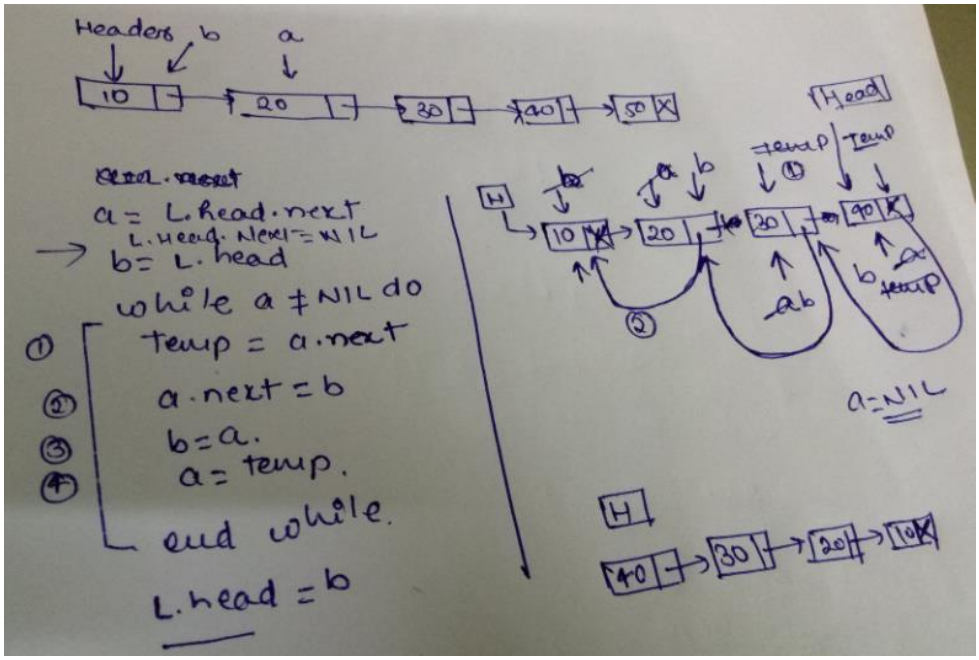
1. The **PUSH(L, x)** operation is exactly the same as **LIST-INSERT(L, x)**.
2. The POP operation sets x equal to L.head, calls **LIST-DELETE(L, L.head)**, then returns x.

Note - Insertion and deletion both are at the front of the linked list.

Que 10.2-3 Implement a queue by a singly linked list L. The operations ENQUEUE and DEQUEUE should still take O(1) time.
Solution:-
FIFO, hence we will perform **dequeue at the start** of Linked list, and will perform enqueue operation **at the end of linked list**.
Hence, In addition to the **head**, also keep **a pointer** to the last node of linked list.

1. To **enqueue**, insert the element after the last element of the list, and set it to be the new last element.
2. To **dequeue**, delete the first element of the list and return it.

Que 10.2-7 Give a **Θ(n)** time **non-recursive** procedure that reverses a singly linked list of n elements. The procedure should use no more than constant storage beyond that needed for the list itself.
Solution



Algorithm 9 REVERSE(L)

```
a = L.head.next
b = L.head
while a ≠ NIL do
    tmp = a.next
    a.next = b
    b = a
    a = tmp
end while
L.head = b
```

We are incrementing a and b to next nodes one by one.

Comparisons among lists

	unsorted, singly linked	sorted, singly linked	unsorted, doubly linked	sorted, doubly linked
`SEARCH(L, k)`	linear	linear	linear	linear
`INSERT(L, x)`	constant	linear	constant	linear
`DELETE(L, x)`	linear	linear	constant	constant
`SUCCESSOR(L, x)`	linear	constant	linear	constant
`PREDECESSOR(L, x)`	linear	linear	linear	constant
`MINIMUM(L, k)`	linear	constant	linear	constant
`MAXIMUM(L, k)`	linear	linear	linear	linear

- `MAXIMUM` assumes that we don't keep track of the tail of the list. If it does, we can make the algorithms linear when the list is sorted

Note: - x is the given pointer, pointing to the required node.

Explanation:

1. To search a given key, we have to traverse whole linked list, hence, O(n)
2. To insert into unsorted linked list, we make insertion at the head of linked list O(1)
3. To insert into sorted list, we have to find correct place
4. Deletion in single linked list, it's O(n) because if we are asked to delete last node we have to traverse the whole linked list, though we are given a pointer pointing to the last node.
5. Deletion can happen in O(1) in doubly linked list because last node has pointer to second last node.
6. Successor (L, x) in sorted list, next node will be successor, and we are given a pointer pointing to the node whose successor to be found.
7. Predecessor(L, x) is linear in single linked list because there is no pre pointer. In sorted doubly linked list the last node key will be predecessor.
8. Maximum will O(n), as we are not maintaining any tail pointer.

Hash Tables

Many applications require a dynamic set that supports only the dictionary operations INSERT, SEARCH, and DELETE.

A hash table is an effective data structure for implementing dictionaries.

Although searching for an element in a hash table can take as long as searching for an element in a linked list **Θ(n)** time in the worst case.

Note: - Under reasonable assumptions, average time to search for an element in a hash table is **O(1).**

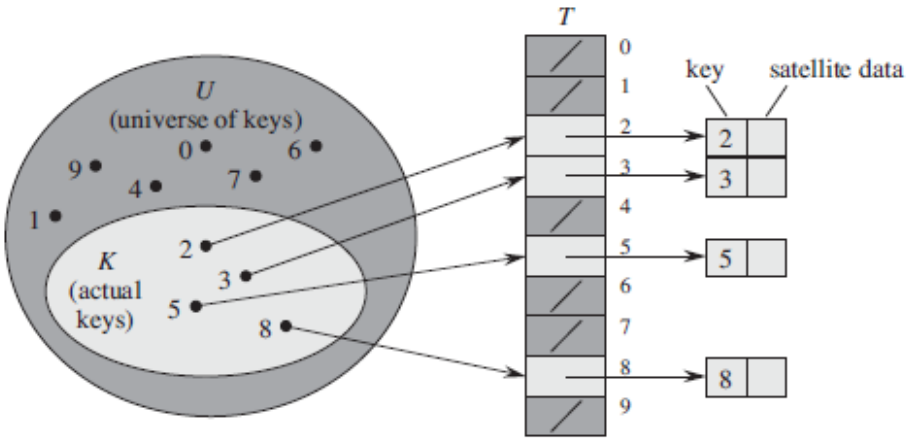
Properties:

1. Hashing is an extremely effective and practical technique: the basic dictionary operations require only **O(1) time on the average**.
2. "Perfect hashing" can support searches in **O(1) worst case time**, when the set of keys being stored is static (that is, when the set of keys never changes once stored).

Direct-address tables

Direct addressing is a simple technique that works well when the **universe U of keys is reasonably small**.
Suppose that an application needs a dynamic set in which each element has a key drawn from the universe U = {0,1, ….., m-1} where m is not too large. **We shall assume that no two elements have the same key**.

To represent the dynamic set, we use an **array**, or **direct-address table**, denoted by T[0, …., m-1], in which each position, or slot, corresponds to a key in the universe U.



Each key in the universe U = {0, 1, … , 9} corresponds to an index in the table. The set K = {2, 3, 5, 8} of actual keys determines the slots in the table that contain pointers to elements.

For some applications, **the direct-address table itself can hold the elements in the dynamic set**.

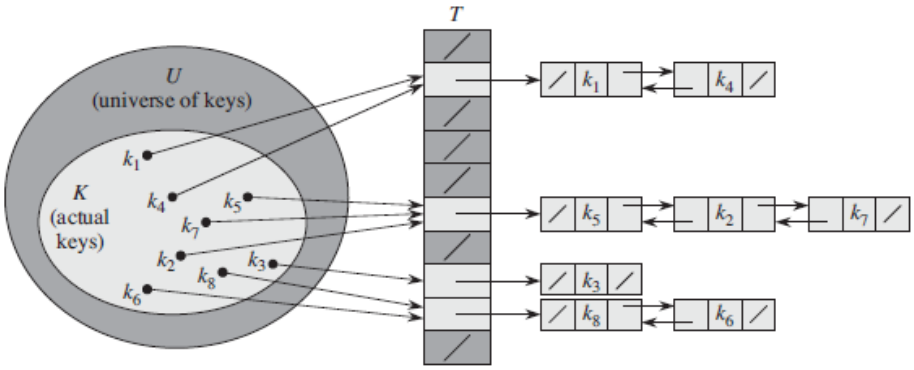
Hash tables

With direct addressing, an element with key k is stored in slot k. With hashing, this element is stored in slot **h(k);** that is, we use a hash function h to compute the slot from the key k.

Two keys may hash to the same slot. We call this situation a **collision**.

Collision resolution by chaining

In **chaining**, we place all the elements that hash to the same slot into the same linked list.



Collision Resolution by chaining

1. The worst-case running time for insertion is **O(1)**. The insertion procedure is fast in part because it assumes that the element x being inserted is not already present in the table.
2. For searching, the worst case running time is proportional to **the length of the list**.
3. We can delete an element in **O(1)** time if the lists are **doubly linked**, Note that **CHAINED-HASH-DELETE** takes as input an element x and not its key k, so that we don't have to search for x first.
4. With **singly linked lists**, both **deletion and searching** would have the same asymptotic running times.

Analysis of hashing with chaining

Given a hash table T with m slots that stores n elements:

Load factor α for T as n/m

That is, the average number of elements stored in a chain. Which can be less than, equal to, or greater than 1.

Note: - The worst-case behavior of hashing with chaining is **terrible**: all n keys hash to the same slot, creating a list of length n. The worst-case time for searching is thus **Θ(n)** plus the time to compute the hash function.

Theorem 11.1

In a hash table in which collisions are resolved by chaining, an unsuccessful search takes average-case time $\Theta(1+\alpha)$, under the assumption of simple uniform hashing.

Theorem 11.2

In a hash table in which collisions are resolved by chaining, a successful search takes average-case time $\Theta(1+\alpha)$, under the assumption of simple uniform hashing.

Universal hashing

Any fixed hash function is vulnerable to such **terrible** worst-case behavior; the only effective way to improve the situation is to choose the hash function randomly in a way that is independent of the keys that are actually going to be stored. This approach, called **universal hashing**.

Open addressing

In open addressing, **all elements occupy the hash table itself**. That is, each table entry contains either an element of the dynamic set or NIL. When searching for an element, we systematically examine table slots until either we find the desired element or we have ascertained that the element is not in the table. No elements are stored outside the table, unlike in chaining.

Linear probing: -

h(k, i) = (h(k) + i) mod m

Linear probing is easy to implement, but it suffers from a problem known as **primary clustering**. Long runs of occupied slots build up, **increasing the average search time**.

Quadratic probing

Quadratic probing uses a hash function of the form:

h(k, i) = (h(k) + c1*i + c2*i^2) mod m

Where c1 and c2 are positive auxiliary constants, and i = 0, 1, 2, ……., m-1.

This method works much better than linear probing. This property leads to a milder form of clustering, called **secondary clustering**. If two keys have the same initial probe position, then their probe sequences are the same.

Double hashing

Double hashing offers one of the best methods available for open addressing. **Double hashing** uses a hash function of the form:

h(k, i) = (h1(k) + i*h2(k)) mod m

Where both h1 and h2 are auxiliary hash functions.

Analysis of open-address hashing

Theorem 11.6

Given an open-address hash table with load factor $\alpha = n/m < 1$, the expected number of probes in an unsuccessful search is at most $1/(1-\alpha)$, assuming uniform hashing.

Corollary 11.7

Inserting an element into an open-address hash table with load factor α requires at most $1/(1-\alpha)$ probes on average, assuming uniform hashing.

Theorem 11.8

Given an open-address hash table with load factor $\alpha < 1$, the expected number of probes in a successful search is at most

$$\frac{1}{\alpha} \ln \frac{1}{1-\alpha},$$

Perfect hashing

Hashing technique **perfect hashing** if **O(1)** memory accesses are required to perform a search in the worst case.

Binary Search Trees

The search tree data structure supports many dynamic-set operations, including SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, and DELETE.

1. Basic operations on a binary search tree take time proportional to the height of the tree.
2. For a complete binary tree with n nodes, such operations run in **Θ(n)** worst-case time.
3. If the tree is a linear chain of n nodes, the same operations take **Θ(n)** **worst-case time**.
4. **Expected height** of a randomly built binary search tree is **O(logn)**. So that basic dynamic-set Operations on such a tree take **Θ(logn)** time on average.

The keys in a **binary search tree** are always stored in such a way as to satisfy the binary-search-tree property.

Let x be a node in a binary search tree. If y is a node in the left subtree of x, then **y.key <= x.key**. If y is a node in the right subtree of x, then **y.key >= x.key**.

1. An **inorder tree walk** print out all the keys in a binary search tree in sorted order.
2. A **preorder tree walk** prints the root before the values in either subtree.
3. A **postorder tree walk** prints the root after the values in its subtrees.

Inorder Tree Walk

To use the following procedure to print all the elements in a binary search tree T, we call **INORDER-TREE-WALK(T.root)**. It takes **Θ(n).**

```
INORDER-TREE-WALK(x)
1  if x ≠ NIL
2      INORDER-TREE-WALK(x.left)
3      print x.key
4      INORDER-TREE-WALK(x.right)
```

It takes Θ(n) time to walk an n-node binary search tree, since after the initial call, the procedure calls itself recursively exactly twice for each node in the tree—once for its left child and once for its right child.

For n>0, suppose that INORDER-TREE-WALK is called on a node x whose left subtree has k nodes and right subtree has n-k-1 nodes, time to perform INORDER-TREE-WALK(x) is

T(n) = T(k) + T(n-k+1) + c

```
Algorithm 1 PREORDER-TREE-WALK(x)
  if x ≠ NIL then
     print x
     PREORDER-TREE-WALK(x.left)
     PREORDER-TREE-WALK(x.right)
  end if
  return
```

```
Algorithm 2 POSTORDER-TREE-WALK(x)
  if x ≠ NIL then
     POSTORDER-TREE-WALK(x.left)
     POSTORDER-TREE-WALK(x.right)
     print x
  end if
  return
```

Querying a binary search tree

1. Searching O(h)

Given a pointer to the root of the tree and a key k, TREE-SEARCH returns a pointer to a node with key k if one exists; otherwise, it returns NIL.

```
TREE-SEARCH(x, k)
1  if x == NIL or k == x.key
2      return x
3  if k < x.key
4      return TREE-SEARCH(x.left, k)
5  else return TREE-SEARCH(x.right, k)
```

The running time of TREE-SEARCH is O(h), where h is the height of the tree.

```
ITERATIVE-TREE-SEARCH(x, k)
1  while x ≠ NIL and k ≠ x.key
2      if k < x.key
3          x = x.left
4      else x = x.right
5  return x
```

The iterative version is more efficient.

Minimum and maximum O(h)

We can always find an element in a binary search tree whose key is a **minimum** by following left child pointers from the root until we encounter a NIL.

The following procedure returns a pointer to the minimum element in the subtree rooted at a given node x.

```
TREE-MINIMUM(x)
1  while x.left ≠ NIL
2      x = x.left
3  return x
```

If a node x has no left subtree, then since every key in the right subtree of x is at least as Large as x.key, the minimum key in the subtree rooted at x is x.key. Following procedure is to find the maximum element in a B.S.T

```
TREE-MAXIMUM(x)
1  while x.right ≠ NIL
2      x = x.right
3  return x
```

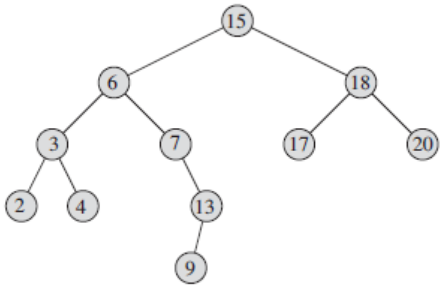


Figure 12.2 Queries on a binary search tree. To search for the key 13 in the tree, we follow the path $15 \to 6 \to 7 \to 13$ from the root. The minimum key in the tree is 2, which is found by following *left* pointers from the root. The maximum key 20 is found by following *right* pointers from the root. The successor of the node with key 15 is the node with key 17, since it is the minimum key in the right subtree of 15. The node with key 13 has no right subtree, and thus its successor is its lowest ancestor whose left child is also an ancestor. In this case, the node with key 15 is its successor.

Successor and predecessor O(h)

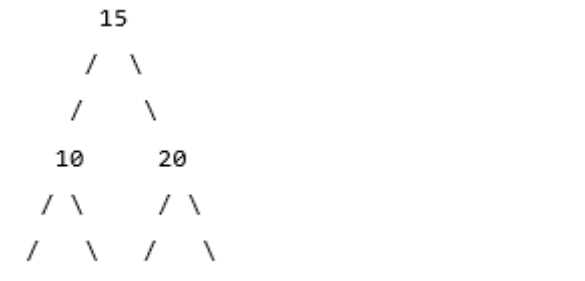
Successor of a node x is the node with the **smallest key greater than x.key**.

```
TREE-SUCCESSOR(x)
1  if x.right ≠ NIL
2       return TREE-MINIMUM(x.right)
3  y = x.p
4  while y ≠ NIL and x == y.right
5       x = y
6       y = y.p
7  return y
```

1. A node has right subtree, then find minimum element from RST.
2. If a node has not a right sub tree, then climb up. Successor of such node will be its lowest ancestor **whose left child is also an ancestor.**

```
Algorithm 5 TREE-PREDECESSOR(x)
  if x.left ≠ NIL then
     return TREE-MAXIMUM(x.left)
  end if
  y = x.p
  while y ≠ NIL and x == y.left do
     x = y
     y = y.p
  end while
  return y
```

1. If a node has left subtree, find the maximum element in LST will be inorder predecessor.
2. If left sub tree doesn't exist then in-order predecessor is one of the ancestor of it. We can move up towards root until we encounter a node which is right child of its parent.

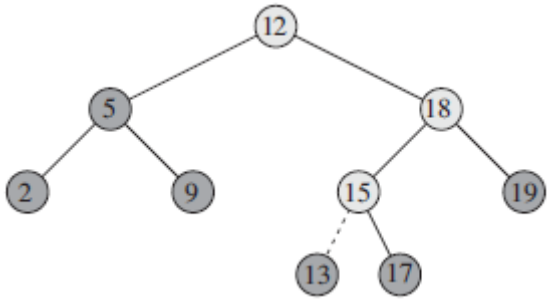



```
in-order predecessor of 8 do not exist.
in-order predecessor of 10 is 8
in-order predecessor of 12 is 10
in-order predecessor of 20 is 15
```

Insertion and deletion O(h)

The procedure takes a node z for which z.key=v, z.left=NIL, and z.right=NIL It modifies T and some of the attributes of z in such a way that it inserts z into an appropriate position in the tree.

```
TREE-INSERT(T, z)
 1  y = NIL
 2  x = T.root
 3  while x ≠ NIL
 4        y = x
 5        if z.key < x.key
 6              x = x.left
 7        else x = x.right
 8  z.p = y
 9  if y == NIL
10        T.root = z          // tree T was empty
11  elseif z.key < y.key
12        y.left = z
13  else y.right = z
```



Inserting an item with key 13 into a binary search tre

Deletion

The overall strategy for deleting a node z from a binary search tree T has three basic cases:

1. If **z has no children**, then we simply remove it by modifying its parent to replace z with NIL as its child.
2. If z has just one child, then we elevate that child to take z's position in the tree by modifying z's parent to replace z by z's child.
3. If z has two children, then we find z's successor y, which must be in z's right subtree and have y take z's position in the tree. The rest of z's original right subtree becomes y's new right subtree, and z's left subtree becomes y's new left subtree.

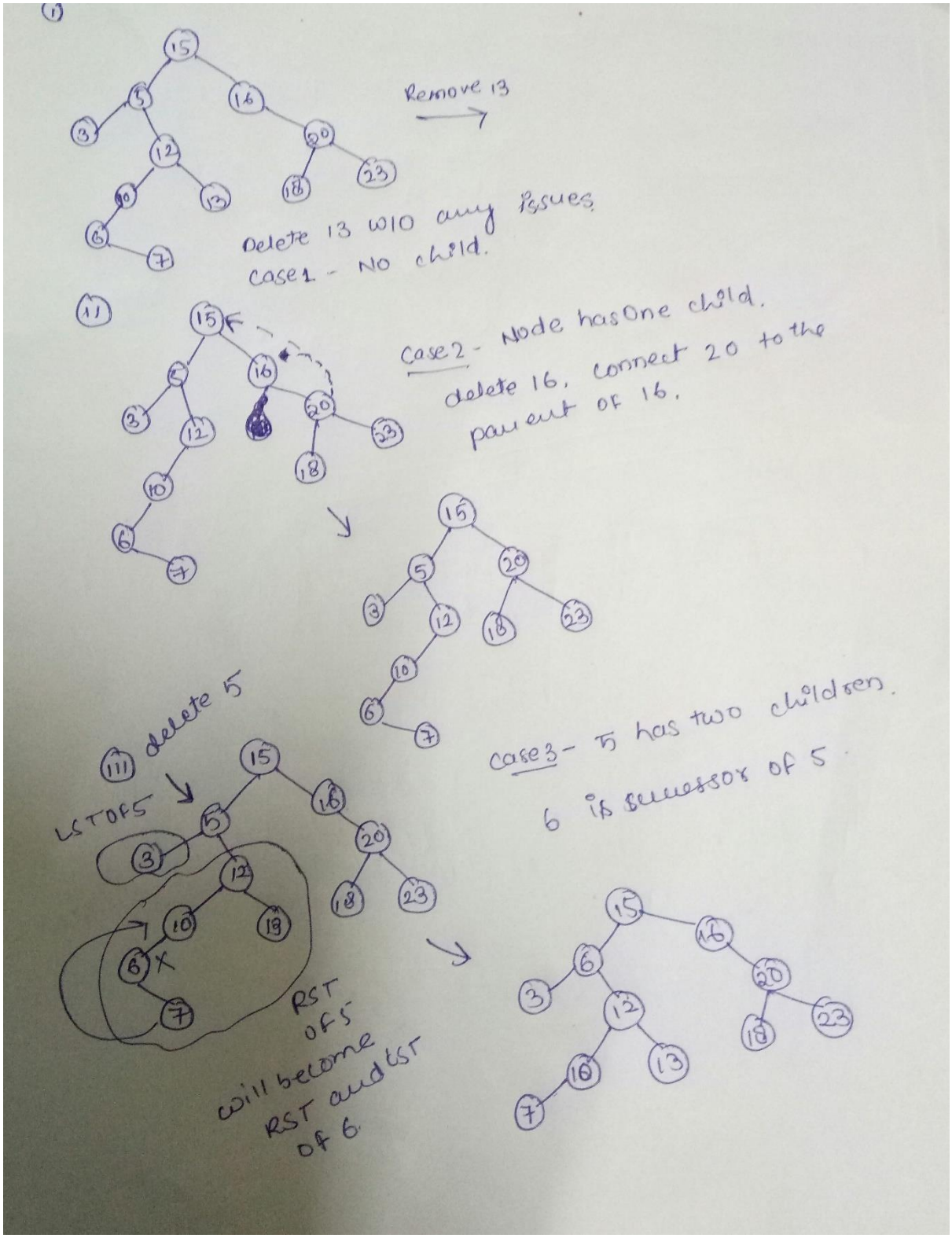
Theorem 12.4 the expected height of a randomly built binary search tree on n distinct keys is O(logn).

Number of Binary trees possible with n nodes

no. of BST = no. of unlabelled B.T = 2nCn/(n+1)

no. of labelled trees = (no. of unlabelled trees)*n!

T(n) = [2nCn/(n+1)]*n!



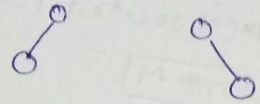
Deletion in BST

⇒ Number of Binary Trees possible with n nodes -

i) How many different unballed Binary Trees can be there with n nodes?

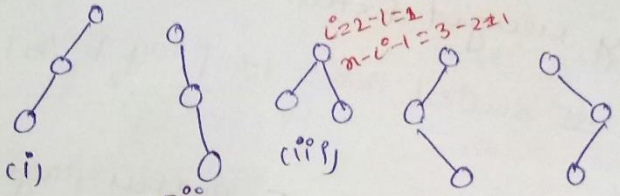
if n=1, there is only one tree

for n=2 there are two trees -



] different structures

for n=3, there are five trees -



for example, let T(n) be count for n nodes -

T(0) = 1 [There is only 1 empty Tree]

T(1) = 1

T(2) = 2

$T(3) = T(0)*T(2) + T(1)*T(1) + T(2)*T(0)$

$= 1*2 + 1*1 + 2*1 = 5.$

$T(4) = T(0)*T(3) + T(1)*T(2) + T(2)*T(1) + T(3)*T(0)$

$= 1X5 + 1X2 + 2X1 + 5X1 = 14.$

$$T(n) = \sum_{i=1}^{n} T(i-1)T(n-i) = \sum_{i=0}^{n-1} T(i)*T(n-i-1) = C_n$$

T(i-1) represents number of nodes on the left subtree

T(n-i) = represents no. of nodes in RST

Minimum Spanning Trees

Spanning Trees

A spanning tree of a connected and undirect graph on n vertices is a subset of n-1 edges that form a tree.

Number of spanning trees in Graphs:

1. **Complete Graph Kn = n*(n^-2)**
2. **Cycle Graph Cn= n**
3. **If graph is already a tree = 1**
4. **Complete Bipartite Graph Km,n =** $t(K_{n,m}) = m^{n-1}n^{m-1}$

Kirchoff's theorem:

If G(V,E) is a graph on n vertices with V = {v1,...,vn} then its **graph Laplacian** L is an n ⇥ n matrix whose entries are

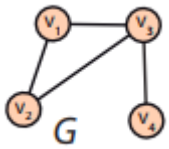
$$L_{ij} = \begin{cases} \deg(v_j) & If\ i = j \\ -1 & If\ i \neq j\ and\ (v_i, v_j) \in E \\ 0 & Otherwise \end{cases}$$

The number NT of spanning trees contained in G is given by the following computation:

1. Choose a vertex vj and eliminate the j-th row and column from L to get a new matrix L'.
2. Determinant of new matrix L' will be equal to the number of spanning trees.

Nt = Det(L')

Example-



Solution:

$$L = D - A$$

$$= \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} - \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

$$= \begin{bmatrix} 2 & -1 & -1 & 0 \\ -1 & 2 & -1 & 0 \\ -1 & -1 & 3 & -1 \\ 0 & 0 & -1 & 1 \end{bmatrix}$$

if we choose vj = v1 then remove 1st row and col

$$\hat{L}_1 = \begin{bmatrix} 2 & -1 & 0 \\ -1 & 3 & -1 \\ 0 & -1 & 1 \end{bmatrix}$$

determinant = 2(3x1 – 1) +1(-1 -0) = 4 -1 =3 spanning tress.

Rooted Spanning Tree:
A rooted spanning tree of a directed graph is rooted tree containing edges of the graph such that every vertex of the graph is an endpoint of one of the edges in tree.
Note:-
In a connected directed graph G in which each vertex has same **in-degree** and **out-degree** has a rooted spanning tree.

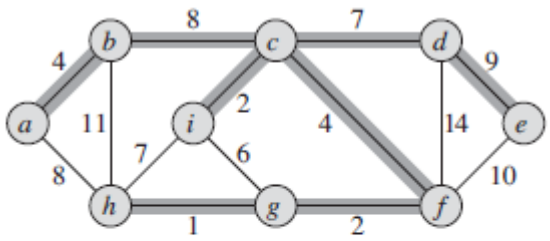


Figure 23.1 A minimum spanning tree for a connected graph. The weights on edges are shown, and the edges in a minimum spanning tree are shaded. The total weight of the tree shown is 37. This minimum spanning tree is not unique: removing the edge (b, c) and replacing it with the edge (a, h) yields another spanning tree with weight 37.

Growing a minimum spanning tree

Assume that we have a connected, undirected graph G = (V, E) with a weight function w: E →R and we wish to find a minimum spanning tree for G.

Properties:-
1. Let (u, v) be a **minimum-weight edge** in a connected graph G, then (u, v) belongs to **some minimum spanning tree of G**.
2. If an edge (u, v) is contained in some minimum spanning tree, then it is a **light edge crossing some cut of the graph**.
3. Let e be a **maximum-weight edge** on some cycle of connected graph G = (V, E), There is a minimum spanning tree that doesn't include e.
4. A graph has a **unique minimum** spanning tree if, for every cut of the graph, there **is a unique light edge crossing the cut**.
5. If all edge weights of a graph are **positive**, then any subset of edges that connects all vertices and has minimum total weight must be a minimum spanning tree.
6. Let T be a minimum spanning tree of a graph G, and let L be the sorted list of the edge weights of T. For any other minimum spanning tree T' of G, the list L is also the sorted list of edge weights of T'.
7. Given a graph G and a minimum spanning tree T , suppose that we decrease the weight of **one of the edges in T** by some positive integer k, it will still remain M.S.T for G.

Kruskal's algorithm (Greedy Algorithm) O(Elogv)

In **Kruskal's algorithm**, the set A is a **forest** whose vertices are all those of the given graph. The safe edge added to A is always a least-weight edge in the graph that **connects two distinct components**.

It uses a disjoint-set data structure to maintain several disjoint sets of elements. Each set contains the vertices in one tree of the current forest.

We can determine whether two vertices u and v belong to the same tree by testing whether **FIND-SET(u) equals FIND-SET(v)**. To combine trees, Kruskal's algorithm calls the **UNION** procedure.

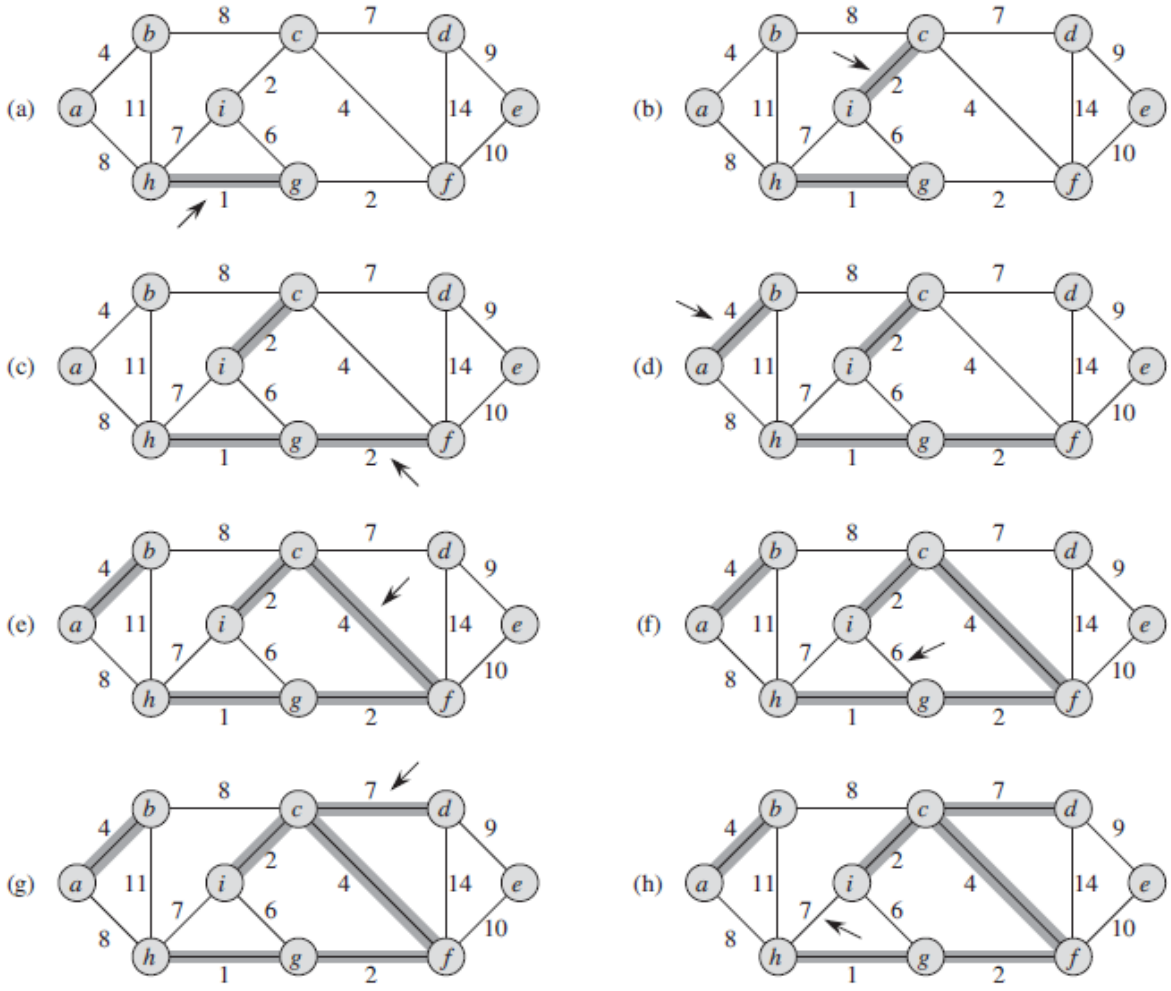
```
MST-KRUSKAL(G, w)
1  A = ∅
2  for each vertex v ∈ G.V
3       MAKE-SET(v)
4  sort the edges of G.E into nondecreasing order by weight w
5  for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
6       if FIND-SET(u) ≠ FIND-SET(v)
7            A = A ∪ {(u, v)}
8            UNION(u, v)
9  return A
```

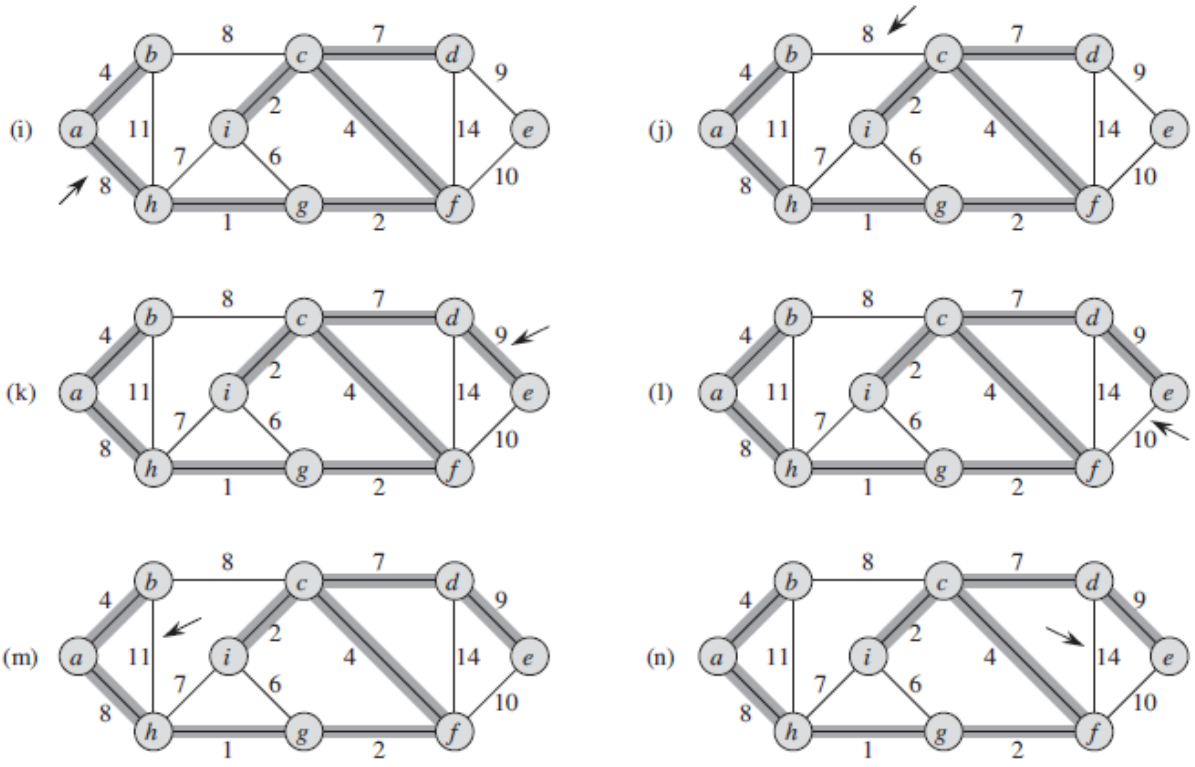
1. Lines 1-3 Initialize the set A the empty set and create |V| trees, one containing each vertex.
2. Line 4, edges are sorted in ascending order **O(ElogE)**
3. The for loop in lines 5–8 examines edges in order of weight, from lowest to highest. The loop checks, for each edge (u, v), whether the endpoints u and v belong to the same tree. If they do, then the edge (u, v) cannot be added to the forest without creating a cycle, and the edge is discarded.
4. If u and v don't belong to same set that edge (u, v) will be added to A.
5. Line 8 merges the vertices in the two trees.

Properties:-

1. **Kruskal's algorithm** can return different spanning trees for the same input graph G, depending on how it breaks ties when the edges are sorted into order. When graph has duplicate weight edges.
2.
   - Therefore, $O(E \lg V)$ time. (If edges are already sorted, $O(E\,\alpha(V))$, which is almost linear.)

Figure 23.4 The execution of Kruskal's algorithm on the graph from Figure 23.1. Shaded edges belong to the forest A being grown. The algorithm considers each edge in sorted order by weight. An arrow points to the edge under consideration at each step of the algorithm. If the edge joins two distinct trees in the forest, it is added to the forest, thereby merging the two trees.





Time Complexity Analysis of Kruskal's algorithm:

The running time of Kruskal's algorithm for a graph G(V, E) depends on how we implement the **disjoint-set data structure**.

1. Initializing the set A in line 1 takes O(1) time
2. The time to sort the edges in line 4 is O(ElogE).
3. Line 2-3, there are V, MAKE-SET operations.
4. The for loop of lines 5–8 performs O(E) FIND-SET and UNION operations on the disjoint-set forest.
5. These, line 2-3 and 5-8, take a total of $O((V + E)\,\alpha(V))$ time.
6. Moreover, since $\alpha(|V|) = O(\lg V) = O(\lg E)$,
7. Total running time of kruskal's algorithm is **O(ElogE)**, As $|E| < |V|^2$, we have logE = Logv

Running time of Kruskal's algorithm is O(ElogV)

Note - A is Minimum Spanning Tree for graph G

Prim's algorithm (Greedy Algorithm) O(Elogv)

Prim's algorithm operates much like Dijkstra's algorithm for finding shortest paths in a graph**. Prim's algorithm has the property that the edges in the set A always form a single tree**. The tree starts from an arbitrary root vertex r and grows until the tree spans all the vertices in V. Each step adds to the tree A, a light edge that connects A to an isolated vertex—one on which no edge of A is incident.

During execution of the algorithm, all vertices that are not in the tree reside in a **min-priority queue Q** based on a key attribute.
For each vertex v, the attribute **v.key** is **the minimum weight of any edge connecting v** to a vertex in the tree;

1. by convention, $v.key = \infty$ if there is no such edge.
2. attribute $v.\pi$ names the parent of v in the tree.

$$A = \{(v, v.\pi) : v \in V - \{r\}\} .$$

```
MST-PRIM(G, w, r)
 1  for each u ∈ G.V
 2      u.key = ∞
 3      u.π = NIL
 4  r.key = 0
 5  Q = G.V
 6  while Q ≠ ∅
 7      u = EXTRACT-MIN(Q)
 8      for each v ∈ G.Adj[u]
 9          if v ∈ Q and w(u, v) < v.key
10              v.π = u
11              v.key = w(u, v)
```

1. Lines 1–5 set the key of each vertex to ∞ (except for the root r, whose key is set to 0 so that it will be the first vertex processed), set the parent of each vertex to NIL, and initialize the min-priority queue Q to contain all the vertices.
2. Line 6, until Q is empty
3. Line 7, extract minimum from min-heap(priority queue)
4. The for loop of lines 8–11 updates the key and π attributes of every **vertex v adjacent to u but not in the tree**.

The running time of Prim's algorithm depends on how we implement the **min-priority queue** Q.
If we implement Q as a binary min-heap

1. We can use the **BUILD-MIN-HEAP** procedure to perform lines 1–5 in **O(V)** time.
2. The body of the while loop executes |V| times.
3. Since each **EXTRACT-MIN** operation takes **O(logv)** time, the total time for all calls to EXTRACT-MIN is **O(vlogv)**.
4. The for loop in lines 8–11 executes **O(E)** times altogether, since the sum of the lengths of all adjacency lists is 2|E|.
5. Within the for loop, we can implement the test for membership in Q in line **9 in constant time by keeping** a bit for each vertex that tells whether or not it is in Q, and updating the bit when the vertex is removed from Q.
6. The assignment in line 11 involves an implicit **DECREASE-KEY** operation on the min-heap, which a binary min-heap supports in **O(log V)** time.

Total Time Complexity: O(vlogv + Elogv) = o(Elogv), same as of Kruskal's algorithm.

Improvements: -

1. If we use **Fibonacci heap** to hold |V| elements, **EXTRACT_MIN** takes O(logv) but **DECREASE_KEAY** takes O(1) hence T.C will be **O(vlogv + E) = O(E)**
2. If we use **Binomial heap** to hold |V| elements, **EXTRACT_MIN** and **DECREASE_KEY** take **O(logn)** hence T.C will be O(V + E)logv

Prim's Algorithm with Adjacency Matrix is O(V^2).

Que 23.2-4 Suppose that all edge weights in a graph are integers in the range from **1 to |V |**. How fast can you make **Kruskal's algorithm** run? What if the edge weights are integers in the range from 1 to W for **some constant W**?

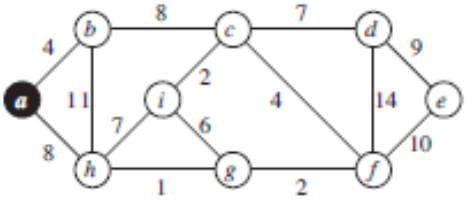
Solution:

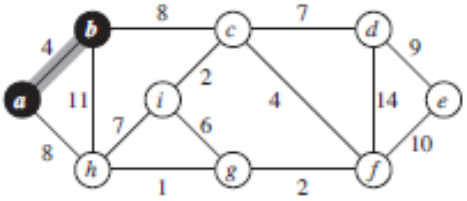
In general total running time of $O(V + E \lg E + E\,\alpha(V)) = O(E \lg E)$. If we knew that all of the edge weights in the graph were integers in the range from 1 to |V|, then we could sort the edges in **O(V + E)** time using **counting sort**. There are E edges and range of integer is up to V. Since the graph is connected, **V = O(E)**, and so **the sorting time is reduced to O(E)**. This would yield a total running time of $O(V + E + E\,\alpha(V))$ =

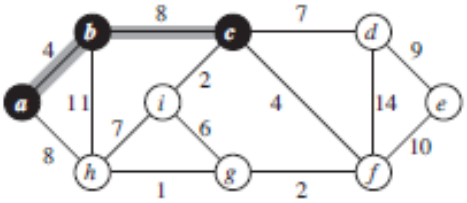
$$O(E\,\alpha(V)), \text{ again since } V = O(E), \text{ and since } E = O(E\,\alpha(V))$$

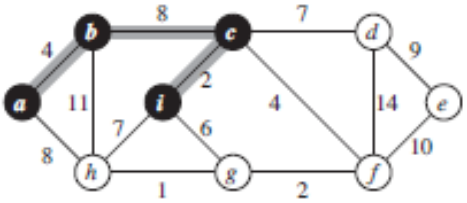
If the edge weights were integers in the range from 1 to W for some constant W. then we can use **counting sort O(E + W) = O(E)** since w is a constant. Same as first part

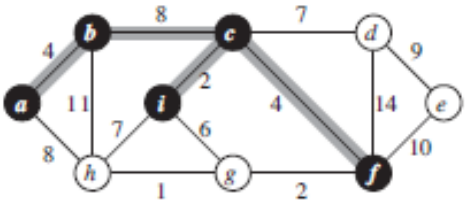
we get a total running time of $O(E\,\alpha(V))$.

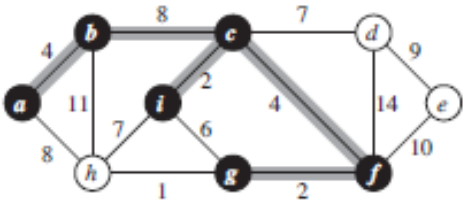
(a)


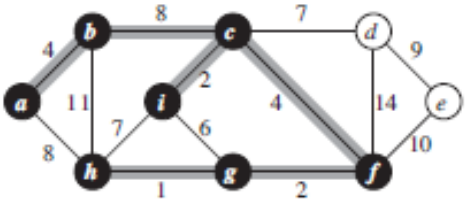
(b)


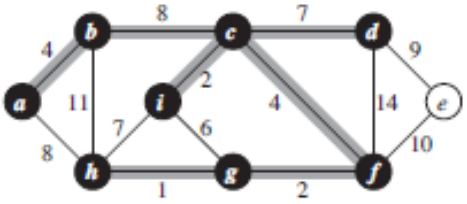
(c)


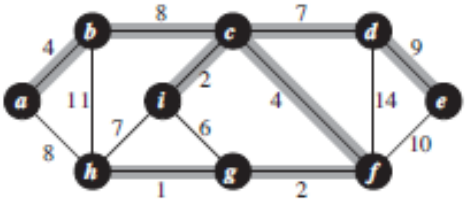
(d)


(e)


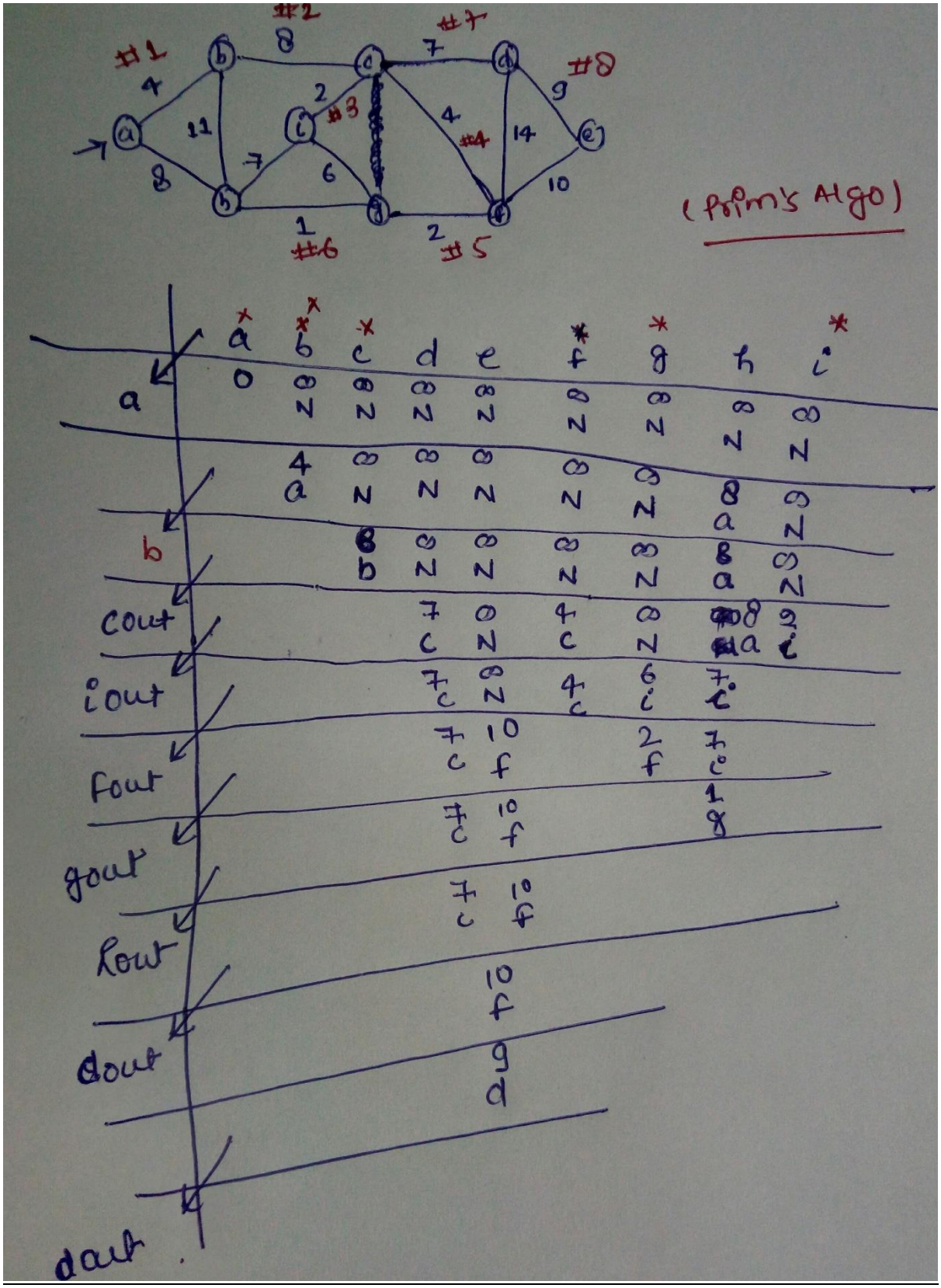
(f)


(g)


(h)


(i)


Prim's Algorithm



Prim's Algorithm

Que 23.2-5

Suppose that all edge weights in a graph are integers in the range from 1 to |V|. How fast can you make **Prim's algorithm run**? What if the edge weights are integers in the range from 1 to W for **some constant W**?

Solution

The time taken by Prim's algorithm is determined by the speed of the **queue operations** (Decrease_Key and Extract_Min). With the queue implemented as a **Fibonacci heap**, it takes **O(logv + E)** time.

We can improve the running time of Prim's algorithm if W is a constant by implementing the queue **as an array Q[0 . .W + 1]** (using the W + 1 slot for key = ∞)

Then **EXTRACT-MIN takes only O(W) = O(1)** and **DECREASE-KEY takes only O(1) time**. **This gives a total running time of O(E).**

However, if the range of edge weights is 1 to |V|, then EXTRACT-MIN takes $\Theta(V)$ **with this data structure**. So the total time spent doing EXTRACT-MIN is $\Theta(V^2)$, slowing the algorithm to $\Theta(E + V^2) = \Theta(V^2)$. In this case, it is better to keep the Fibonacci-heap priority queue, which gave the $\Theta(E + V \lg V)$ time.

Que 23.2-7

Suppose that **a graph G has a minimum spanning tree already computed**. How quickly can we **update the minimum spanning tree if we add a new vertex and incident edges to G**?

Solution:

We can compute a minimum spanning tree of G in **O(VlogV)** time using Prim's algorithm with Fibonacci-heap.

Single-Source Shortest Paths

In a **shortest-paths problem (Greedy Algorithm)**, we are given a weighted, directed graph **G(V, E),** with weight function **w: E →R** mapping edges to real-valued weights. The **weight w(p) of path p** = {v0, v1, …… vk} is the sum of the weights of its constituent edges.

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

We define the **shortest-path weight** $\delta(u, v)$ from u to v by

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \overset{p}{\rightsquigarrow} v\} & \text{if there is a path from } u \text{ to } v\,, \\ \infty & \text{otherwise}\,. \end{cases}$$

A **shortest path from vertex u to vertex v** is then defined as any path p with weight $w(p) = \delta(u, v)$

1. **Single-source shortest-paths problem:** given a graph **G(V, E),** we want to find a shortest path **from a given source vertex** $s \in V$ **to each vertex** $v \in V$.

2. **Single-destination shortest-paths problem:** find a shortest path to a given destination vertex t from each vertex v . By reversing the direction of each edge in the graph, we can reduce this problem to a single-source problem.

3. **Single-pair shortest-path problem**: Find a shortest path from u to v for given vertices u and v. If we solve the single-source problem with source vertex u, we solve this problem also.

4. **All-pairs shortest-paths problem:** find a shortest path from u to v for every pair of vertices u and v. Although we can solve this problem by running a single source algorithm once from each vertex, we usually can solve it faster**.**

Optimal substructure of a shortest path: a shortest path between two vertices contains other shortest paths within it. **Sub-paths of shortest paths are shortest paths**.

Negative-weight edges If the graph G(V, E) contains **no negative weight cycles reachable from the source s**, then for all $v \in V$, the shortest-path weight $\delta(s, v)$ remains well defined, even if it has a negative value.
If the graph contains a negative-weight cycle reachable from s, however, shortest-path weights are not well defined. **No path from s to a vertex on the cycle can be a shortest path,** we can always find a path with lower weight by following the proposed "shortest" path and then traversing the negative-weight cycle. If there is a negative-weight cycle on some path from s to v,
we define $\delta(s, v) = -\infty$

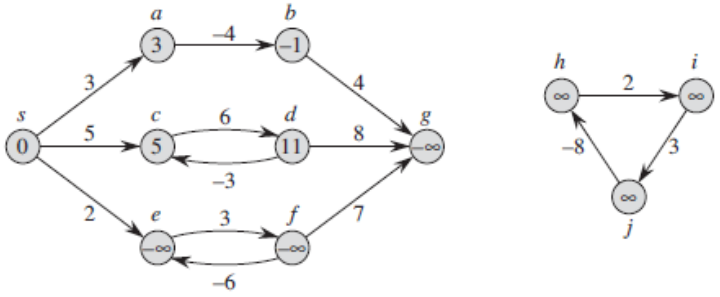


Figure 24.1 illustrates the effect of negative weights and negative-weight cycles on shortest-path weights. Because there is only one path from s to a (the path $\langle s, a\rangle$), we have $\delta(s, a) = w(s, a) = 3$. Similarly, there is only one path from s to b, and so $\delta(s, b) = w(s, a) + w(a, b) = 3 + (-4) = -1$. There are infinitely many paths from s to c: $\langle s, c\rangle$, $\langle s, c, d, c\rangle$, $\langle s, c, d, c, d, c\rangle$, and so on. Because the cycle $\langle c, d, c\rangle$ has weight $6 + (-3) = 3 > 0$, the shortest path from s to c is $\langle s, c\rangle$, with weight $\delta(s, c) = w(s, c) = 5$. Similarly, the shortest path from s to d is $\langle s, c, d\rangle$, with weight $\delta(s, d) = w(s, c) + w(c, d) = 11$. Analogously, there are infinitely many paths from s to e: $\langle s, e\rangle$, $\langle s, e, f, e\rangle$, $\langle s, e, f, e, f, e\rangle$, and so on. Because the cycle $\langle e, f, e\rangle$ has weight $3 + (-6) = -3 < 0$, however, there is no shortest path from s to e. By traversing the negative-weight cycle $\langle e, f, e\rangle$ arbitrarily many times, we can find paths from s to e with arbitrarily large negative weights, and so $\delta(s, e) = -\infty$. Similarly, $\delta(s, f) = -\infty$. Because g is reachable from f, we can also find paths with arbitrarily large negative weights from s to g, and so $\delta(s, g) = -\infty$. Vertices h, i, and j also form a negative-weight cycle. They are not reachable from s, however, and so $\delta(s, h) = \delta(s, i) = \delta(s, j) = \infty$.

Note:-

1. Dijkstra's algorithm, assume that all edge weights in the input graph are nonnegative.
2. Bellman-Ford algorithm, allow negative-weight edges in the input graph and produce a correct answer as long as **no negative-weight cycles are reachable from the source**. If there is such a negative-weight cycle, the algorithm can detect and report its existence**.**

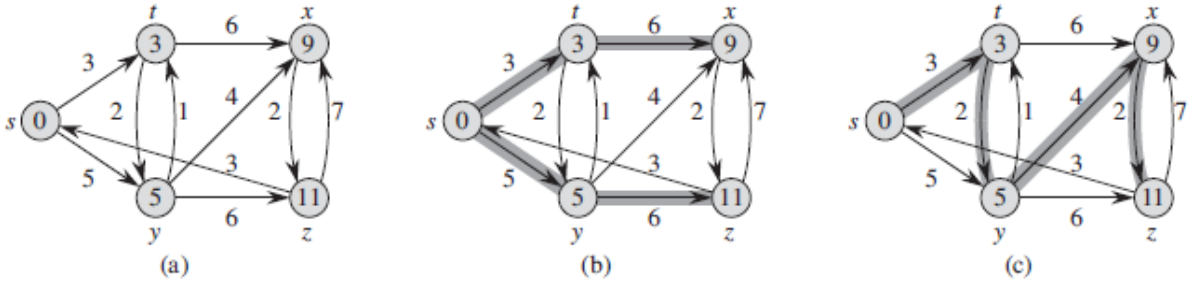


Figure 24.2 (a) A weighted, directed graph with shortest-path weights from source s. (b) The shaded edges form a shortest-paths tree rooted at the source s. (c) Another shortest-paths tree with the same root.

Relaxation

For each vertex $v \in V$, we maintain an attribute v.d, which is an upper bound on the weight of a shortest path from source s to v. We call v.d a **shortest-path estimate**.

We initialize the shortest-path estimates and predecessors by the following **Θ(V)** time procedure.

```
INITIALIZE-SINGLE-SOURCE(G, s)
1  for each vertex v ∈ G.V
2      v.d = ∞
3      v.π = NIL
4  s.d = 0
```

After initialization, we have $v.\pi = \text{NIL}$ for all $v \in V$, $s.d = 0$, and $v.d = \infty$ for $v \in V - \{s\}$.

The process of **relaxing** an edge (u, v) consists of testing whether we can improve the shortest path to v found so far by going through u and, if so updating $v.d$ and $v.\pi$

Following code performs a **relaxation** step on edge (u, v) in **O(1)** time:

```
RELAX(u, v, w)
1  if v.d > u.d + w(u, v)
2      v.d = u.d + w(u, v)
3      v.π = u
```

Dijkstra's algorithm

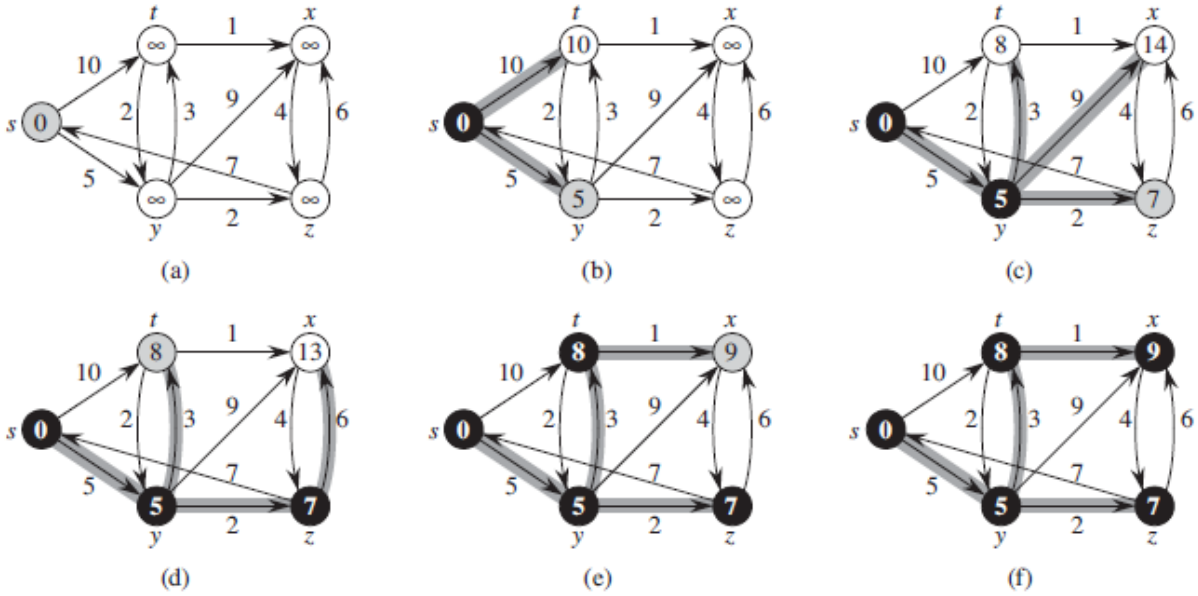
Dijkstra's algorithm solves the single-source shortest-paths problem on a weighted, directed graph G(V, E) for the case in which all edge weights are nonnegative. We assume that $w(u, v) \geq 0$ for each edge $(u, v) \in E$.

Dijkstra's algorithm maintains **a set S of vertices whose final shortest-path weights from the source s have already been determined**. The algorithm repeatedly selects the vertex $u \in V - S$ with the minimum shortest-path estimate, adds u to S, and relaxes all edges leaving u. We use a min-priority **queue Q of vertices**, keyed by their d values.

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```

1. Line 1 initializes the d and π values in the usual way.
2. Line 2 initializes the set S to the empty set.
3. Line 3 initializes the min-priority queue Q to contain all the vertices in V.
4. While loop of lines 4–8, line 5 extracts a vertex u from **Q = V- S** and line 6 adds it to set S.
5. Lines 7–8 relax each edge (u, v) leaving u, thus updating the estimate v.d and the predecessor $v.\pi$ if we can improve the shortest path to v found so far by going through u.

While loop of lines 4–8 iterates exactly |V| times. The running time of Dijkstra's algorithm depends on how we implement the **min-priority queue**. Since the total number of edges in all the adjacency lists is |E|, 7-8 lines for loop iterates a total of |E| times, and thus the algorithm calls DECREASE-KEY at most |E| times overall.



(a) The situation just before the first iteration of the while loop of lines 4–8. The shaded vertex has the minimum d value and is chosen as vertex u in line 5. (b)–(f) The situation after each successive iteration of the while loop. The shaded vertex in each part is chosen as vertex u in line 5 of the next iteration. The d values and predecessors shown in part (f) are the final values.

1. Consider first the case in which we maintain the min-priority queue by taking advantage of the vertices being numbered 1 to |V|. **We simply store v.d in the vth entry of an array**.

a. Each INSERT and DECREASE-KEY operation takes O(1) time.

b. Each EXTRACT-MIN operation takes O(V) time (since we have to search the entire array).

c. Hence total time will be $O(V^2 + E) = O(V^2)$

2. If we us Binary Heap to maintain min priority queue.
   a. Each EXTRACT-MIN operation then takes time O(logv) As before, there are |V| such operations. The time to build the binary min-heap is O(V).
   b. Each DECREASE-KEY operation takes time O(logV), and there are still at most |E| such operations.
   c. The total running time is therefore O(vlogv + Elogv) which is **O(Elogv)** if all vertices are reachable from the source.

3. If we use Fibonacci Heap, EXTRACT_MIN operation takes O(logv) and DECREASE_KEY operation takes O(1), total TC will be **O(vlogv +E)**

The Bellman-Ford algorithm O(VE)

The **Bellman-Ford algorithm** solves the single-source shortest-paths problem in the general case in which edge weights may be negative.
Given a weighted, directed graph G(V, E) with source s and weight function w: E→ R, the Bellman-Ford algorithm returns a Boolean value indicating whether or not there is a negative-weight cycle that is reachable from the source. If there is such a cycle, the algorithm indicates that no solution exists. If there is no such cycle, the algorithm produces the shortest paths and their weights.

The algorithm relaxes edges, progressively decreasing an estimate v.d on the weight of a shortest path from the **source s to each vertex** $v \in V$ until it achieves the actual shortest-path weight $\delta(s, v)$.

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

1. Line 1 initializes the d and π for all vertices.
2. Line 2, for loop iterates |V| - 1 times. Each pass is one iteration of the for loop of lines 2–4 and consists of **relaxing each edge of the graph once**.
3. After making |V| - 1 passes, lines 5–8 check for a negative-weight cycle and return the appropriate Boolean value.

The Bellman-Ford algorithm runs in the O(VE), since the initialization in line 1 takes **Θ(V)** time, each of the |V|- 1 passes over the edges in lines 2–4 takes **Θ**(E) time, and the for loop of lines 5–7 takes **O(E)** time.

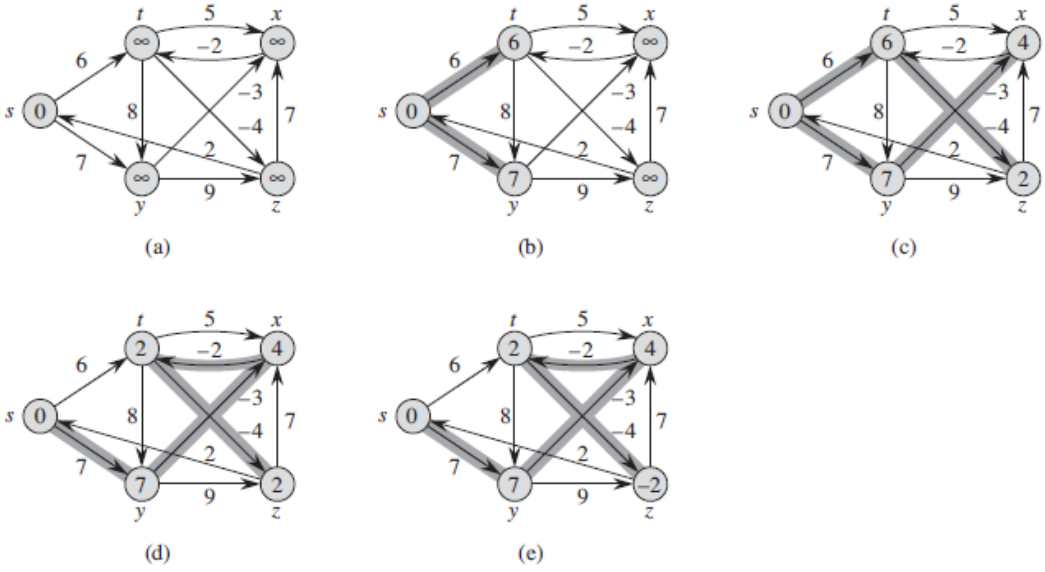
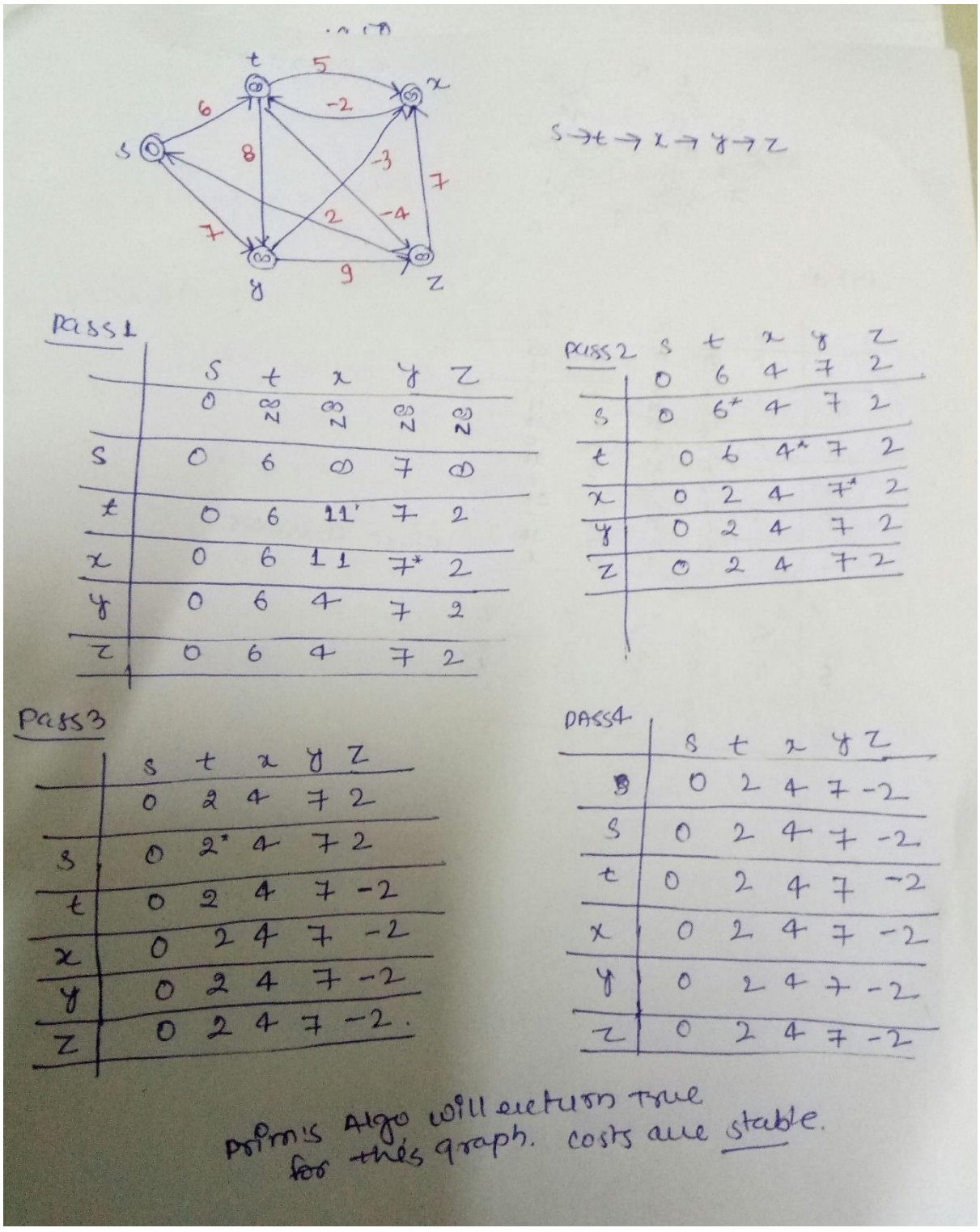


Figure 24.4 The execution of the Bellman-Ford algorithm. The source is vertex s. The d values appear within the vertices, and shaded edges indicate predecessor values: if edge (u, v) is shaded, then $v.\pi = u$. In this particular example, each pass relaxes the edges in the order $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$. **(a)** The situation just before the first pass over the edges. **(b)–(e)** The situation after each successive pass over the edges. The d and π values in part (e) are the final values. The Bellman-Ford algorithm returns TRUE in this example.



Prim's algorithm

Single-source shortest paths in directed acyclic graphs Θ(V + E)

By relaxing the edges of a weighted dag (**directed acyclic graph**) G = (V, E) according to a **topological sort** of its vertices, we can compute shortest paths from a single source in **Θ(V + E)** time.
Shortest paths are always well defined in a dag, since even if there are negative-weight edges, no negative-weight cycles can exist.

All-Pairs Shortest Paths

We can solve an all-pairs shortest-paths problem by running a single-source shortest-paths algorithm |V| times, once for each vertex as the source.

1. If all edge weights are nonnegative, we can use **Dijkstra's algorithm**.
   a. If we use the **linear-array implementation of the min-priority queue**, the running time is $O(V^3 + VE) = O(V^3)$.
   b. The **binary min-heap implementation of the min-priority queue** yields a running time of $O(VE \lg V)$
   c. The **min-priority queue with a Fibonacci heap**, yielding a running time of $O(V^2 \lg V + VE)$

2. If the graph has negative-weight edges, we cannot use Dijkstra's algorithm
   We must run the slower Bellman-Ford algorithm once from each vertex. The resulting running time is $O(V^2E)$ which on a **dense graph is** $O(V^4)$

Floyd-Warshall algorithm Θ(V^3)
It's **dynamic programming algorithm,** which runs in time $\Theta(V^3)$.'

Greedy Technique

Algorithms for **optimization problems** typically go through a sequence of steps, with a set of choices at each step.
A **greedy algorithm** always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.

Note: Greedy algorithms do not always yield optimal solutions, but for many problems they do.

Optimal substructure
A problem exhibits **optimal substructure** if an optimal solution to the problem contains within it optimal solutions to subproblems.

Fractional Knapsack: O(nlogn)

Objective: Object need to be taken to maximize profit without damage to knapsack
N = no. of objects
M = Capacity of Object
Wi = Weight of ith object
Pi = Profit of ith object

We first compute the value per pound **pi/wi** for each item. Obeying a greedy strategy, the thief begins by taking as much as possible of the item with the greatest value per pound.
Thus, by sorting the items by value per pound, **the greedy algorithm runs in O(nlogn) time**.

Algorithm: Greedy fractional knapsack gives optimal solution by giving priority to both profit and weight.

1. Take a array to store Profit/Weight info
   for(i = 1 to n)
   job[i] = Pi/Wi
2. Sort the array in descending order **O(nlogn)**
3. Take one by one object until the capacity of knapsack becomes zero **O(n)**

Que 16.2-6 Show how to solve the fractional knapsack problem in O(n) time.
Solution
Use a linear-time median algorithm to calculate the **median m** of the pi/wi ratios. Next, partition the items into three sets: G = {i : pi/wi > m}, E = {i : pi/wi = m}, and L = {i : pi/wi < m}; **this step takes linear time**.
The running time is given by the recurrence $T(n) \leq T(n/2) + \Theta(n)$, whose solution is **T(n)=O(n)**.

Huffman codes O(nlgon)

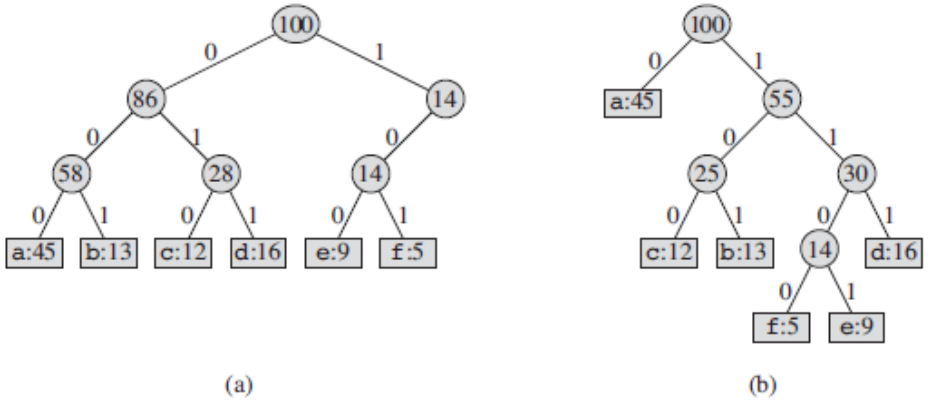
Suppose we have a 100,000-character data file that we wish to store compactly. Only 6 different characters appear.

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

If we use a **fixed-length code**, we need 3 bits to represent 6 characters: a = 000, b = 001. . . f = 101. This method requires **300,000** bits to code the entire file. Can we do better?

A **variable-length** code can do considerably better than a fixed-length code, by giving frequent characters short codewords and infrequent characters long codewords.

$$(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1{,}000 = 224{,}000 \text{ bits}$$



(a) The tree corresponding to the fixed-length code a = 000. . . f = 101.
(b) The tree corresponding to the optimal prefix code a = 0, b = 101. . . f = 1100.

To analyze the running time of Huffman's algorithm, we assume that **Q is implemented as a binary min-heap**.

1. For a set C of n characters, we can initialize Q in line 2 in **O(n)** time using the **BUILD-MIN-HEAP** procedure.
2. Repeatedly extract the two nodes and replace them in the queue and replace them in the queue with a new node z with their merger. The for loop in lines 3–8 executes exactly **n - 1 times.**
3. Each loop iteration require **O(logn)**
4. Thus, loop contributes **O(nlogn)**

```
HUFFMAN(C)
1  n = |C|
2  Q = C
3  for i = 1 to n − 1
4       allocate a new node z
5       z.left = x = EXTRACT-MIN(Q)
6       z.right = y = EXTRACT-MIN(Q)
7       z.freq = x.freq + y.freq
8       INSERT(Q, z)
9  return EXTRACT-MIN(Q)      // return the root of the tree
```

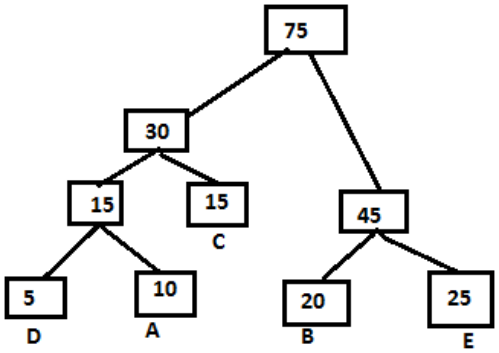
Optimal Merge Pattern: O(nlogn)

There are given n files with different number of records. Merge all files into one file optimally with minimum number of records movement.

1. Every time take 2 minimum and merge them

T.C Is O(nlogn)

Example Minimum number of records movement required to merge five files.
A-10 B-20 C-15 D-5 E-25



Total movements required: 15 + 30 + 45 + 75 = 165

Job Sequencing with Deadline: O(n^2)

1. Single CPU at a time, single job can be executed only.
2. In a span of time more than one job can be executed in round – robin fashion. 5 jobs are executing one by one
3. Interchange is not allowed (R.R not allowed)
4. Arrival time of all jobs are equal
5. 1 – unit of time of running time for every job eg: one month compulsory neither less or nor more

Jobs	J1	J2	J3	J4
Deadline	DL1	DL2	DL3	DL4
Profit	P1	P2	P3	P4

Eg: consider the following job sequencing with deadline problem

Jobs	J1	J2	J3	J4
Deadline	2	1	2	1
Profit	55	75	45	80

Solution {J2, J1} is possible but not {J1, J2}, find the optimal solution that gives maximum profit.

Max allowed month is 2, take an array of two slots, Month 2->[J1] take maximum profit job that can be completed in 2nd month, among all take maximum job which can be completed either in first month or 2nd month. That is J4
|J4|J1| total profit = 80+ 55 = 135

Algorithm:

Jobs	J1	J2	J3	J4	J5	J6
Profit	24	18	22	10	12	30
Deadline	5	3	3	2	4	2

1. Sort all the jobs in descending order based on profit. **O(nlogn)**
2. Find max deadline and take array of that size **O(n)** to find max dead line
3. For every slot i, apply linear search to find a job which contains **deadline >= i**. this step will take **O(n^2)** time because we have sorted on profit but searching based on deadline. Hence binary search can't be applied
4. As we find first job, stop and delete the job.

Jobs are sorted in descending order:

Jobs	J6	J1	J3	J2	J5	J4
Profit	30	24	22	18	12	10
Deadline	2	5	3	3	4	2

Max deadline is 5, so, take an array of size 5.

1	2	3	4	5

i=5, we start searching from LHS, first job is J1 whose deadline >=5, copy it and delete it from.

1	2	3	4	5
				J1

i=4, J5 is the first job whose deadline >= 4, copy it

1	2	3	4	5
			J5	J1

i=3

1	2	3	4	5
		J3	J5	J1

i=2

1	2	3	4	5
	J6	J3	J5	J1

i=1

1	2	3	4	5
J2	J6	J3	J5	J1

Dynamic Programming

Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems.
We typically apply **dynamic programming** to optimization problems. Such problems can have many possible solutions. Each solution has a value, and we wish to find a solution with the **optimal** (minimum or maximum) value.

1. D.P is applicable when the subproblems are not independent. Subproblems share subproblems.
2. DAC algorithm does more work than necessary repeatedly solving the common subproblems.
3. D.P solves each problem **exactly once** and save answer to a table.

Two methods of storing the result in memory

1. **Memorization (Top – Down)**
   Whenever we solve a problem first time we store it and reuse it next time.
2. **Tabulation (Bottom – Up)**
   We pre-compute the solutions in linear fashion and later use them.

Dynamic Programming is used when

1. Overlapping Subproblems
2. Optimal Substructure
   Eg: Floyd warshall, Bell-man ford etc

Note – Longest path problem doesn't have optimal substructure property.

Matrix-chain multiplication

We are given a sequence (chain) <A1, A2, ……., An> f n matrices to be multiplied, and we wish to compute the product

A1A2 … An

If the chain of matrices is <A1, A2, A3, A4>then we can **fully parenthesize** the product in 5 distinct ways:

$$(A_1(A_2(A_3A_4))),$$
$$(A_1((A_2A_3)A_4)),$$
$$((A_1A_2)(A_3A_4)),$$
$$((A_1(A_2A_3))A_4),$$
$$(((A_1A_2)A_3)A_4).$$

We state the **matrix-chain multiplication** problem as follows: given a chain $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices, where for $i = 1, 2, \ldots, n$, matrix A_i has dimension $p_{i-1} \times p_i$, Fully parenthesize the product A1A2 . . . An in a way that minimizes the number of **scalar multiplications**.

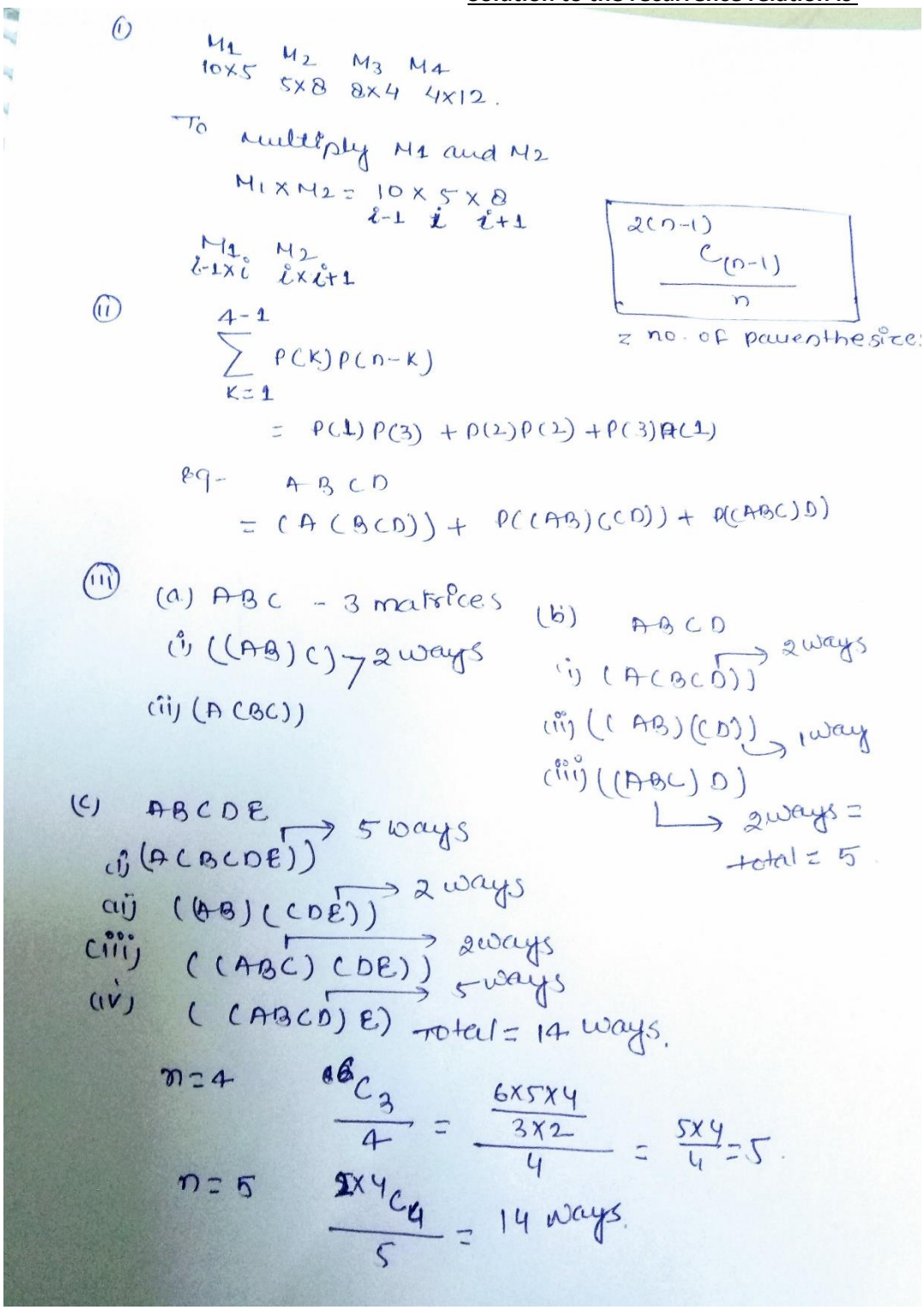
Counting the number of parenthesizations

Denote the number of alternative parenthesizations of a sequence of n matrices by **P(n).** When n = 1 we have just one matrix and therefore only one way to fully parenthesize the matrix product.

When n >= 2, a fully parenthesized matrix product is the product of two fully parenthesized matrix subproducts, and the split between the two subproducts may occur between the kth and (k+1)st matrices for any k = 1, 2, 3, ….. n-1 Thus, we obtain the recurrence:

$$P(n) = \begin{cases} 1 & \text{if } n = 1, \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2. \end{cases}$$

Solution to the recurrence relation is $\Omega(2^n)$.



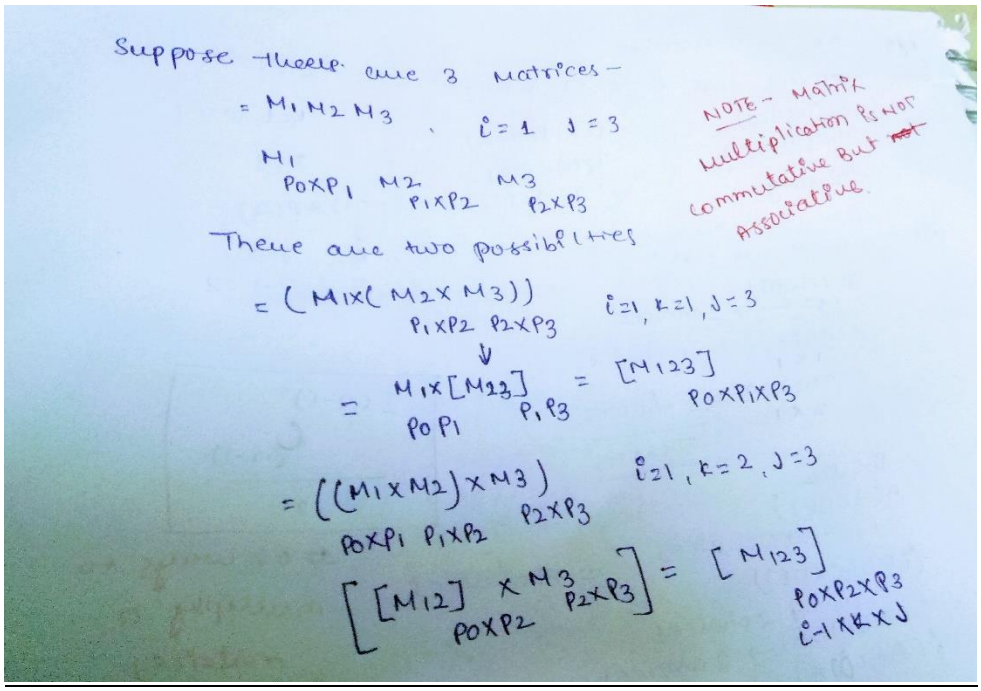
A recursive solution

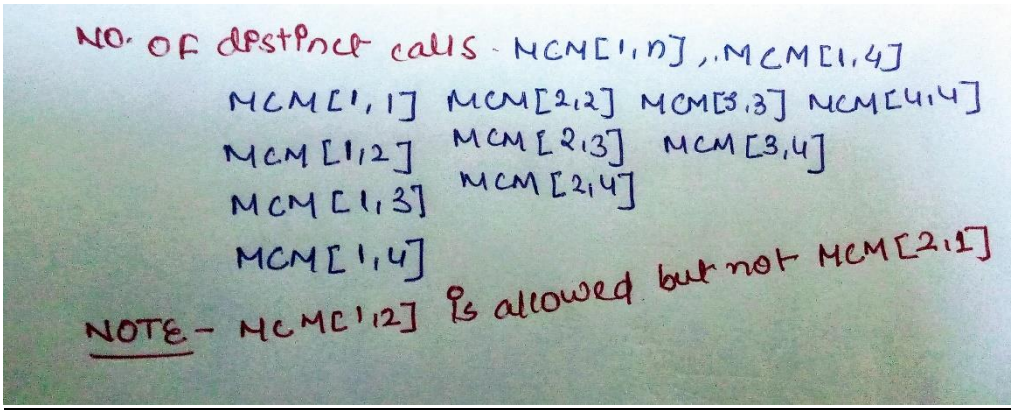
To determine the minimum cost of parenthesizing $A_i A_{i+1} \cdots A_j$ for $1 \le i \le j \le n$. Let $m[i, j]$ **be the minimum number of scalar Multiplications needed to compute the matrix** $A_{i..j}$

m[i , j] if i=j then it's a trivial problem.

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

Note: i <=k < j





n(n+1)/2 = 10 distinct function calls

total number of ways to multiply n matrix = (n-1)th catalan number = $\frac{\binom{2(n-1)}{(n-1)}}{n}$

	1	2	3	4
1	✓	✓	✓	✓
2	x	✓	✓	✓
3	x	x	✓	✓
4	x	x	x	✓

n(n+1)/2 distinct function calls, and each function call needs O(n). overall time complexity will be O(n^3).

Example - A2x5 B5x4 C4x2 D2x3

	1	2	3	4
1	x	40	56	68
2	x	x	40	70
3	x	x	x	24
4	x	x	x	x

Answer is 68 multiplications are required to multiply 4 matrices A,B,C and D

0/1 knapsack problem

- n items.
- Item i is worth $vi, weighs wi pounds.
- Find a most valuable subset of items with total weight ≤ W.
- Have to either take an item or not take it, but can't part of it.

Recursive Relation:

m = capacity of knapsack, n= number of items

$$01KS(m, n) \begin{cases} 0 & \text{if } m=0 \text{ or } n=0 \\ 01KS[n-1, m] & \text{if } m > wn \\ \max[01KS(n-1, m), 01KS(n-1, W - wn) + Pn] & \text{else} \end{cases}$$

Recursive Program: O(2^n)

```
int knapSack(int W, int wt[], int val[], int n)
{

   if (n == 0 || W == 0)
       return 0;
   if (wt[n-1] > W)
       return knapSack(W, wt, val, n-1);

   else return max( val[n-1] + knapSack(W-wt[n-1], wt, val, n-1),
                    knapSack(W, wt, val, n-1)
                  );
}
```

Time Complexity: it will be a complete binary tree. **O(2^n)**

Using Dynamic Programming O(m*n)

There will be distinct function calls **(m+1)*(n+1).**

The Time complexity using Dynamic Programming will be O(m*n), Where m is the capacity of knapsack and n is the number of items.

Longest Common Subsequence:

Given two sequences X and Y, we say that a sequence Z is a **common subsequence** of X and Y if Z is a subsequence of both X and Y.

1. For example, if **X = <A, B, C, B, D, A, B>** and **Y = <B, D, C, A, B, A>** the sequence **<B, C, A>** is a common subsequence of both X and Y.
2. The sequence **<B, C, A>** is not a longest common subsequence (LCS) of X and Y, it has length 3.
3. The sequence **<B, C, B, A>**, which is also common to both X and Y, has length 4.
4. The sequence **<B, C, B, A>** is an **Longest Common Subsequence** of X and Y, since X and Y have no common subsequence of length 5 or greater.

In the *longest-common-subsequence problem*, we are given two sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ and wish to find a maximum-Length common subsequence of X and Y.

Brute Force Approach:

We would **enumerate all subsequences of X** and check each subsequence to see whether it is also a subsequence of Y, keeping track of the longest subsequence we find. X has 2^m subsequences hence This **approach requires O(n*2^m) time complexity.**

Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

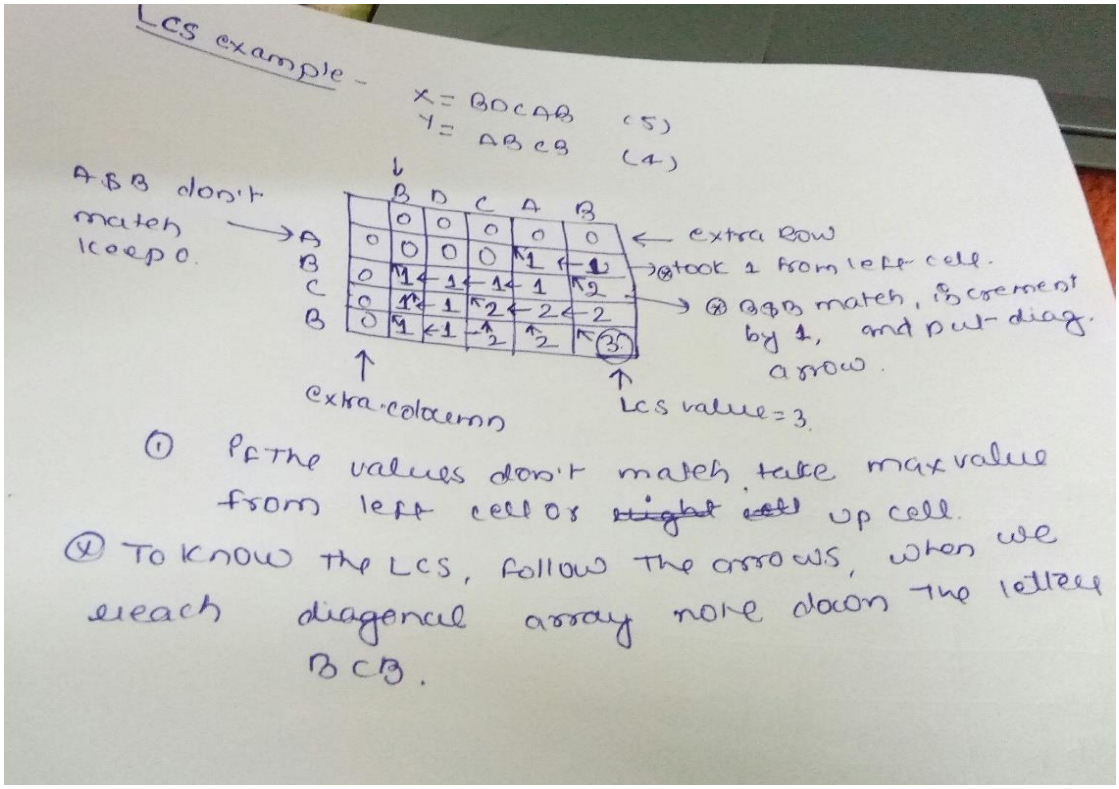
A recursive solution

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

Worst case **time complexity of this recurrence relation will be o(2^(m+n)**

Using Dynamic Programming

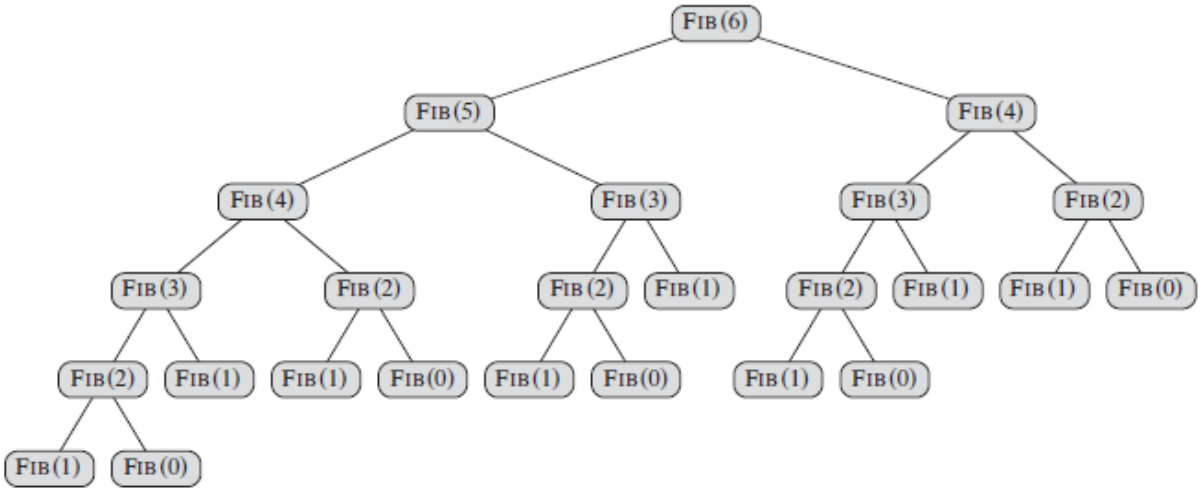
There are total distinct (m+1)(n+1) function calls, hence **time complexity** will be **O(m*n)** and Space complexity will be **O(m*n)** to store dynamic programming table.



When the value matches take the value from diagonal up cell and increment it by 1.

Fibonacci Series:-

$$
\begin{aligned}
F_0 &= 0\,, \\
F_1 &= 1\,, \\
F_i &= F_{i-1} + F_{i-2} \qquad \text{for } i \geq 2\,.
\end{aligned}
$$



Recursive Program:

```
FIB(n)
1  if n ≤ 1
2        return n
3  else x = FIB(n − 1)
4       y = FIB(n − 2)
5       return x + y
```

Let T(n) denote the running time of FIB(n). Since FIB(n) contains two recursive calls plus a constant amount of extra work, we obtain the recurrence

$$T(n) = T(n-1) + T(n-2) + \Theta(1)$$

I. **Time complexity for this recurrence relation is O(2^n).**
II. **Space Complexity is O(n)**

Fibonacci Series using Dynamic Programming:

In dynamic programming we will solve every problem **only once**. There are 6 unique function calls.

```
//Fibonacci Series using Dynamic Programming

int fib(int n)
{
  /* Declare an array to store Fibonacci numbers. */
  int f[n+1];
  int i;

  /* 0th and 1st number of the series are 0 and 1*/
  f[0] = 0;
  f[1] = 1;

  for (i = 2; i <= n; i++)
  {
      /* Add the previous 2 numbers in the series
         and store it */
      f[i] = f[i-1] + f[i-2];
  }

  return f[n];
}
```

Time complexity = O(n)
Space Complexity = O(n)

Multistage Graph

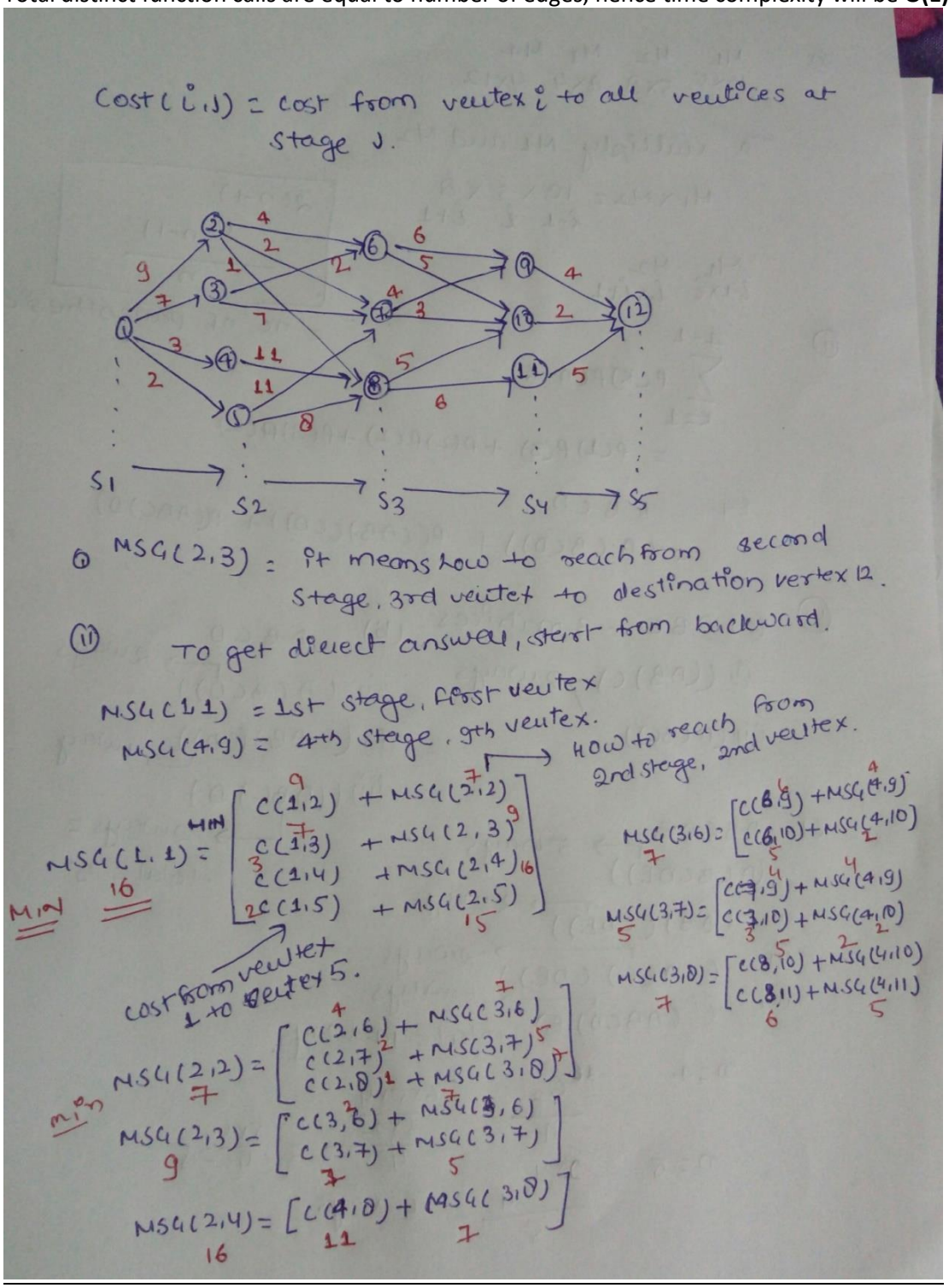
A multistage graph G = (V, E) is directed graph in which vertices are partitioned in K – disjoint sets where k>=2

In addition, if (u, v) is an edge E then u ∈ Vi and v ∈ Vi+1.

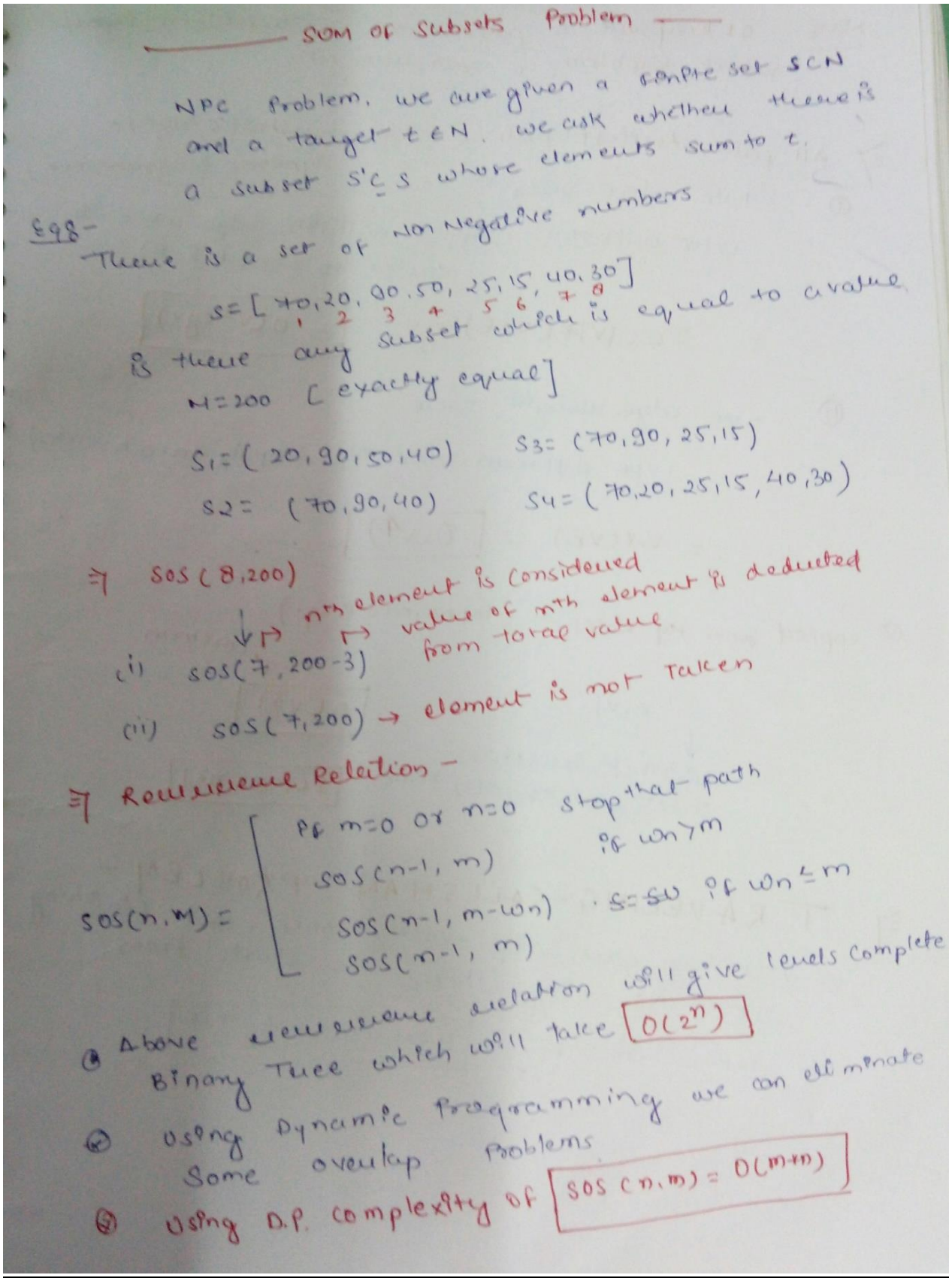
Let C(i, j) be the cost of edge (i, j), the cost of a path from S to T is the sum of the costs of the edge on the path.

The multistage graph problem is to find the min cost path from S to T.

Total distinct function calls are equal to number of edges, hence time complexity will be **O(E)**



Sum of Subset Problem



Sum of Subsets Problem

NPC Problem. We are given a finite set S of N and a target $t \in N$. We ask whether there is a subset $S' \subseteq S$ whose elements sum to t.

Eg:- There is a set of non Negative numbers

S = [70, 20, 90, 50, 25, 15, 40, 30]
1 2 3 4 5 6 7 8

Is there any subset which is equal to a value M = 200 [exactly equal]

S_1 = (20, 90, 50, 40)
S_2 = (70, 90, 40)
S_3 = (70, 90, 25, 15)
S_4 = (70, 20, 25, 15, 40, 30)

⇒ SOS(8, 200)
↓ nth element is considered
(i) SOS(7, 200-3) → value of nth element is deducted from total value.
(ii) SOS(7, 200) → element is not taken

⇒ Recurrence Relation –

$$SOS(n, M) = \begin{cases} \text{If } m=0 \text{ or } n=0 & \text{stop that path} \\ SOS(n-1, m) & \text{if } w_n > m \\ SOS(n-1, m-w_n) \text{ or } SOS(n-1, m) & \text{if } w_n \le m \end{cases}$$

① Above recurrence relation will give Binary Tree which will take $O(2^n)$ levels complete

② Using Dynamic Programming we can eliminate some overlap problems

③ Using D.P. complexity of $SOS(n, m) = O(m*n)$

Representations of graphs

We can choose between two standard ways to represent a graph G = (V, E) as a collection of **adjacency lists** or as an **adjacency matrix**.
Either way applies to both **directed and undirected** graphs.
Because the adjacency-list representation provides a compact way to represent **sparse** graphs those for which |E| is much less than |V^2|.
We may prefer an adjacency-matrix representation, however, when the graph is **dense** |E| is close to |V^2| or or when we need to be able to tell quickly if there is an edge connecting two given vertices.

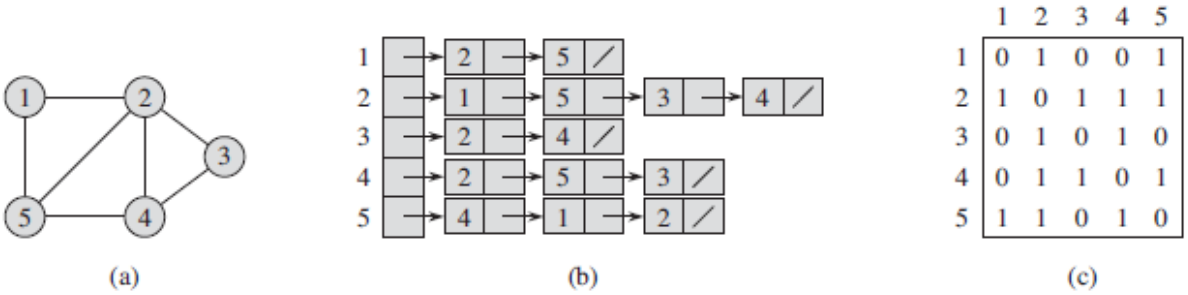


Figure 22.1 Two representations of an undirected graph. **(a)** An undirected graph G with 5 vertices and 7 edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.

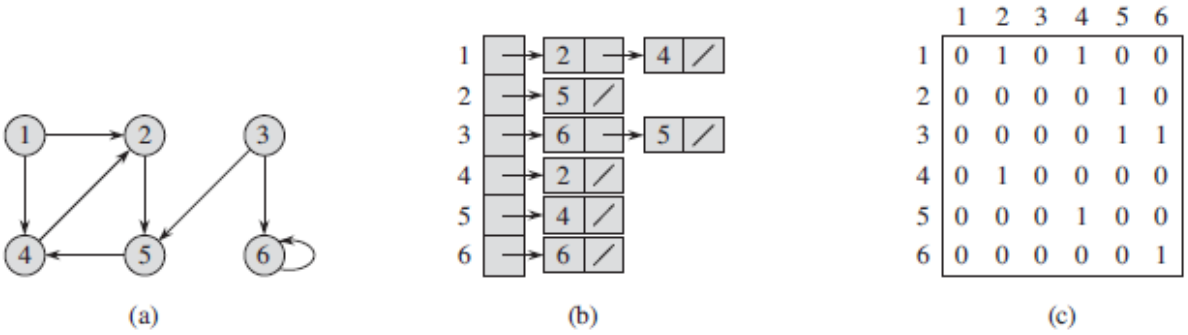


Figure 22.2 Two representations of a directed graph. **(a)** A directed graph G with 6 vertices and 8 edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.

Adjacency list representations of a graph G = (V, E) consists of **an array** Adj of , one for each vertex in V. For each $u \in V$ the adjacency list Adj[u] contains all the vertices v such that there is an edge $(u, v) \in E$ That is, **Adj[u] consists of all the vertices adjacent to u in G**.

1. If G is a **directed graph**, the sum of the lengths of all the adjacency lists is |E|
2. If G is a **undirected graph**, the sum of the lengths of all the adjacency lists is 2*|E|
3. The adjacency-list requires the amount of memory **Θ(V + E)**
4. For **weighted graph**, we store the weight w(u, v) of the edge $(u, v) \in E$ with vertex v in u's adjacency list.
5. **No quicker way** to determine whether a given edge (u, v) is present in the graph than to search for v in the adjacency list Adj[u].

Adjacency-matrix representation of a graph G(V, E), we assume that the vertices are numbered 1, 2, ...|V| in some arbitrary manner. Then the adjacency-matrix representation of a graph G consists of a |V|x|V| matrix.
A = Aij such that

$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E\ , \\ 0 & \text{otherwise}\ . \end{cases}$$

1. The adjacency matrix of a graph requires **Θ(V^2)** memory, independent of the number of edges in the graph.
2. The adjacency matrix A of an undirected graph is its own transpose: $A = A^{\mathrm{T}}$
3. For weighted graph, we store w(u, v) of the edge $(u, v) \in E$ as the entry in **row u and column v** of the adjacency matrix, otherwise **nil** if row doesn't exist.
4. Adjacency matrices carry a further advantage for unweighted graphs: they require only one bit per entry.

Que 22.1-1
Given an **adjacency-list representation** of a directed graph, how long does it take to compute the **out-degree of every vertex**? How long does it take to compute the **in-degrees**?
Solution
Out-degree of each vertex
1. Graph out-degree of a vertex u is equal to the length of Adj[u].
2. The sum of the lengths of all the adjacency lists in Adj is |E|.
3. Thus the time to compute the **out-degree of every vertex** is **Θ(V + E)**

In-degree of each vertex
1. The in-degree of a vertex u is equal to the number of times it appears in all the lists in **Adj**.
2. If we search all the lists for each vertex, time to compute the in-degree of every vertex is **Θ(VE)**
3. Alternatively, we can allocate an **array T** of size **|V|** and initialize its entries to zero.
4. We only need to **scan the lists in Adj once**, incrementing T[u] when we see u in the lists.
5. The values in T will be the in-degrees of every vertex.
6. **This can be done in Θ(V + E) time with Θ(V) additional storage.**

Que:-
Given an **adjacency-matrix representation** of a directed graph, how long does it take to compute the **out-degree of every vertex**? How long does it take to compute the **in-degrees**?

The adjacency-matrix A of any graph has **Θ(V^2)** entries, regardless of the number of edges in the graph.
For a directed graph, computing the out-degree of a vertex u is equivalent to scanning the row corresponding to u in A and summing the 1's.
Computing the out-degree of every vertex is equivalent to scanning all entries of A. hence, **Θ(V^2)**
Computing the in-degree of a vertex u is equivalent to scanning the column corresponding to u in A and summing the 1's. Thus it requires **Θ(V^2)**

Que 22.1-3

The *transpose* of a directed graph $G = (V, E)$ is the graph $G^T = (V, E^T)$, where $E^T = \{(v, u) \in V \times V : (u, v) \in E\}$. Thus, G^T is G with all its edges reversed.

For computing GT from G, for both the **adjacency list and adjacency-matrix representations of G**. What will be time complexity respectively?

Solution

For the **adjacency matrix** representation

1. To compute the graph transpose, we just take the **matrix transpose**.
2. Every entry above the diagonal, and swapping it with the entry that occurs below the diagonal.
3. This takes **O(V^2).**

For adjacency list:

We will maintain an **initially empty adjacency list representation of the transpose**. Then, we scan through every list in the original graph. If we are in the list corresponding to vertex v and see u as an entry in the list, then we add an entry of v to the list in the transpose graph corresponding to vertex u. Since this only requires a scan through all of the lists, it only takes time **O(|E| + |V|)**.

Breadth-first search

Given a graph **G = (V, E)** and a **distinguished source vertex s**, **breadth-first search** systematically explores the edges of G to "discover" every vertex that is reachable from s.

1. It computes the distance (**smallest number of edges**) from s to each reachable vertex.
2. It also produces a "breadth-first tree" with **root s** that contains all reachable vertices. For any vertex v reachable from s.
3. The simple path in the breadth-first tree from s to v corresponds to a "shortest path" from s to v in G, that is, a path containing the smallest number of edges.
4. **The algorithm works on both directed and undirected graphs**.
5. The algorithm discovers all vertices at distance **k** from s before discovering any vertices at distance **k + 1**.

1. To keep track of progress, breadth-first search colors each vertex **white**, **gray**, or **black**.
2. All vertices start out white and may later become gray and then black.
3. If $(u, v) \in E$ and vertex u is black, then vertex v is either gray or black; that is, all vertices adjacent to black vertices have been discovered.
4. Gray vertices may have some adjacent white vertices.

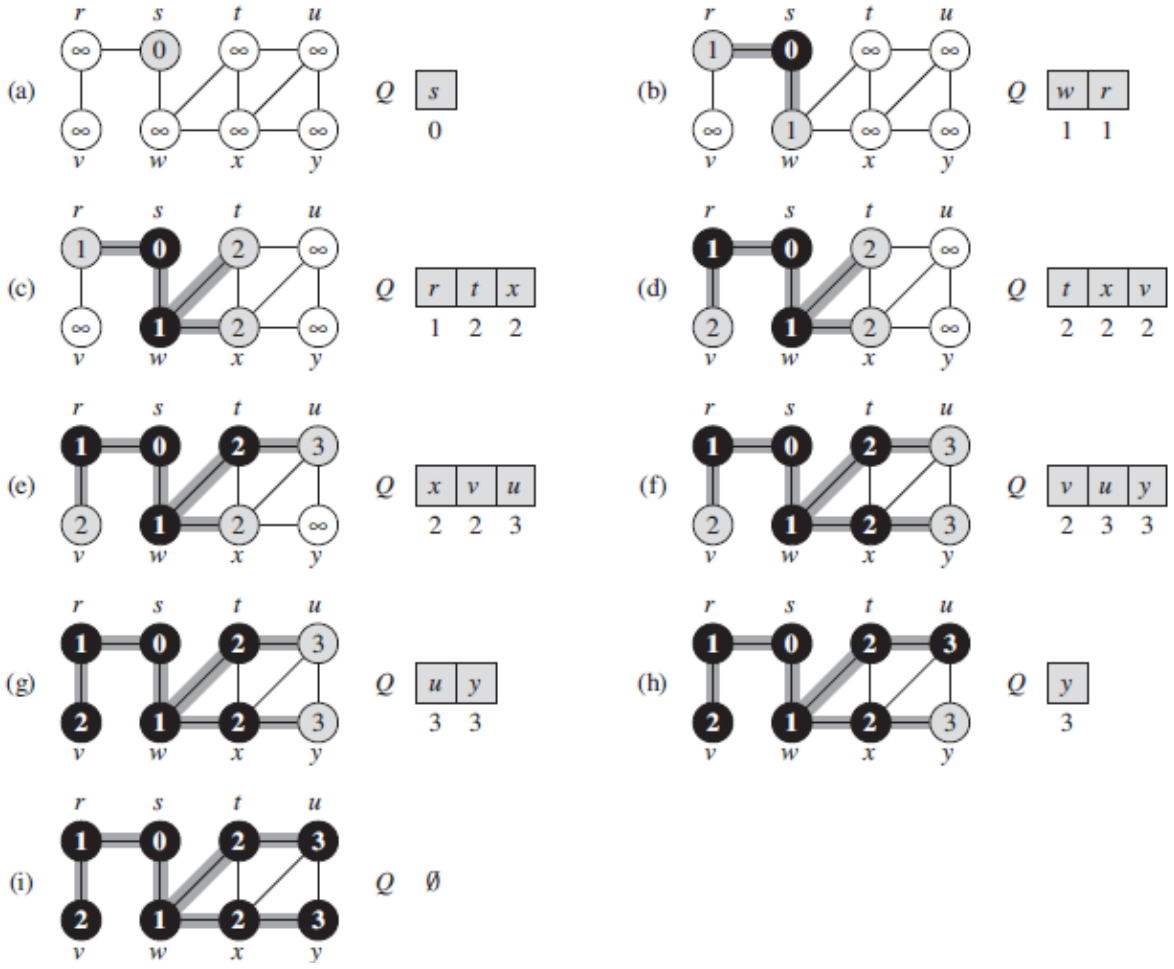
The **breadth-first-search** procedure BFS below assumes that the input graph G = (V, E) is represented using adjacency lists.

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2       u.color = WHITE
 3       u.d = ∞
 4       u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11       u = DEQUEUE(Q)
12       for each v ∈ G.Adj[u]
13            if v.color == WHITE
14                 v.color = GRAY
15                 v.d = u.d + 1
16                 v.π = u
17                 ENQUEUE(Q, v)
18       u.color = BLACK
```

The procedure BFS works as follows:

1. Except source vertex s, lines 1–4 paint every vertex white, set u.d to be infinity for each vertex u, and set the parent of every vertex to be NIL
2. Line 5 paints s **gray**, since we consider it to be discovered as the procedure begins.
3. Line 6 initializes s.d to 0, and line 7 sets the predecessor of the source to be NIL.
4. Lines 8–9 initialize **Q to the queue** containing just the vertex s.

5. The while loop of lines 10–18 iterates as long as there remain gray vertices, which are discovered vertices that have not yet had their adjacency lists fully examined.



Analysis:

1. After initialization, breadth-first search never whitens a vertex, and thus the test in line 13 ensures that each vertex is **enqueued at most once**, and hence **dequeued** at most once.
2. The operations of enqueuing and dequeuing take **O(1)** time, and so the total time devoted to queue operations is **O(V)**.
3. Because the procedure scans the adjacency list of each vertex only when the vertex is dequeued, it scans each adjacency list at most once.
4. Since the sum of the lengths of all the adjacency lists is |E|, the total time spent in scanning adjacency lists is **O(E)**.
5. The overhead for initialization is **O(V)**.
6. Thus the total running time of the BFS procedure is **O(V + E)** with adjacency-list representation of G.
7. If **adjacency Matrix** is used, BFS T.C will be **O(V^2)**, for each vertex u we **dequeue** we'll have to examine all vertices v to decide whether or not v is adjacent to u.

Shortest Path

BFS finds the distance to each reachable vertex in a graph G = (V, E) from a given source vertex $s \in V$. Define the **shortest-path distance** $\delta(s, v)$ from s to v as the minimum number of edges in any path from vertex s to vertex v; if there is no path from s to v, then $\delta(s, v) = \infty$

Lemma 22.1
Let G = (V, E) be a **directed or undirected graph**, and let $s \in V$ be an arbitrary vertex. Then, for any edge $(u, v) \in E$

$$\delta(s, v) \leq \delta(s, u) + 1$$

Proof If u is reachable from s, then so is v. In this case, the shortest path from s to v cannot be longer than the shortest path from s to u followed by the edge (u, v), and thus the inequality holds. If u is not reachable from s, then $\delta(s, u) = \infty$, and the inequality holds. ■

Lemma 22.2
Let G = (V, E) be a **directed or undirected graph**, and suppose that BFS is run on G from a given source vertex $s \in V$ Then upon termination, for each vertex $v \in V$, the value $v.d$ computed by BFS satisfies $v.d \geq \delta(s, v)$.

BFS Applications

1. If graph is connected.
2. If graph is cyclic.
3. Single shortest path if unweighted graph or weight is same for all edges.
4. To check Bipartite Graph property.
5. To know number of connected components.
6. No. of levels in the graph

Depth-first search

"To search "deeper" in the graph whenever possible"

Depth-first search explores edges out of the most recently discovered vertex v that still has unexplored edges leaving it.
Once all of v's edges have been explored, the search "backtracks" to explore edges leaving the vertex from which v was discovered. This process continues until we have discovered all the vertices that are reachable from the original source vertex.

If any undiscovered vertices remain, then depth-first search selects one of them as a new source, and it repeats the search from that source. The algorithm repeats this entire process until **it has discovered every vertex**.
Unlike breadth-first search, whose predecessor subgraph forms a tree, the predecessor subgraph produced by a depth-first search **may be composed of several trees**, because the search may repeat from multiple sources.

The predecessor subgraph of a depth-first search forms a depth-first forest comprising several depth-first trees.

Each vertex is initially **white**, is **grayed** when it is **discovered** in the search, and is **blackened** when it is **finished**, that is, **when its adjacency list has been examined completely**. This technique guarantees that each vertex ends up in exactly one depth-first tree, so that these trees are disjoint.

Besides creating a depth-first forest, **depth-first search also timestamps each vertex**. Each vertex v has two timestamps: the first timestamp v.d records when v is first **discovered** (and grayed), and the second timestamp v.f records when the search finishes examining v's adjacency list (and blackens v).

The procedure DFS below records when it **discovers vertex u in the attribute u.d** and when it **finishes vertex u in the attribute u.f** . These timestamps are integers between 1 and 2|V|, since there is one discovery event and one finishing event for each of the |V| vertices.
For every vertex u, u.d < u.f

Note: -

1. Vertex u is WHITE before time u.d, GRAY between time u.d and time u.f, and BLACK thereafter.
2. The input graph G may be undirected or directed.
3. The variable time is a global variable that we use for timestamping.

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
1  time = time + 1            // white vertex u has just been discovered
2  u.d = time
3  u.color = GRAY
4  for each v ∈ G.Adj[u]      // explore edge (u, v)
5      if v.color == WHITE
6          v.π = u
7          DFS-VISIT(G, v)
8  u.color = BLACK            // blacken u; it is finished
9  time = time + 1
10 u.f = time
```

Analysis

What is the running time of **DFS**? The loops on lines 1–3 and lines 5–7 of DFS take time **Θ(V).**

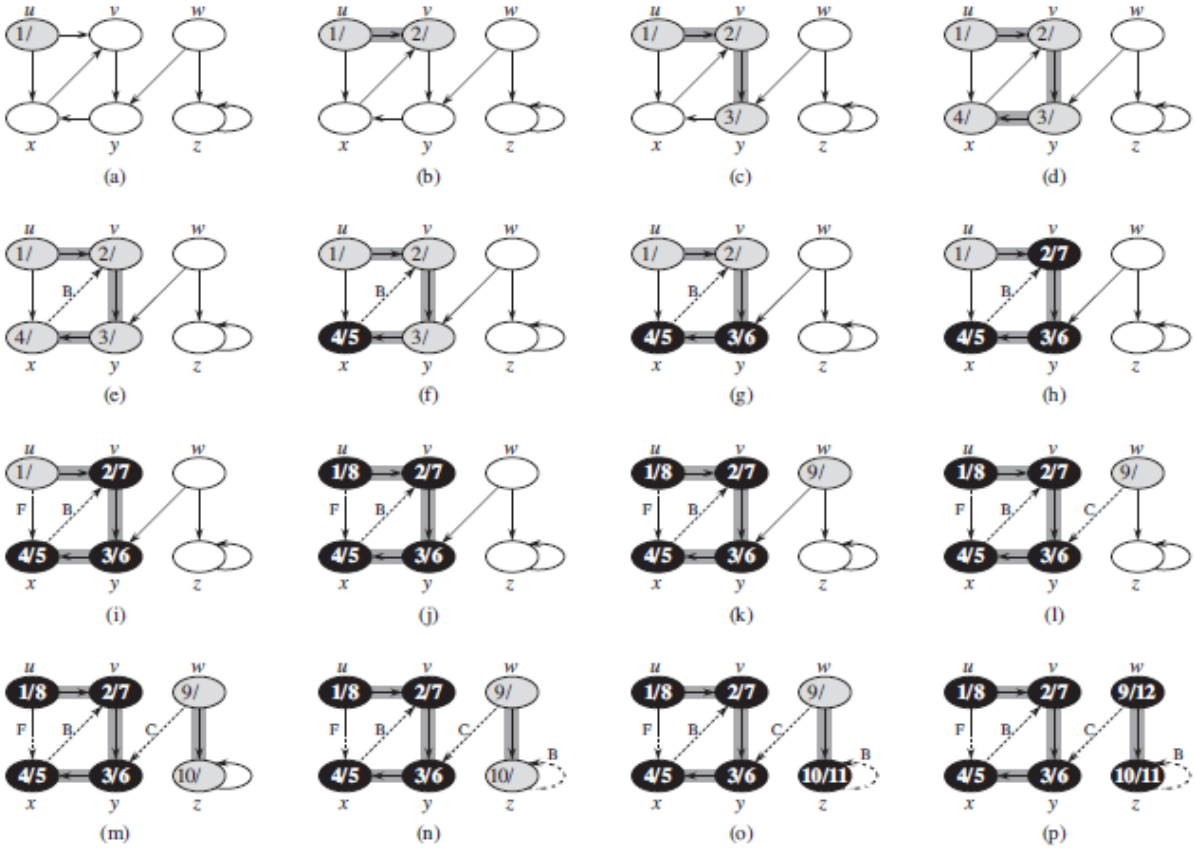
The procedure **DFS-VISIT** is called exactly once for each vertex $v \in V$. Since the vertex u on which **DFS-VISIT** is invoked must be white and the first thing DFS-VISIT does is paint vertex u gray.

During an execution of DFS-VISIT G(V, E), the loop on lines 4–7 executes **|Adj[v]|** times.

Since

$$\sum_{v \in V} |Adj[v]| = \Theta(E)$$

The total cost of executing lines 4–7 of DFS-VISIT is **Θ(E)**. **Running time of DFS is therefore Θ(V+E)**



Depth First Search Tree

Classification of edges

Tree edges are those **e**dges (u, v) is a tree edge if v was first discovered by exploring edge (u, v).

Back edges are those edges (u, v) connecting a vertex u to an ancestor v in a depth-first tree. We consider self-loops, which may occur in directed graphs, to be back edges.

Forward edges are those nontree edges (u, v) connecting a vertex u to a descendant v in a depth-first tree.
Cross edges they can go between vertices in the same depth-first tree, as long as one vertex is not an ancestor of the other, or they can go between vertices in different depth-first trees.
Note: - In a depth-first search of an undirected graph G, every edge of G is either a tree edge or a back edge.

Applications of DFS:

1. Graph is connected.
2. Finding no. of connected components.
3. To check if graph contains cycle.
4. Finding no. of **articulation point** in **O(E)** time.
5. Finding no. of **Bridges** in **O(E)** time.
6. Topological sort