Decrease-and-conquer: It is based on exploiting the relationship between a solution to a given instance of a problem and a solution to a smaller instance of the same problem. Once such a relationship is established, it can be exploited either top down (recursively) & bottom up (without a recursion). There are three major variations of decrease-and-conquer. They are

1) decrease by a constant.

2) decrease by a constant factor.

3) Variable size decrease.

1) **Decrease-by-a-constant**: The size of an instance is reduced by the same constant on each iteration of the algorithm. Typically, this constant is equal to one.

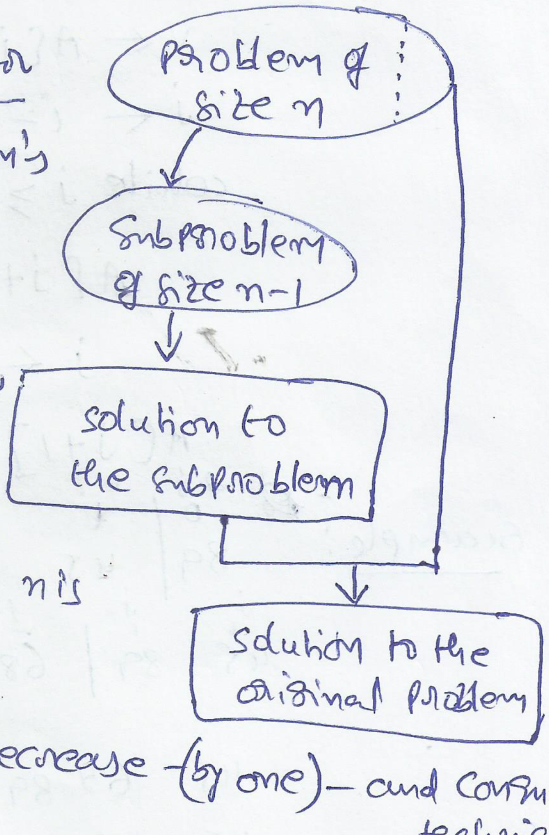
Ex: 1. Exponentiation problem of computing a^n.
2. Insertion sort.

2) **Decrease-by-a-constant-factor**

It suggests reducing a problem's instance by the same constant factor on each iteration of the algorithm. In most applications, this constant factor is equal to two.

Ex: Exponentiation problem if n is even, odd & if n=1.

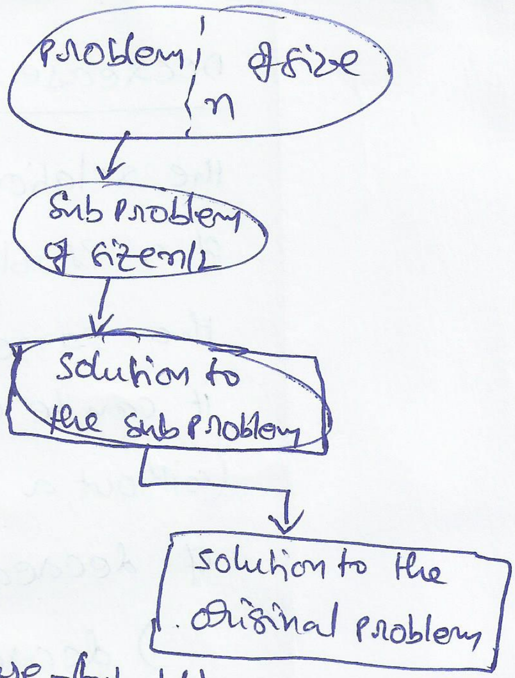
$(a^{n/2})^2$ $\quad (a^{(n-1)/2})^2 \quad$ a



Decrease-(by one)-and conquer technique

3. variable-size-decrease

A size reduction pattern varies from one iteration of an algorithm to another.



Decrease-(byhalf)-and-conquer technique.

Ex: Euclid's algorithm for computing the GCD.

$\gcd(m, n) = \gcd(n, m \bmod n)$.

Insertion sort

```
Algorithm insertionsort( A[0.... n-1])
//sorts a given array by insertion sort
// Input: An array A[0...n-1] of n orderable elements.
// output: Array A[0...n-1] sorted in nondecreasing order.
for i ← 1 to n-1 do
    v ← A[i]
    j ← i-1
    while j ≥ 0 and A[j] > v do
        A[j+1] ← A[j]
        j ← j-1
    A[j+1] ← v.
```

Example:

```
j    v i
0    1    2    3    4    5    6
89 | 45   68   90   29   34   17

j    j,i  i
0    1    2    3    4    5    6
45   89 | 68   90   29   34   17     shift

45   68   89 | 90   29   34   17     shift
45   68   89   90 | 29   34   17
29   45   68   89   90 | 34   17
29   34   45   68   89   90 | 17
```

17 29 34 45 68 89 90

Ex 2: 8 9 5 6 7 0 1 3 2

Ex 3: I, N, S, E, R, T, I, O, N.

Insertion sort is an application of the decrease-by-one technique to sort an array A[0 ··· n-1].

technisue: we assume that the smaller problem of sorting the array A[0 ··· n-2] has already been solved to give us a sorted array of size n-1: A[0] ≤ ··· ≤ A[n-2]. start with A[n-1], continue checking the remaining elements with A[n-1] and place them in right position.

There are three reasonable alternatives for doing this.

1. we can scan the sorted subarray from left to right until the first element greater than & equal to A[n-1] is encountered and then insert A[n-1] right before that element.

2. we can scan the sorted subarray from right to left until the first element smaller than or equal to A[n-1] is encountered and then insert A[n-1] right after that element. — straight insertion sort & simply insertion sort.

3. use binary search to find an appropriate position for A[n-1] in the sorted portion of the array. — binary insertion sort.

The basic operation of the algorithm is the key comparison A[j] > v.

Best case : It happens only when the given elements are already in ascending order.

Ex: 1 2 3 4 5 6 — no of comparisons

1 | 2 3 4 5 6 — 1

1 2 | 3 4 5 6 — 1

1 2 3 | 4 5 6 — 1

1 2 3 4 | 5 6 — 1

1 2 3 4 5 | 6 — 1

1 2 3 4 5 6 — 1

$C_{best}(n) = \Theta(n)$

$$C_{best}(n) = \sum_{i=1}^{n-1} i$$
$$= n-1$$
$$\in \Theta(n)$$

the comparison $A[j] > v$ is executed only once on every iteration of the outer loop.

worst case: In the worst case, $A[j] > v$ is executed the largest number of times. The worst case input is an array of strictly decreasing values. The no of key comparisons for such an input is as follows

6 5 4 3 2 1 — no of comparisons

6 | 5 4 3 2 1 — 1

5 6 | 4 3 2 1 — 2

4 5 6 | 3 2 1 — 3

3 4 5 6 | 2 1 — 4

2 3 4 5 6 | 1 — 5

1 2 3 4 5 6 | — 15

avg-case is approximately equal to worst-case

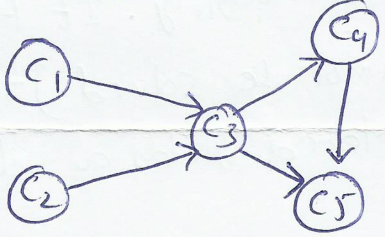
Total no of comparisons
$= 5+4+3+2+1$
$= 15$

$$C_{worst}(n) = \frac{n(n-1)}{2}$$
$$\in \Theta(n^2).$$

Topological sorting the Topological sorting problem asks to list vertices of a digraph in an order such that for every edge of the digraph, the vertex it starts at is listed before the vertex it points to. This problem has a solution if and only if a digraph is a dag (i.e., directed acyclic graph), i.e., it has no directed cycles.

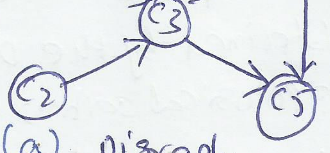
A digraph is a graph with directions on its ~~[illegible]~~ edges.

There are two algorithms for solving the topological sorting problem. The first one is based on DFS; the second is based on the direct implementation of the decrease-by-one technique.



Digraph

The first algorithm is a simple application of DFS: Perform a DFS traversal and note the order in which vertices become dead ends (i.e., vertices are popped off the traversal stack). Reversing this order yields a solution to the topological sorting problem, provided, of course, no back edge has been encountered during the traversal. If a back edge has been encountered, the digraph is not a dag, and topological sorting of its vertices is impossible.



(a) Digraph

C_{5_1}

C_{4_2}

C_{3_3}

C_{1_4} C_{2_6}

(b) DFS traversal

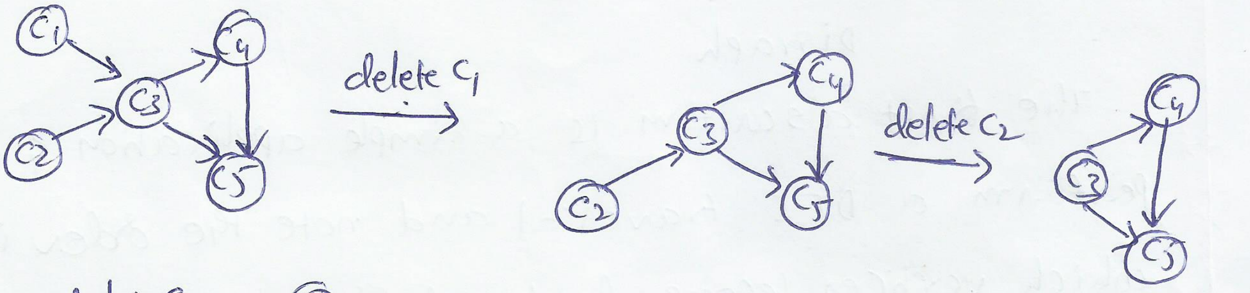
The popping-off order:
C5, C4, C3, C1, C2.
The topologically sorted list:
C2 C1 → C3 → C4 → C5

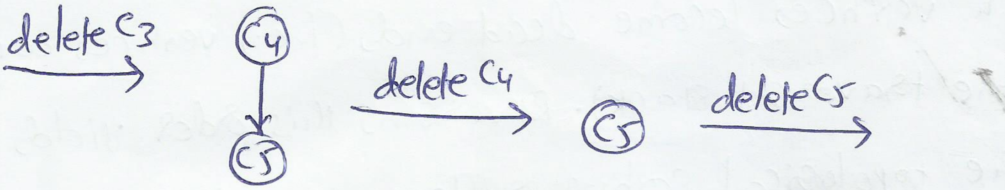
(c) solution to the problem.

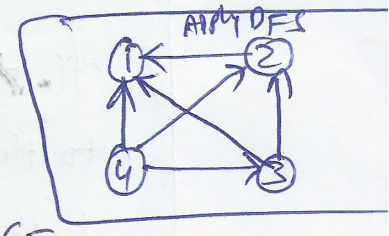
why does the algorithm work?

When a vertex v is popped off a DFS stack, no vertex u with an edge from u to v can be among the vertices popped off before v. (Otherwise, (u, v) would have been a back edge.) Hence, any such vertex u will be listed after v in the popped-off order list, and before v in the reversed list.

The <u>second algorithm</u> is based on a direct implementation of the decrease (by one) - and - conquer technique: repeatedly, identify in a remaining digraph a source, which is a vertex with no incoming edges, and delete it along with all the edges outgoing from it. (If there are several sources, break the tie arbitrarily. If there is none, stop because the problem cannot be solved.) The order in which the vertices are deleted yields a solution to the topological sorting problem.







The solution obtained is c1, c2, c3, c4, c5.

Illustration of the source-removal algorithm for the topological sorting problem. On each iteration, a vertex with no incoming edges is deleted from the digraph.

Note that the solution obtained by the source-removal algorithm is different from the one obtained by the DFS-based algorithm. Both of them are correct; of course; the topological sorting problem may have several alternative solutions.

Decrease-by-a-Constant-Factor algorithms

Falce-coin Problem The problem is to design an efficient algorithm for detecting the fake coin.

The most natural idea for solving this problem is to divide n coins into two piles of $\lfloor n/2 \rfloor$ coins each, and put the two piles on the scale if n is even, otherwise leaving one extra coin apart if n is odd. If the piles weigh the same, the coin put aside must be fake; otherwise we can proceed in the same manner with the lighter pile, which must be the one with the fake coin.

Note that even though we divide the coins into two subsets, after one weighing we are left to solve a single problem of half the original size. It is by decrease (by half)-and-conquer Technique.

Recurrence relation for the number of weighings $W(n)$ needed by this algorithm in the worst case:

$$W(n) = W(\lfloor n/2 \rfloor) + 1 \quad \text{for } n > 1,$$

$$W(1) = 0.$$

The solution to the recurrence for the no. of weighings is $W(n) = \lfloor \log_2 n \rfloor$.

$\log_2 2 = 1$ ~~$\log_2 3 = 1.5848$~~

$\log_3 2 = 0.63$

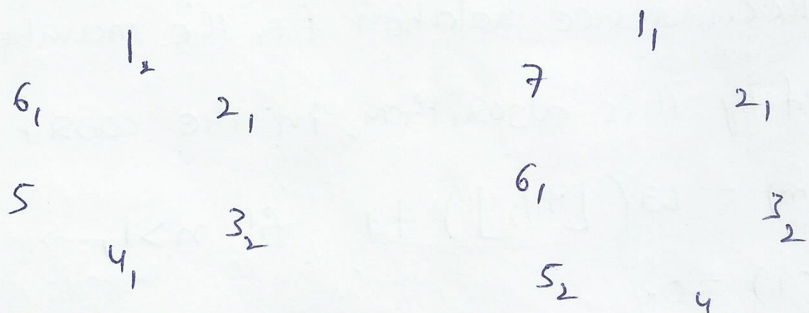
This algorithm is not the most efficient solution. We would better off dividing the coins not into two but into three piles of about $n/3$ coins each. After weighing two of the piles, we can reduce the instance size by a factor of three ($\log_3 n$ is smaller than $\log_2 n$) can you tell by what factor?

Josephus Problem

Let n people numbered 1 to n stand in a circle. Starting the count with person number 1, we eliminate every second person until only one survivor is left.

The problem is to determine the survivor's number J(n).

For example, if n is 6, people in positions 2, 4, & 6 will be eliminated on the first pass through the circle, and people in initial positions 3 and 1 will be eliminated on the second pass, leaving a sole survivor in initial position 5 — thus, J(6) = 5. To give another example, if n is 7, people in positions 2, 4, 6, and 1 will be eliminated on the first pass and people in positions 5 and 3 on the second — thus, J(7) = 7.



(a) (b)

Compute J(n) for n = 1, 2, ⋯, 15

Instances of the Josephus problem for a) n = 6 and b) n = 7. Subscript numbers indicate the pass on which one person in that position is eliminated. The solutions are J(6) = 5 and J(7) = 7, respectively.

If n is even, i.e n = 2k, To get the initial position of a (survival) person, the relationship will be

$$J(2k) = 2J(k) - 1$$

J(2k) = ~~new~~ initial position
J(k) = ~~initial~~ new position.

If n is odd ($n>1$), i.e., $n = 2k+1$, to get the initial position that corresponds to the new position numbering, we set

$$\overset{\text{initial old position}}{J(2k+1)} = \overset{\text{new position}}{2\,J(k)+1}$$

can we get a closed-form solution to the two-case recurrence (subject to the initial condition $J(1) = 1$)? The answer is yes.

The most elegant form of the closed-form answer involves the binary representation of size n: $J(n)$ can be obtained by a <u>one-bit cyclic shift left of n itself</u>.

For ex, $J(6) = J(110_2) = 101_2 = 5$

$J(7) = J(111_2) = 111_2 = 7.$

~~[illegible]~~

$J(10) = J(1010)_2 = 0101 = 5$

$J(15) = J(1111)_2 = 1111 = 15$

$J(41) = J(101001)_2 = 010011 = 19$

<u>Exercise</u>

1. solve the falce coin recurrence for $n = 3^k$.

2. Find $J(30)$ - the solution to the Josephus problem for $n = 30$.

Prepared by: G. Venugopal

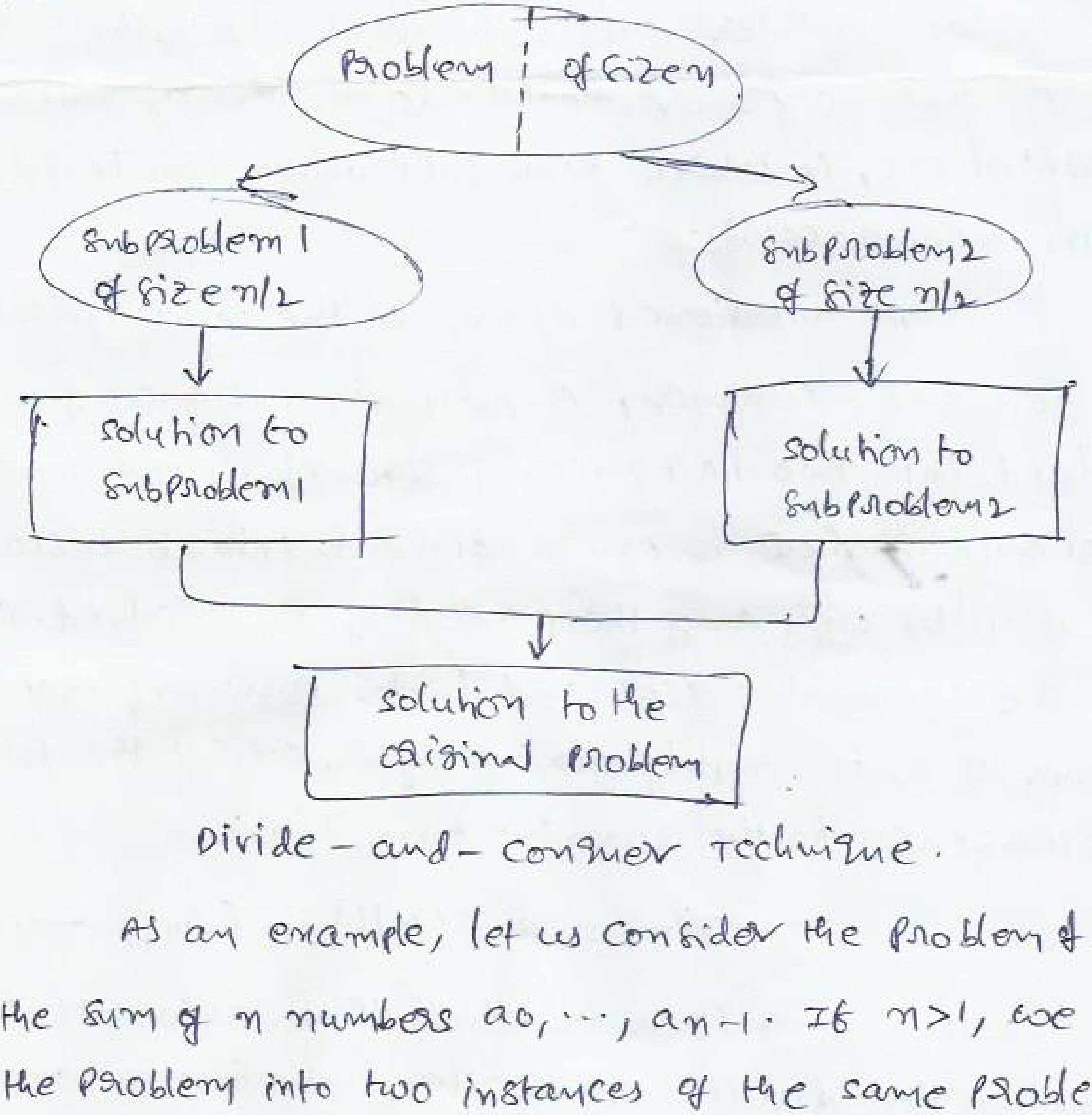
Abstract from: Introduction to the Design and analysis of Algorithms, 2e, Anany Levitin, Pearson

UNIT- III

DIVIDE- AND- CONQUER

Divide- and - Conquer algorithms works according to the following general plan:

1. A problem's instance is divided into several smaller instances of the same problem, ideally of about the same size.
2. The smaller instances are solved.
3. If necessary, the solutions obtained for the smaller instances are combined to get a solution to the original instance.



Divide - and- Conquer Technique.

As an example, let us consider the problem of computing the sum of n numbers $a_0, \cdots, a_{n-1}$. If $n > 1$, we can divide the problem into two instances of the same problem:

to compute the sum of the first $\lfloor n/2 \rfloor$ numbers and to compute the sum of the remaining $\lceil n/2 \rceil$ numbers. (of course, if $n=1$, we simply return a_0 as the answer.) once each of these two sums is computed, we can add their values to get the sum in questio

$$a_0 + \dots + a_{n-1} = (a_0 + \dots + a_{\lfloor n/2 \rfloor -1}) + (a_{\lfloor n/2 \rfloor} + \dots a_{n-1})$$

is this an efficient way to compute the sum of n numbe a moment of reflection, a small example of summing, say, four numbers by this algorithm, a formal analysis, and common sense all lead to a negative answer to this question.

Thus, not every divide-and-conquer algorithm is necessarily more efficient than even a brute-force solution. Though we consider only sequential algorithms here, the divide-and-conquer technique is ideally suited for Parall computations, in which each subproblem can be solved simultan by its own processor.

As mentioned above, in the most typical case of divide-and-conquer, a problem's instance of size n is divided into two instances of size n/2. more generally, an instance of size n can be divided into b instances of size n/b, with a of them needing to be solved. (Here, a and b are constants; $a \geq 1$ and $b > 1$.) Assuming that size n is power of b, to simplify our analysis, we get the following recurrence for the running time T(n);

$$T(n) = a\,T(n/b) + f(n), \longrightarrow \textcircled{1}$$

where f(n) is a function that accounts for the time spent on the dividing the problem into smaller ones and on combining their solutions (For the summation example, $a=b=2$ and $f(n)=1$.) Recurrence ① is called the general divide-and-conquer recurrence. obviously, the order of growth of its solution $T(n)$ depends on the values of the constants a and b and the order of growth of the function $f(n)$. The efficiency analysis of many divide-and-conquer algorithms is greatly simplified by the following theorem (master theorem)

theorem (master theorem) If $f(n) \in \Theta(n^d)$ with $d \geq 0$ in recurrence equation ①, then.

$$T(n) \in \begin{cases} \Theta(n^d) & \text{case 1} \quad \text{if } a < b^d \\ \Theta(n^d \log n) & \text{case 2} \quad \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{case 3} \quad \text{if } a > b^d \end{cases}$$

(Analogous results hold for the O & Ω notations, too).

For ex, the recurrence equation for the number of additions $A(n)$ made by the divide-and-conquer sum-computation algorithm on inputs of size $n = 2^k$ is

$$A(n) = 2A(n/2) + 1.$$

thus, for this example, $a=2, b=2$, and $d=0$; hence, since $a > b^d$;

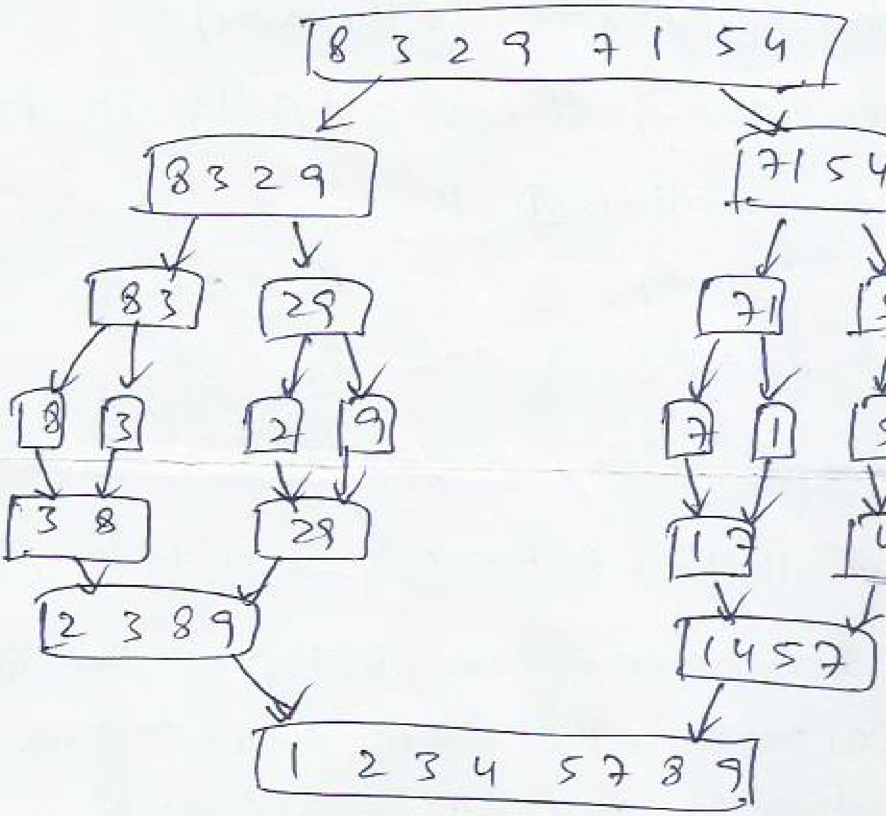
$$A(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 2}) = \Theta(n).$$

Ex1: Find the order of growth for solutions of the following recurrences.

1. $T(n) = 4T(n/2) + n, \ T(1) = 1.$ $d=1 \xrightarrow{\text{case 3}} \Theta(n^{\log_2 4}) = \Theta(n^2)$
2. $T(n) = 4T(n/2) + n^2, \ T(1) = 1.$ $d=2 \xrightarrow{\text{case 2}} \Theta(n^d \log n) = \Theta(n^2 \log n)$
3. $T(n) = 4T(n/2) + n^3, \ T(1) = 1.$ $d=3 \xrightarrow{\text{case 1}} \Theta(n^d) = \Theta(n^3).$

mergesort

mergesort is a perfect example of a successful application of the divide-and-conquer technique. It sorts a given array $A[0 \cdots n-1]$ by dividing it into two halves $A[0 \cdots \lfloor n/2 \rfloor - 1]$ and $A[\lfloor n/2 \rfloor \cdots n-1]$, sorting each of them recursively, and then merging the two smaller sorted arrays into a single sorted one.



Example of mergesort operation.

```
Algorithm mergesort ( $A[0 \cdots n-1]$ )
//sorts array $A[0 \cdots n-1]$ by recursive mergesort.
// Input: An array $A[0 \cdots n-1]$ of orderable elements.
// output: Array $A[0 \cdots n-1]$ sorted in nondecreasing order.
 if $n > 1$
      copy $A[0 \cdots \lfloor n/2 \rfloor - 1]$ to $B[0 \cdots \lfloor n/2 \rfloor - 1]$
      copy $A[\lfloor n/2 \rfloor \cdots n-1]$ to $C[0 \cdots \lceil n/2 \rceil - 1]$
      mergesort($B[0 \cdots \lfloor n/2 \rfloor - 1]$)
      mergesort($C[0 \cdots \lceil n/2 \rceil - 1]$)
      merge($B, C, A$).
```

The merging of two sorted arrays can be done as follows: Two Pointers (array indices) are initialized to point to the first elements of the arrays being merged. The elements pointed to are compared, and the smaller of them is added to a new array being constructed; after that, the index of the smaller element is incremented to point to its immediate successor in the array it was copied from. This operation is repeated until one of the two given arrays is exhausted, and then the remaining elements of the array are copied to the end of the new array.

```
Algorithm  merge(B[0... p-1], C[0...q-1], A[0... p+q-1])
//merges two sorted arrays into one sorted array.
//Input: Arrays B[0... p-1] and C[0... q-1] both sorted.
//Output: Sorted array A[0... p+q-1] of the elements of B
          and C.
   i ← 0; j ← 0; k ← 0
   while i < p and j < q do
        if B[i] ≤ C[j]
           A[k] ← B[i];  i ← i+1.
        else
           A[k] ← C[j]; j ← j+1
        k ← k+1
   if i == p
        copy C[j .... q-1] to A[k... p+q-1].
   else
        copy B[i... p-1] to A[k... p+q-1].
```

How efficient is mergesort? Assuming for simplicity that n is a power of 2, the recurrence relation for the number of key comparisons $C(n)$ is

$$C(n) = 2C(n/2) + C_{merge}(n) \text{ for } n>1, \; C(1)=0.$$

let us analyze $C_{merge}(n)$, the number of key comparisons performed during the merge stage. At each step, exactly one comparison is made, after which the total number of elements in the two arrays still needed to be processed is reduced by one. In the worst case, neither of the two arrays becomes empty before the other one contains just one element (ex., smaller elements may come from the alternating arrays). Therefore, for the worst case, $C_{merge}(n) = n-1$, and we have the recurrence.

$$C_{worst}(n) = 2C_{worst}(n/2) + n-1 \text{ for } n>1, \; C_{worst}(1)=0.$$

Hence, according to the master theorem, $C_{worst}(n) \in \Theta(n \log n)$.

Ex: Apply merge sort to sort the list E, X, A, M, P, L, E in alphabetical order.

Quicksort It is based on the divide-and-conquer approach unlike mergesort, which divides its input's elements according to their position in the array, quicksort divides them according to their value. quicksort rearranges elements of a given array A [0 ... n-1] to achieve its Partition. partition is a situation where all the elements ~~of a given array A[0...n-1]~~ before some position s are smaller than or equal to A[s] and all the elements after position s are greater than or equal to A[s]. i.e.,

$$\underbrace{A[0] \; \ldots\ldots \; A[s-1]}_{\text{all are } \leq A[s]} \quad A[s] \quad \underbrace{A[s+1] \ldots A[n-1]}_{\text{all are } \geq A[s]}$$

obviously, after a partition has been achieved, A[s] will be in its final position in the sorted array, and we can continue sorting the two subarrays of the elements preceding & following A[s] independently. (ex, by the same method)

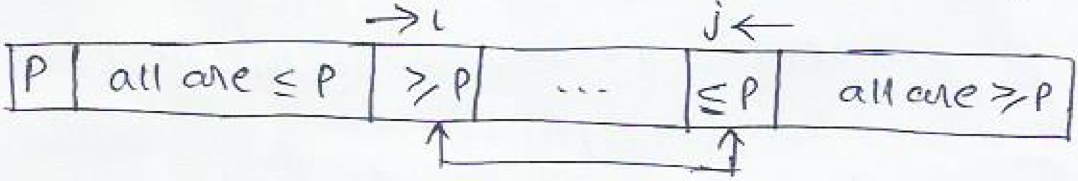
```
ALGORITHM  quicksort(A[l...r])
//sorts a subarray by quicksort.
//Input: A subarray A[l...r] of A[0...n-1], defined by
its left and right indices l and r.
//Output: subarray A[l...r] sorted in nondecreasing order

        if l<r
              s ← Partition (A[l...r])  //s is a split position.
              quicksort(A[l...s-1])
              quicksort(A[s+1...r])
```

A partition of $A[0 \cdots n-1]$ and its subarray $A[l \cdots r]$ $(0 \le l < r \le n-1)$ can be achieved by the following algorithm.

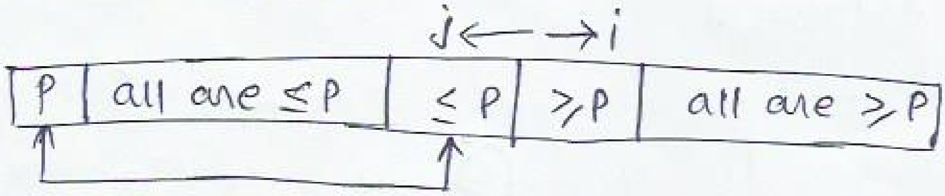
First, we select an element w.r.to whose value we are going to divide the subarray. we call this element as the pivot. There are several different strategies for selecting a pivot; we will return to this issue when we analyze the algorithm's efficiency. For now, we use the simplest strategy of selecting the subarray's first element: $p = A[l]$.

There are also several alternative procedures for rearranging elements to achieve a partition. Here we use an efficient method based on two scans of the subarray: one is left-to-right and the other right-to-left, each comparing the subarray's elements with the pivot. The left-to-right scan, denoted below by index i, starts with the second element. Since we want elements smaller than the pivot to be in the first part of the subarray, this scan skips over elements that are smaller than the pivot and stops on encountering the first element greater than & equal to the pivot. The right-to-left scan, denoted below by index j, starts with the last element of the subarray. Since we want elements larger than the pivot to be in the second part of the subarray, this scan skips over elements that are larger than the pivot and stops on encountering the first element smaller than or equal to the pivot.

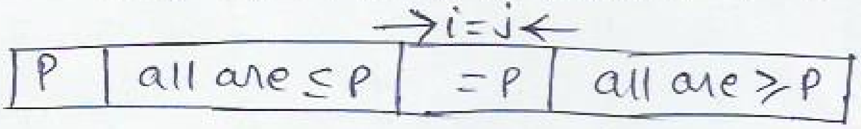
After both scans stop, three situations may arise, depending on whether & not the scanning indices have crossed. If scanning indices i and j have not crossed, i.e., i < j, exchange A[i] & A[j] and resume the scans by incrementing i and decrementing j, respectively:



If the scanning indices have crossed over, i.e., i > j, you will have partitioned the array after exchanging the pivot with A[j]:



Finally, if the scanning indices stop while pointing to the same element, i.e., i = j, the value they are pointing to must be equal to p (why?). Thus, you have the array partitioned, with the split position s = i = j:



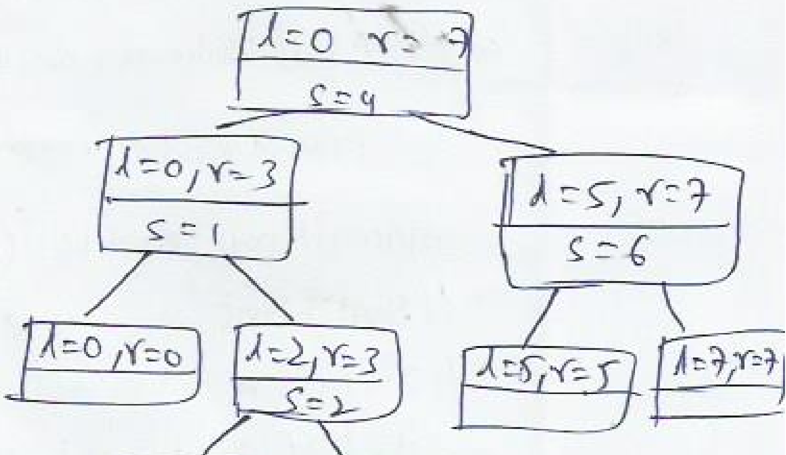
You can combine the last case with the case of crossed-over indices (i > j) by exchanging the pivot with A[j] whenever i ≥ j.

Pseudocode implementing this partitioning procedure.

```
Algorithm partition( A[l...r] )
// Partitions a subarray by using its first element as a pivot
// Input: A subarray A[l...r] of A[0...n-1], defined by
   its left and right indices l and r (l < r).
// Output: A partition of A[l...r], with the split position
   returned as this function's value.
```

```
P ← A[l]
i ← l;  j ← r+1.
repeat
    repeat  i ← i+1  until A[i] ≥ P.
    repeat  j ← j-1  until A[j] ≤ P.
    swap(A[i], A[j]).
until i ≥ j.
  swap(A[i], A[j])  // undo last swap when i≥j.
  swap(A[l], A[j])
  return j
```

```
Ex:       0    1    2    3    4    5    6    7          l=0    r=7
               i                             j
Pivot     5    3    1    9   (8)   2    4    7
                         i              j
          5    3    1    9    8    2    4    7
                         i              j
          5    3    1    4    8    2    9    7   after scoapingA[i] & A[j]
                              i    j
          5    3    1    4    8    2    9    7
                              i    j
          5    3    1    4    2    8    9    7   abter scoapping A[i] & A[j]
                              j    i
          5    3    1    4    2    8    9    7
          2    3    1    4    5    8    9    7   abter scoapping A[l] & A[j]
                         s-1  s    s+1
Subarray       i         j
Pivot     2    3    1    4
               i    j
          2    3    1    4
               i    j
          2    1    3    4   A[i] & A[j] swap
               j    i
          2    1    3    4
          1    2    3    4   swap A[l] & A[j]
Subarray
          1
                Subarray  i j
                    3    4
                    j    i
                    3    4
                         4
```



```
      i   j
$\underline{8}$   9   7
      i   j
$\underline{8}$   7   9
      j   i
$\underline{8}$   7   9

7   $\underline{8}$   9   swap A[j] & A[l]

7

          9
```

Example of quicksort

Quicksort efficiency analysis If all the splits happen in the middle of corresponding subarrays, we will have the best case. The no. of key comparisons in the best case will satisfy the recurrence

$$C_{best}(n) = 2 \cdot C_{best}(n/2) + n \quad \text{for } n>1, \; C_{best}(1) = 0.$$

According to the master theorem, $C_{best}(n) \in \Theta(n \log_2 n)$; solving it exactly for $n = 2^k$ yields $C_{best}(n) = n \log_2 n$.

```
 0    1    2    3    4     5     6    7
$\underline{15}$   12   13   11   20    18    22   14
          (7)     ↓split

apply      $\underline{14}$   12  13   11   15     18   22   20
recursion       (3)  ↓split
to left      0   1   2   3           5   ↓split (2)
subarray    11   12  13  14                6    7
                ↓split               $\underline{18}$   22   20   apply recursion
(2)  0       1   2                                         to right subarray
    $\underline{11}$      12  13                       ↓split
           ↓split
           1   2                    (1)  6    7
(1)  $\underline{12}$   13                          20   22
```

Total no. of comparisons = 7+3+2+2+1+1=16.

no. of comparisons in each split is n−1 i.e., except pivot, do the comparisons for remaining elements.

If the split is unbalanced, too many items are at one side and too few on the other side. which results in worst case analysis. for example,

0 1 2 3 4 5 6 7
$\underline{1}$ 2 3 4 5 6 7 8

↓ split

0 1 2 3 4 5 6 7
1 2 3 4 5 6 7 8 ⑦ no. of comparisons

↓ split

1 2 3 4 5 6 7
$\underline{2}$ 3 4 5 6 7 8 ⑥

↓ split

2 3 4 5 6 7
$\underline{3}$ 4 5 6 7 8 ⑤

⋮ ⋮

6 7
$\underline{7}$ 8 ① no. of comparisons

Total no. of comparisons = 1+2+3+4+5+6+7

= 7*(7+1)/2

= 28 i.e., n.(n-1)/2

$C_{worst}(n) = O(n^2)$.

In other Extreme, If the divisions are as even as possible, for example.

0 1 2 3 4 5 6 7
$\underline{4}$ 1 3 2 7 6 5 8

↓ split

0 1 2 3 4 5 6 7
2 1 3 4 7 6 5 8 ⑦

↓ split ↓ split

② 0 1 2   4 5 6 7
1 $\underline{2}$ 3   5 6 $\underline{7}$ 8 ③

↓ split

4 5
$\underline{5}$ 6 ①

Total no. of comparisons $= 7+2+3+1$

$= 13$ (less than half of worst case)

In the average case,

→ If $n=16$, what or how many comparisons should be made in the best & worst case using Quicksort.

	worst case	Best case	Average case
Quicksort	$O(n^2)$	$O(n\log n)$	$O(n\log n)$ ✓
Insertion sort	$O(n^2)$	$O(n)$ ✓	$O(n^2)$

Let $C_{avg}(n)$ be the average no. of key comparisons made by quicksort on a randomly ordered array of size n. Assuming that the partition split can happen in each position s $(0 \le s \le n-1)$ with the same probability $1/n$, we get the following recurrence relation.

$$C_{avg}(n) = \overset{(n+1)+}{\frac{1}{n}} \sum_{s=0}^{n-1} \left[\cancel{(n+1)} + C_{avg}(s) + C_{avg}(n-1-s)\right]$$

for $n>1$,

$C_{avg}(0) = 0$, $C_{avg}(1) = 0$.

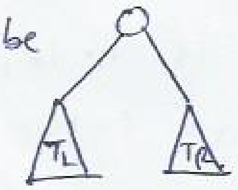
$$C_{avg}(n) \simeq 2n\log n$$

$n+1$ – The no. of element comparisons required by partition on its first call.

Binary Tree Traversals & Related Properties

A binary tree T is defined as a finite set of nodes that is either empty or consists of a root and two disjoint binary trees T_L and T_R called, respectively, the left and right subtree of the root.

Many problems about binary trees can be solved by applying the divide-and-conquer technique.

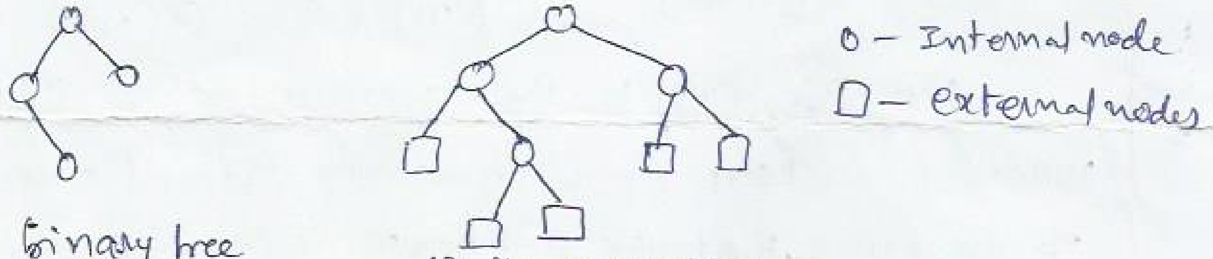


binary tree

Ex: Computing the height of a binary tree.

→ height is defined as the length of the longest path from the root to a leaf.

→



a) binary tree

b) its extension.

*) The extra nodes are called external nodes (denoted by x); the original nodes are called internal nodes (denoted by n).

how many external nodes an extended binary tree with n internal nodes can have?

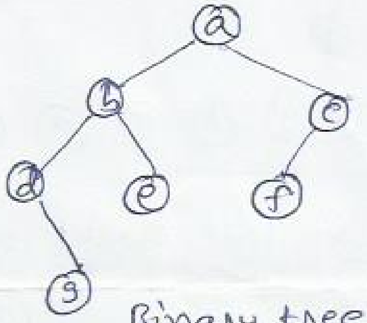
Ans: $x = n+1$.

→ Full binary tree every node has either zero or two children; for a full binary tree, n & x denote the numbers of parental nodes and leaves, respectively.

The most important divide-and-conquer algorithms for binary trees are the three classic traversals: preorder, inorder, and postorder. All three traversals visit nodes of a binary tree recursively i.e., by visiting the tree's root and its left and right subtrees. They differ just by the timing of the root's visit:

Preorder traversal: the root is visited before the left and right subtrees are visited.

```
preorder(root){
    if (root==NULL) return;
   print(root->data);
   preorder(~~node~~ root->left);
   preorder(root->right);
 }
```



Binary tree

preorder: a, b, d, g, e, c, f

Inorder traversal: The root is visited after visiting its left subtree but before visiting the right subtree.

```
inorder(~~node~~ root){
if (~~node~~ root==NULL) return;
 inorder(root->left);
   print (root->data);
} inorder (root->right);
```

inorder: d, g, b, e, a, f, c.

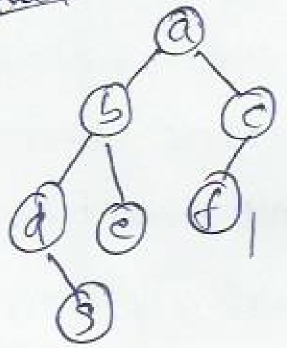
Post order traversal: The root is visited after visiting the left and right subtrees.

```
Postorder(root){
 if (root==NULL) return;
 postorder(root -> left);
 postorder(root -> right);
 print(root ->data);
}.
```

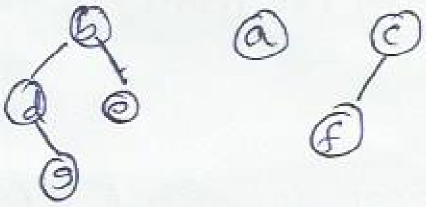
Postorder: g, d, e, b, f, c, a.

~~Inorder~~

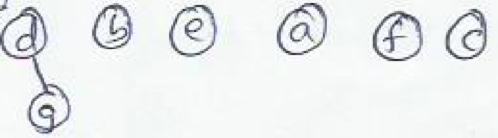
Ex:1



Inorder: left root right

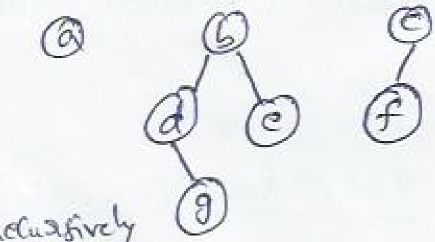


recursively

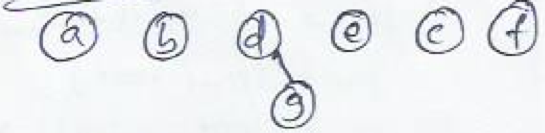


d g b e a f c

Preorder root left right

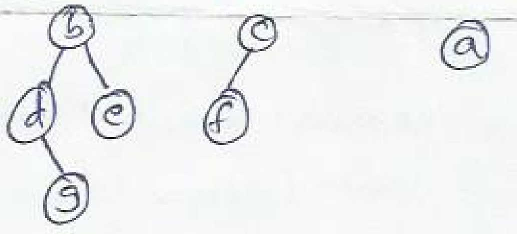


recursively

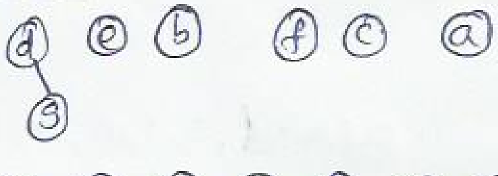


a b d g e c f

Postorder left right root



recursively

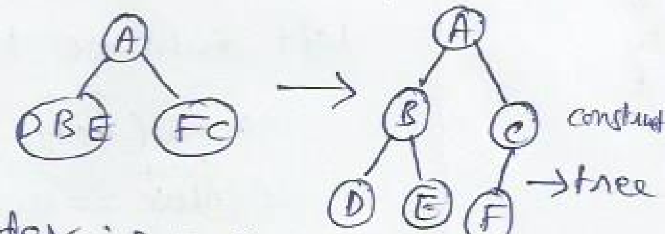


g d e b f c a

→ find the post order traversal from the following. (1)

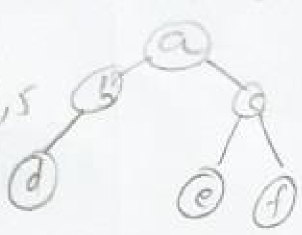
Inorder: D B E A F C

Preorder: A B D E C F



Postorder: D E B F C A

→ ~~find~~ the a. preorder b. inorder c. postorder



→ Draw a binary tree

(2) inorder: 9, 3, 1, 0, 4, 2, 7, 6, 8, 5

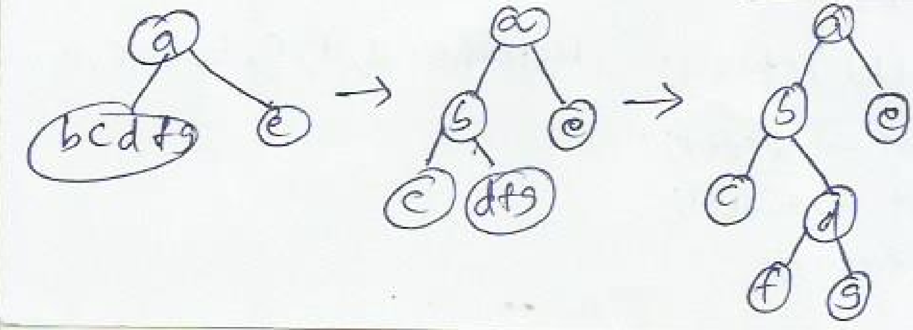
Postorder: 9, 1, 4, 0, 3, 6, 7, 5, 8, 2

→ find the inorder traversal

(3) from the following

Preorder: a b c d f g e

Post order: c f g d b e a



inorder: c b f d g a e

0 1 2 3 4 5 6 7 8

pivot $\underline{65}$ $\overset{i}{70}$ 75 80 85 60 55 50 $\overset{j}{45}$

$\underline{65}$ $\overset{i}{45}$ 75 80 85 60 55 50 $\overset{j}{70}$ swap A[i] & A[j] i>j ?

$\underline{65}$ 45 $\overset{i}{75}$ 80 85 60 55 $\overset{j}{50}$ 70

$\underline{65}$ 45 $\overset{i}{50}$ 80 85 60 55 $\overset{j}{75}$ 70 swap A[i] & A[j] i>j?

$\underline{65}$ 45 50 $\overset{i}{80}$ 85 60 $\overset{j}{55}$ 75 70

$\underline{65}$ 45 50 $\overset{i}{55}$ 85 60 $\overset{j}{80}$ 75 70 swap A[i] & A[j]. i>j?

$\underline{65}$ 45 50 55 $\overset{i}{85}$ $\overset{j}{60}$ 80 75 70

$\underline{65}$ 45 50 55 $\overset{i}{60}$ $\overset{j}{85}$ 80 75 70 swap A[i] & A[j] i>j?

$\underline{65}$ 45 50 55 $\overset{j}{60}$ $\overset{i}{85}$ 80 75 70

$\underset{l}{60}$ 45 50 $\underset{s-1}{55}$ $\underset{s}{65}$ $\underset{s+1}{85}$ 80 75 $\underset{r}{70}$ swap A[j] & A[l]

i l 0 1 2 3 r j

pivot $\underline{60}$ 45 50 55

$\underline{60}$ $\overset{i}{45}$ 50 $\overset{j}{55}$

$\underline{60}$ 45 $\overset{i}{50}$ $\overset{j}{55}$

| $\underline{60}$ 45 50 $\overset{ij}{55}$ | swap a[l], a[j] ← Extremity condition

p l 0 1 2 r

$\underline{55}$ 45 50 $\underset{s}{60}$ 65

l $\overset{p\ i\times}{55}$ $\overset{i}{45}$ 50 j

| $\underline{55}$ 45 $\overset{ij}{50}$ |

swap a[l], a[j]

| $\underline{50}$ 45 | 55

45 50 55 swap a[l], a[i]

$\overset{i\ 5}{l}$ 6 7 $\underset{r}{8}$ j

p $\underline{85}$ 80 75 70

85 $\overset{i\times}{80}$ $\overset{i\times}{75}$ $\overset{i\times}{70}$ j

| 85 80 75 $\overset{ij}{70}$ |

Average case efficiency on quicksort ⑨

$$C_{avg}(n) \overset{(n+1)+}{=} \frac{1}{n}\sum_{s=0}^{n-1}\left[C_{avg}(s) + C_{avg}(n-1-s)\right]$$

$C_{avg}(0) = C_{avg}(1) = 0$ where $n+1$ - no. of element comparisons required by partition on its first call.

$$C_{avg}(n) = (n+1) + \frac{1}{n}\sum_{s=0}^{n-1}\left[C_{avg}(s) + C_{avg}(n-1-s)\right] \rightarrow ①$$

multiply both sides of ① by n, we get

$$n\,C_{avg}(n) = n(n+1) + 2\left[C_{avg}(0) + \cdots + C_{avg}(n-1)\right] \rightarrow ②$$

Replace n by $n-1$ in ② gives

$$(n-1)\,C_{avg}(n-1) = n(n-1) + 2\left[C_{avg}(0) + \cdots + C_{avg}(n-2)\right]$$

Subtract this from ②, we get

$$n\,C_{avg}(n) - (n-1)\,C_{avg}(n-1) = 2n + 2\,C_{avg}(n-1).$$

$$\boxed{\begin{aligned} n\,C_{avg}(n) &= 2n + C_{avg}(n-1)\left[2+n-1\right] \\ &= (n+1)\,C_{avg}(n-1) + 2n \\ \frac{C_{avg}(n)}{(n+1)} &= \frac{C_{avg}(n-1)}{n} + \frac{2}{n+1}. \end{aligned}}$$

(or)

divide both sides by $n(n+1)$

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

Repeatedly using this equation to substitute for $C_{avg}(n-1)$, $C_{avg}(n-2)$, ... we get

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

$$= \frac{C_{avg}(n-3)}{n-2} + \frac{2}{n-1} + \frac{2}{n} + \frac{2}{n+1} \quad ③$$

$\vdots$

$\left[\because \sum_{n=1}^{\infty}\frac{1}{n} = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \cdots + \frac{1}{n}\right]$ series stop at $n=4$.

$$= \frac{C_{avg}(1)}{2} + 2\sum_{s=3}^{n+1}\frac{1}{s}$$

$$= 2\sum_{s=3}^{n+1}\frac{1}{s}. \longrightarrow ②$$

Since

$$\sum_{s=3}^{n+1}\frac{1}{s} \le \int_{2}^{n+1}\frac{1}{x}\,dx = \log_e(n+1) - \log_e 2.$$

① yields

multiplication of large integers & Strassen's matrix multiplicatio

The main purpose of introducing these algorithms is to decrease the total number of multiplications performed by increasing the number of additions.

multiplication of large integers (multiplying two numbers)

multiplication of integers that are over 100 decimal digits long need more sophisticated algorithms. multiplying two n-digit integers, each of the n digits of first number is multiplied by each of the n digits of the second number for the total of n^2 digit multiplications.

is it possible to design an algorithm with fewer than n^2 digit multiplications ?

Ans yes, by using divide-and-conquer we can achieve this.

for ex, let us start with a case of two-digit integers, say, 35 and 23. These numbers can be represented as follows:

$$35 = 3\cdot10^1 + 5\cdot10^0 \quad \text{and} \quad 23 = 2\cdot10^1 + 3\cdot10^0.$$

Now let us multiply them:

$$35 * 23 = (3\cdot10^1 + 5\cdot10^0) * (2\cdot10^1 + 3\cdot10^0).$$

$$= \underbrace{(3\cdot2)}_{①} 10^2 + (\underbrace{3\cdot3}_{②} + \underbrace{5\cdot2}_{③}) 10^1 + \underbrace{(5\cdot3)}_{④} 10^0.$$

$$= 600 + 190 + 15 = 805.$$

number of multiplications are four. we can compute the middle term with just one digit multiplication. It can be computed as follows:

$a = a_1 a_0$ & $b = b_1 b_0$

$$c = a * b$$

$$= (a_1 \cdot 10^1 + a_0 \cdot 10^0) * (b_1 \cdot 10^1 + b_0 \cdot 10^0)$$

$$= c_2 10^2 + c_1 10^1 + c_0 .$$

$$= \underbrace{(a_1 \cdot b_1) 10^2}_{c_2} + \underbrace{(a_1 \cdot b_0 + a_0 \cdot b_1) 10^1}_{c_1} + \underbrace{a_0 \cdot b_0}_{c_0}$$

we have to make this as one multiplication

where

$c_2 = a_1 \cdot b_1$ is the product of their first digits,

$c_0 = a_0 \cdot b_0$ is the product of their second digits,

$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$ is the product of the sum of the a's digits and the sum of the b's digits minus the sum of c_2 and c_0.

i.e.,

$$c_1 = (3+5) * (2+3) - (6+15)$$

$$= 40 - 21$$

$$= 19$$

apply this trick to multiply two n-digit integers a and b where n is a positive even number. apply divide-and-conquer strategy, i.e., denote the first half of the a's digits by a_1 and the second half by a_0; for b, the notations are b_1 and b_0, respectively.

for ex, $a = 123456$ & $b = 357236$

$$a = \underbrace{123}_{a_1} \cdot 10^3 + \underbrace{456}_{a_0} \qquad b = \underbrace{357}_{b_1} \cdot 10^3 + \underbrace{236}_{b_0}$$

$$c = a * b$$

$$= (123 \cdot 10^3 + 456) * (357 \cdot 10^3 + 236)$$

$$= (123 \cdot 357) 10^6 + (123 \cdot 236 + 456 \cdot 357) 10^3 + 456 \cdot 236$$

$$c = a * b$$

$$= (a_1 10^{n/2} + a_0) * (b_1 10^{n/2} + b_0)$$

$$= (a_1 \cdot b_1) 10^n + (a_1 \cdot b_0 + a_0 \cdot b_1) 10^{n/2} + (a_0 \cdot b_0)$$

$$= c_2 10^n + c_1 10^{n/2} + c_0.$$

where,

$c_2 = a_1 * b_1$ is the product of their first halves;

$c_0 = a_0 * b_0$ is the product of their second halves;

$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$ is the product of the sums of the a's halves and the sum of the b's halves minus the sum of c_2 and c_0.

How many multiplications does this algorithm make? Since multiplication of n-digit numbers requires three multiplications of n/2-digit numbers, the recurrence for the no. of multiplications $M(n)$ will be

$$M(n) = 3M(n/2) \quad \text{for } n > 1,$$

$$M(1) = 1 \quad \text{for } n = 1.$$

$M(n) = 3M(n/2)$ $\quad M(n/2) = 3 \cdot M(n/4)$ $\quad M(n/4) = 3 \cdot M(n/8)$

. . . .

apply backward substitution, we get

$$M(n) = 3 \cdot M(n/2)$$

$$= 3 \cdot 3\, M(n/4)$$

$$= 3^2 \cdot 3\, M(n/8)$$

$$\vdots$$

$$= 3^k\, M(n/2^k)$$

let $n = 2^k \Rightarrow k = \log_2 n$

$$M(2^k) = 3^{\log_2 n} M(1)$$

$$\left[\because a^{\log_b c} = c^{\log_b a}\right]$$

$$\simeq n^{\log_2 3}$$

$$\simeq n^{1.585}$$

→ compute 2101 * 1130 by applying the divide-and-conquer.

Strassen's matrix multiplication (multiplying two square matrices)

$$A = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix}_{2\times2} \qquad B = \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}_{2\times2}$$

$$C = A * B$$

$$= \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} * \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$

$$= \begin{bmatrix} a_{00}\cdot b_{00} + a_{01}\cdot b_{10} & a_{00}\cdot b_{01} + a_{01}\cdot b_{11} \\ a_{10}\cdot b_{00} + a_{10}\cdot b_{10} & a_{10}\cdot b_{01} + a_{11}\cdot b_{11} \end{bmatrix}$$

Brute - force algorithm need 8 multiplications & 4 additions. reduce no. of multiplications to 7 & by increasing additions / subtractions (Proposed by V. Strassen).

$$= \begin{bmatrix} m_1 + m_2 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}.$$

where,

$\overset{p}{m_1} = (a_{00}+a_{11}) * (b_{00}+b_{11})$ $\qquad \overset{q}{m_2} = (a_{10}+a_{11}) * b_{00}.$

$\overset{r}{m_3} = a_{00} * (b_{01}-b_{11})$ $\qquad \overset{s}{m_4} = a_{11} * (b_{10}-b_{00}).$

$\overset{t}{m_5} = (a_{00}+a_{01}) * b_{11}$ $\qquad \overset{u}{m_6} = (a_{10}-a_{00}) * (b_{00}+b_{01})$

$\overset{v}{m_7} = (a_{01}-a_{11}) * (b_{10}+b_{11})$

$C_{00} =$ ~~P+~~ $m_1 + m_4 - m_5 + m_7$ & $p+s-t+v$

$C_{01} = m_3 + m_5$ & $r+t$

$C_{10} = m_2 + m_4$ & $q+s$

$C_{11} = m_1 + m_3$ ~~+~~ $- m_2 + m_6$ & $p+r-q+u$

Apply Strassen's algorithm to compute:

$$\begin{bmatrix} 1 & 2 & 1 & 1 \\ 0 & 3 & 2 & 4 \\ 0 & 1 & 1 & 1 \\ 5 & 0 & 1 & 0 \end{bmatrix} * \begin{bmatrix} 2 & 1 & 0 & 2 \\ 1 & 2 & 1 & 1 \\ 0 & 3 & 2 & 1 \\ 4 & 0 & 0 & 4 \end{bmatrix}$$

exiting the recursion when n=2, compute the product of 2-by-2 matrices by brute-force algorithm.

let A and B ~~be~~ bae two n-by-n matrices where n is a Power of 2. we can divide A, B, and their product C into four n/2-by-n/2 submatrices each as follows

$$C = A * B$$

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} * \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix}$$

$$= \begin{bmatrix} A_{00} \cdot B_{00} + A_{01} \cdot B_{10} & A_{00} \cdot B_{01} + A_{01} \cdot B_{11} \\ A_{10} \cdot B_{00} + A_{11} \cdot B_{10} & A_{10} \cdot B_{01} + A_{11} \cdot B_{11} \end{bmatrix}$$

C_{00} can be computed as either $A_{00} \cdot B_{00} + A_{01} \cdot B_{10}$ or as $M_1 + M_4 - M_5 + M_7$ where $M_1, M_4, M_5, \& M_7$ are found by strassen's formulas. If the seven products of n/2-by-n/2 matrices are computed recursively by the same method, we have strassen's algorithm for matrix multiplication.

asymptotic efficiency If M(n) is the no. of multiplications made by strassen's algorithm in multiplying two n-by-n matrices (where n is a Power of 2), we get the following recurrence relation for it:

$$M(n) = 7M(n/2) \quad \text{for } n > 1,$$
$$M(1) = 1 \quad n = 1.$$

$M(n/2) = 7 \cdot M(n/4)$ $M(n/4) = 7 \cdot M(n/8)$ $M(n/8) = 7 \cdot M(n/16)$.

. . . .

Apply Backward substitutions, we get.

$$M(n) = 7M(n/2)$$
$$= 7 \cdot 7M(n/4)$$
$$= 7^2 \cdot 7M(n/8)$$
$$\vdots$$
$$= 7^k M(n/2^k)$$

let $n = 2^k$ $\therefore k = \log_2 n$.

$$M(n) = 7^{\log_2 n} M(1)$$

$a=7$ $b=2$ $n=c$ $\left[\because a^{\log_b c} = c^{\log_b a}\right]$

$$\therefore M(n) = n^{\log_7 2} \simeq n^{2.807}.$$

Brute-force algorithm takes n^3

Transform - and - Conquer

It is a two-stage procedure. They are

1: transformation stage - The problem's instance is modified to be more suitable to solution.

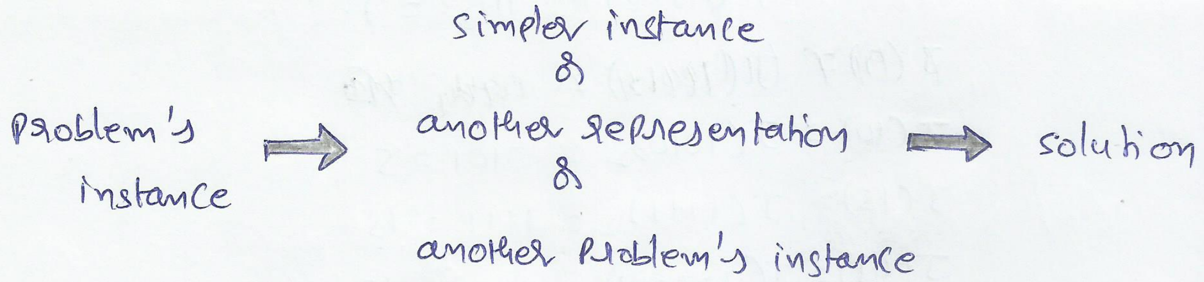
2. conquering stage - It is solved.

These are three major variations of this. They are

(i) instance simplification: Transformation to a simpler instance of the same problem. Ex: Presorting, AVL trees

(ii) representation change: Transformation to a different representation of the same instance. Ex: Horner's rule

(iii) Problem reduction: Transformation to an instance of a different problem for which an algorithm is already available. Ex: complexity theory problems.



Transform - and - Conquer strategy.

Presorting It is an old idea in computer science. Many problems about a lists & arrays are easier to answer if the list is sorted. obviously, the time efficiency of algorithms that involve sorting may depend on the efficiency of the sorting algorithm being used.

Are there faster sorting algorithms?

Ans: yes, comparison-based sorting algorithm mergesort takes $O(n\log n)$ time in the worst case.

examples that illustrate Presorting are

1. checking element uniqueness in an array.
2. Computing a mode.
3. searching Problem.

1. checking element uniqueness in an array: The brute-force algorithm compared pairs of the array's elements until either two equal elements were found or no more pairs were left. Its worst-case efficiency was $O(n^2)$.

Alternatively, we can sort the array first and then check only its consecutive elements: if the array has equal elements, a pair of them must be next to each other and vice versa. if A[i] == A[i+1] //array is duplicated // otherwise it's unique

runtime: α time spent on sorting + time spent on checking consecutive elements.

$$T(n) = T_{sort}(n) + T_{scan}(n)$$

$$\in O(n\log n) + O(n) = O(n\log n).$$

2. Computing a mode:

A mode is a value that occurs most often in a given list of numbers.

Ex: 5, 1, 5, 7, 6, 5, 7 mode is 5.

→ If several different values occur most often, any of them can be considered a mode.

→ The brute-force approach to computing a mode would scan the list and compute the frequencies of all its distinct values, then find the value with the largest frequency

worst case Time complexity = $O(n^2)$

→ alternate solution: First sort the input. Then all equal values will be adjacent to each other. To compute the mode, all we need to do is to find the longest run of adjacent equal values in the sorted array.

run time $\doteq$ timespent on sorting + algorithm's time

$= \Theta(n \log n)$

3. Searching problem:

Consider the problem of searching for a given value v in a given array of n sortable items.

→ The brute-force solution

→ sequential search - which needs n comparisons in the worst case.

→ If the array is sorted first, we can then apply binary search, which requires only $\lfloor \log_2 n \rfloor + 1$ comparisons in the worst case.

→ Assuming the most efficient n logn sort, the total running time of such a searching algorithm in the worst case will be

$$T(n) = T_{sort}(n) + T_{search}(n)$$
$$= \Theta(n \log n) + \Theta(\log n)$$
$$= \Theta(n \log n).$$ - which is inferior to sequential search.

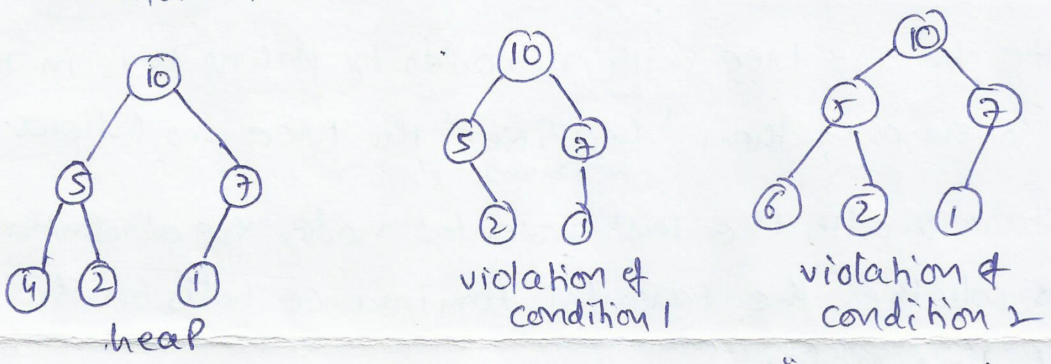
Of course, if we are to search in the same list more than once, the time spent on sorting might well be justified.

Heaps and Heapsort

Definition of heap: A heap can be defined as a binary tree with keys assigned to its nodes (one key per node) provided the following two conditions are met:

(i) The tree's shape requirement — The binary tree is essentially complete, except in the last level, i.e., only some right most leaves may be missing.

(ii) The parental dominance requirement — The key at each node is greater than & equal to the keys at its children.

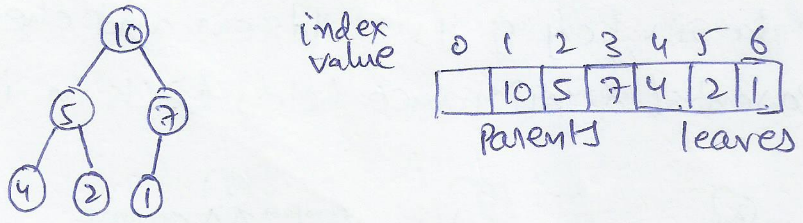


heap

violation of condition 1

violation of condition 2

Illustration of the definition of "heap": only the left most tree is a heap.



Heap and its array representation.

Properties of heaps

1. There exists exactly one essentially complete binary tree with n nodes. Its height is equal to $\lfloor \log_2 n \rfloor$.

2. The root of a heap always contains its largest element.

3. Descendant nodes of a tree is also a heap.

4. A heap is implemented as an array in such a way that

a) the parental node keys will be in the first $\lfloor n/2 \rfloor$

Positions of the array, while the leaf keys will occupy the last $\lceil n/2 \rceil$ positions;

b) The children of a key in array's parental position i will be in position $\overset{left}{2i}$ & $\overset{right}{2i+1}$.

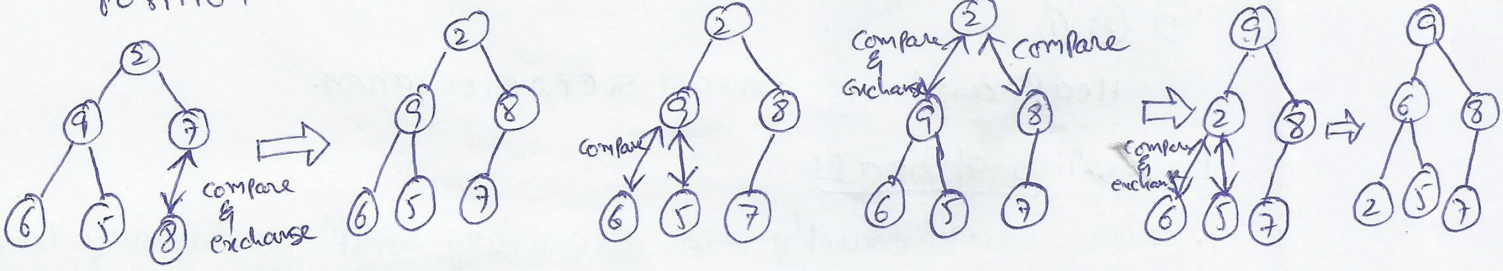
How can we construct a heap for a given list of keys? Ans Two ways. They are

1. Bottom-up heap construction.
2. Top-down heap construction.

1. Bottom-up heap construction: It initializes the essentially complete binary tree with n nodes by placing keys in the order given and then "$\underset{rearrange}{\text{heapifies}}$" the tree as follows.

1. Starting with the last parental node, the algorithm checks whether the parental dominance holds for the key at this node.

2. If it does not, the algorithm exchanges the node's key K with the larger key of its children and checks whether the parental dominance holds for K in its new position.



Bottom-up construction of a heap for the list 2, 9, 7, 6, 5, 8.

• This process continues until the parental dominance requirement ~~holds~~ for K in its new is satisfied.

3. After completing the "heapification" of the subtree rooted at the current parental node, the algorithm proceeds to do the same for the node's immediate predecessor.

4. The algorithm stops after this is done for the tree's root.

```
Algorithm HeapBottomUP(H[1...n])
//constructs a heap from the elements of a given array.
// by the bottom-up algorithm.
// Input: An array H[1...n] of orderable items
// output: A heap H[1...n].
  for i ← ⌊n/2⌋ down to 1 do
      k ← i;  v ← H[k]
      heap ← false
      while not heap and 2*k ≤ n do
            j ← 2*k.
        if j<n  // there are two children.
            if H[j] < H[j+1]    j ← j+1
        if v ≥ H[j]
            heap ← true
        else  H[K] ← H[j];  k ← j
    H[k] ← v
```

The total no. of key comparisons in the worst case will b

$$C_{worst}(n) = \sum_{i=0}^{h-1} \sum_{\text{level } i \text{ keys}} 2(h-i) = \sum_{i=0}^{h-1} 2(h-i)2^i = 2(n-\log_2(n+1)$$

Thus, with bottom-up algorithm, a heap of size n can be constructed with fewer than 2n comparisons.

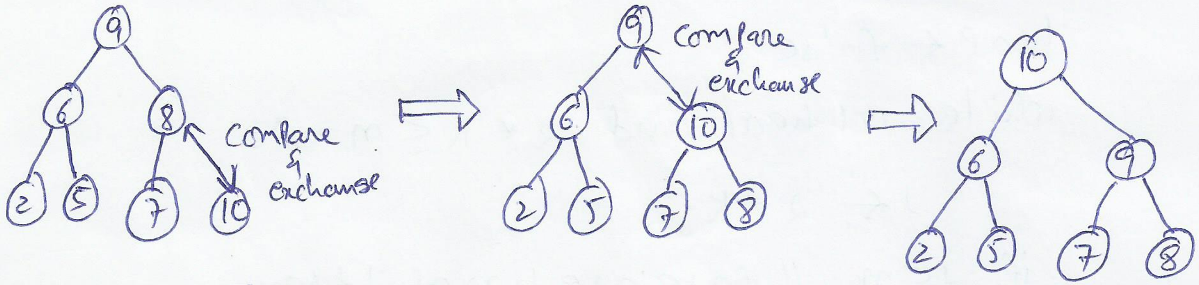
2. <u>Top-down heap construction</u>: it is less efficient & it constructs a heap by successive insertions of a new key into a previously constructed heap.

how can we insert a new key K into a heap?

1. attach a new node with key K in it $\overset{\text{(heap)}}{}$ after the last $\overset{\text{leaf}}{}$ of the existing heap. Then shift k up to its appropriate place in the new heap as follows.

compare k with its parent's key: if the parent is greater than & equal to k, stop; otherwise, swap these two keys and compare k with its new parent. This swapping continues until k is not greater than its last parent & it reaches the root.

The height of a heap with n nodes is about $\log_2 n$, the time efficiency of insertion is in O(logn).



Insert a key 10 into the heap constructed. The new key is shifted up via a swap with its parent until it is not larger than its parent (& is in the root).

How can we delete an item from a heap?

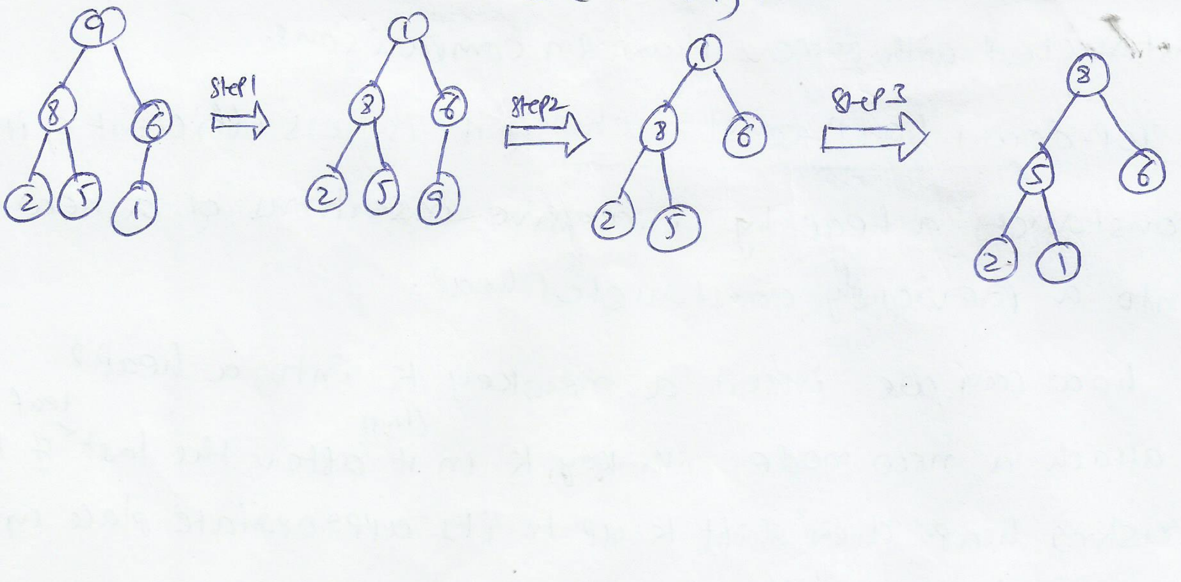
maximum key deletion from a heap

step1: Exchange the root's key with the last key K of the heap.

step2: Decrease the heap's size by 1.

step3: "Heapify" the smaller tree by shifting k down the tree exactly in the same way we did in the bottom-up heap construction.

Time efficiency is O(logn).



Heapsort This is a two-stage algorithm that works as follows.

Stage1: (heap construction) Construct a heap for a given array.

Stage2: (maximum deletions) Apply the root-deletion operation n-1 times to the remaining heap.

Stage1 (heap construction)

2 9 7 6 5 8

2 9 8 6 5 7

2 9 8 6 5 7

9 2 8 6 5 7

9 6 8 2 5 7

Stage2 (maximum deletions)

9 6 8 2 5 7

7 6 8 2 5 | 9 delete

heapify

8 6 7 2 5

5 6 7 2 | 8 delete

heapify

7 6 5 2

2 6 5 | 7 delete

heapify

6 2 5

5 2 | 6 delete

5 2

2 | 5 delete

2 delete (elements are deleted in decreasing order) high-low

Sorting the array 2, 9, 7, 6, 5, 8 by heapsort.

Time efficiency for stage1 – $O(n)$.

stage2 – $n \log_2 n$.

$$C(n) = O(n) + O(n \log n)$$

$$= O(n \log n).$$

Q2. 3, 2, 4, 1, 6, 5 apply heapsort.

3. 1, 8, 6, 5, 3, 7, 4 〃

4. H, E, A, P, S, O, R, T 〃

5. 3, 4, 2, 1, 5, 6 apply heapsort

It is a good example for representation change technique

Horner's Rule It is the problem of computing the value of a polynomial $P(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$ at $\rightarrow ①$ a given point x and computing x^n.

rewrite the polynomial $P(x) = (\cdots(a_n x + a_{n-1})x + \cdots)x + a_0$. $\rightarrow ②$

For ex, for the polynomial

$$P(x) = 2x^4 - x^3 + 3x^2 + x - 5, \text{ we set.}$$

$$\begin{aligned} P(x) &= 2x^4 - x^3 + 3x^2 + x - 5 \\ &= x(2x^3 - x^2 + 3x + 1) - 5 \\ &= x(x(2x^2 - x + 3) + 1) - 5 \\ &= x(x(x(2x-1)+3)+1) - 5 \rightarrow ③ \end{aligned}$$

A new formula ② can be obtained from ① by successively taking x as a common factor in the remaining polynomials of diminishing degrees.

we can evaluate the polynomial with a two-row table. The first row contains the polynomial's coefficients (including all the coefficients equal to zero, if any) listed from the highest a_n to the lowest a_0. Except for its first entry, which is a_n, the second row is filled left to right as follows:

the next entry is computed as the x's value times the last entry in the second row plus the next coefficient from the first row. The final entry computed in this fashion is the value being sought.

Ex1: Evaluate $P(x) = 2x^4 - x^3 + 3x^2 + x - 5$ at $x = 3$.

coefficients	2	-1	3	1	-5
$x=3$	2	$3\cdot2+(-1)=5$	$3\cdot5+3=18$	$3\cdot18+1=55$	$3\cdot55+(-5)=160$

so $P(3) = 160$.

$3\cdot2 + (-1) = 5$ is the value of $2x-1$ at $x=3$.
$3\cdot5 + 3 = 18$ " $x(2x-1)+3$ at $x=3$.
$3\cdot18 + 1 = 55$ " $x(x(2x-1)+3)+1$ at $x=3$
finally $3\cdot55 + (-5) = 160$ " $x(x(x(2x-1)+3)+1)-5 = P(x)$ at $x=3$

↓ Polynomial coefficients

```
ALGORITHM Horner(P[0...n], x)
// Evaluates a polynomial at a given point by Horner's rule.
// Input: An array P[0...n] of coefficients of a polynomial of
//        degree n (stored from the lowest to the highest) and a
//        number x.
// Output: The value of the polynomial at x.
    K ← P[n].
    for i ← n−1 downto 0 do
        K ← x * K + P[i]
    return K.
```

The number of multiplications and the number of additions are given by the same sum:

$$M(n) = A(n) = \sum_{i=0}^{n-1} 1 = n.$$

Exercise 1. Apply Horner's rule to evaluate the polynomial

$$P(x) = 3x^5 + 2x^4 - 5x^3 + x^2 + 7 \text{ at } x = -3$$