# Divide and Conquer Algorithms

## Sorting

Sorting is among the most basic problems in algorithm design. We are given a sequence of items,each associated with a given key value. The problem is to permute the items so that they are in increasing (or decreasing) order by key. Sorting is important because it is often the first step in more complex algorithms. Sorting algorithms are usually divided into two classes, internal sorting algorithms, which assume that data is stored in an array in main memory, and external sorting algorithm, which assume that data is stored on disk or some other device that is best accessed sequentially. We will only consider internal sorting. Sorting algorithms often have additional properties that are of interest, depending on the application. Here are two important properties.

**In-place:** The algorithm uses no additional array storage, and hence (other than perhaps the system's recursion stack) it is possible to sort very large lists without the need to allocate additional working storage.

**Stable:** A sorting algorithm is stable if two elements that are equal remain in the same relative position after sorting is completed. This is of interest, since in some sorting applications you sort first on one key and then on another. It is nice to know that two items that are equal on the second key, remain sorted on the first key.

## Merge Sort

To sort an array A[l . . r]:

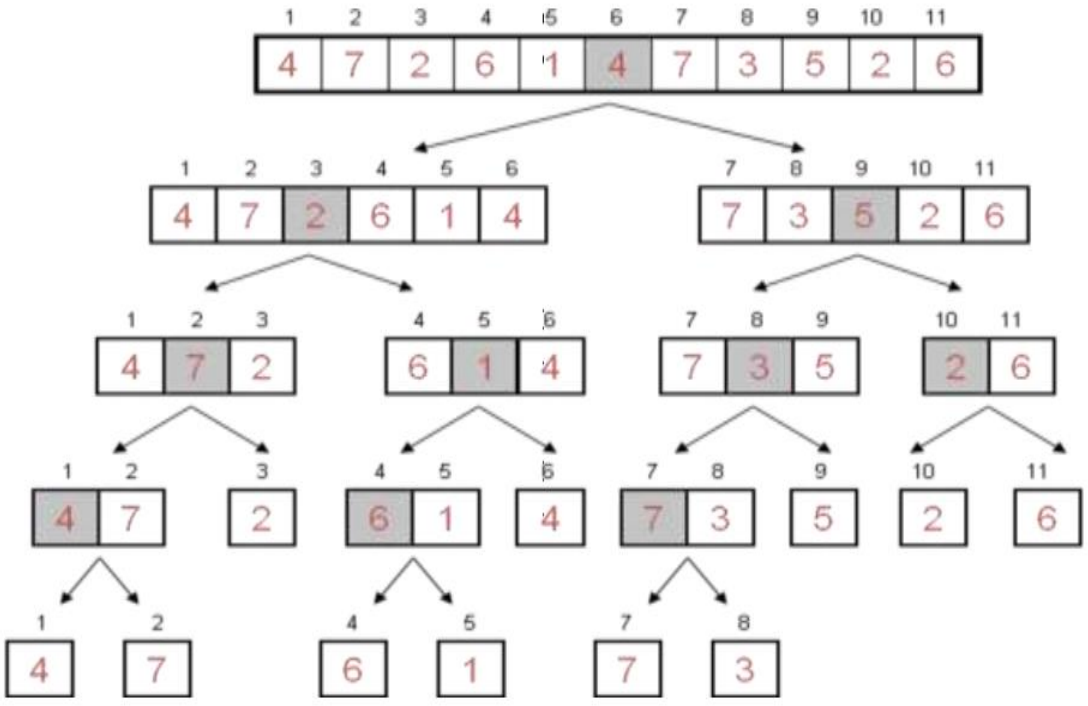
• Divide– Divide the n-element sequence to be sorted into two subsequences of n/2 elements each

Conquer– Sort the subsequences recursively using merge sort. When the size of the sequences is 1 there is nothing more to do

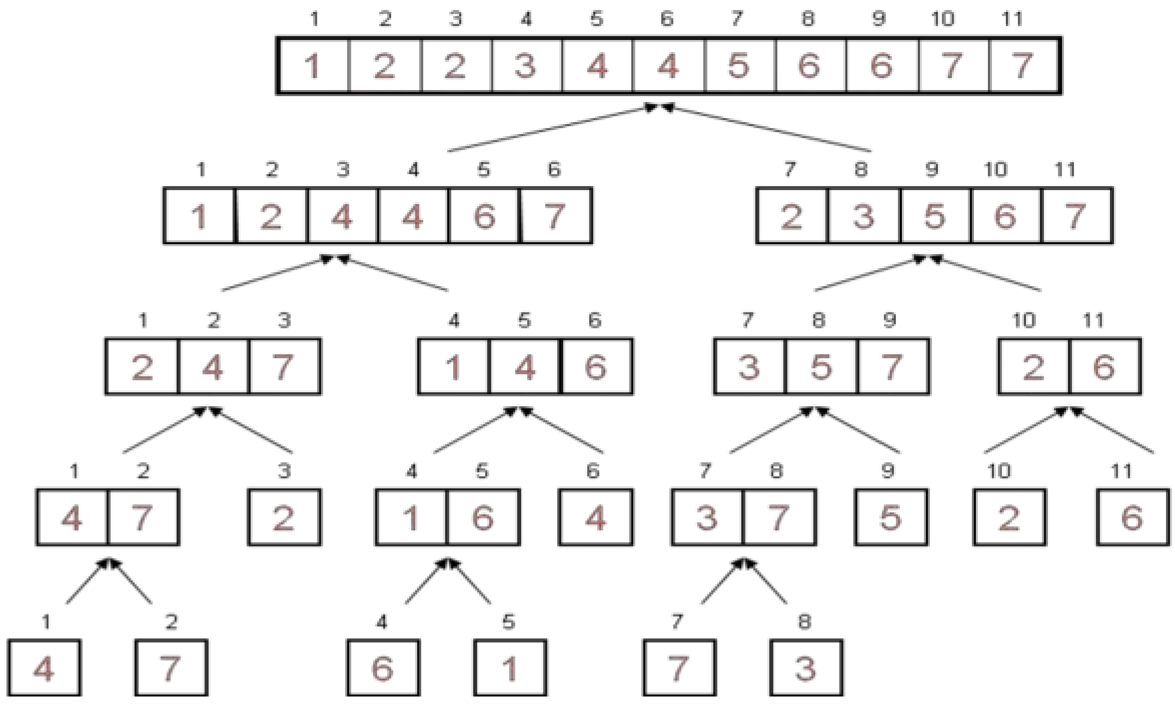
Combine

– Merge the two sorted subsequence

### Divide



## Merging:



## Algorithm:

**MergeSort(A, l, r)**

```
{
If ( l < r)
{
m = ⌊(l + r)/2⌋
MergeSort(A, l, m)
MergeSort(A, m + 1, r)
Merge(A, l, m+1, r)
}
}
```

**Merge(A,B,l,m,r)**

```
{
x=l, y=m;
k=l;
while(x<m && y<r)
{
if(A[x] < A[y])
{
B[k]= A[x];
k++; x++;
}
else
{
B[k] = A[y];
k++; y++;
}
}
 while(x<m)
{
A[k] = A[x];
k++; x++;
} w
hile(y<r)
{
A[k] = A[y];
k++; y++;
}
for(i=l;i<= r; i++)
{
A[i] = B[i]
}
}
```

## Max and Min Finding

Here our problem is to find the minimum and maximum items in a set of n elements.

**Iterative Divide and Conquer Algorithm for finding min-max:**

Main idea behind the algorithm is: if the number of elements is 1 or 2 then max and min are obtained trivially. Otherwise split problem into approximately equal part and solved recursively.

```
MinMax(l,r)
{
if(l = = r)
max = min = A[l];
else if(l = r-1)
{
if(A[l] < A[r])
{
max = A[r]; min = A[l];
}
else
{
max = A[l]; min = A[r];
}
}
else
{
//Divide the problems
mid = (l + r)/2; //integer division
//solve the subproblems
{min,max}=MinMax(l,mid);
{min1,max1}= MinMax(mid +1,r);
//Combine the solutions
if(max1 > max) max = max1;
if(min1 < min) min = min1;
}
}
```

## Analysis:

We can give recurrence relation as below for MinMax algorithm in terms of number of comparisons.

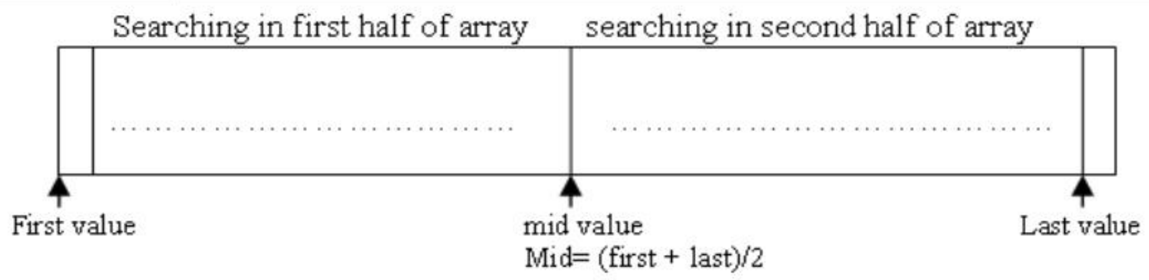
$T(n) = 2T( n / 2 ) + 1$ , if $n>2$

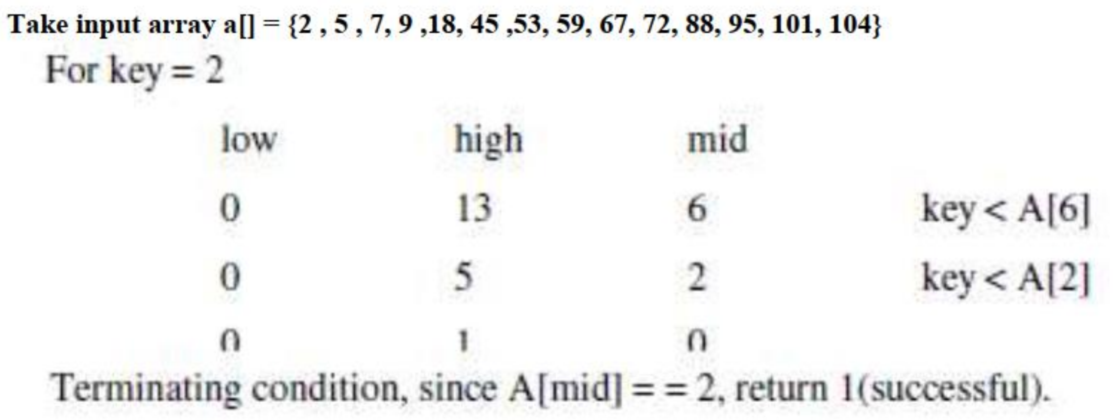
$T(n) = 1$ , if $n \leq 2$

Solving the recurrence by using master method complexity is (case 1) O(n).

## Binary Search

**To find a key in a large array containing keys z[0; 1; : : : ; n-1] in sorted order, we first compare key with z[n/2], and depending on the result we recurse either on the first half of the file, z[0; : : : ; n/2 - 1], or on the second half, z[n/2; : : : ; n-1].**


Searching in first half of array
searching in second half of array
First value
mid value
Mid= (first + last)/2
Last value

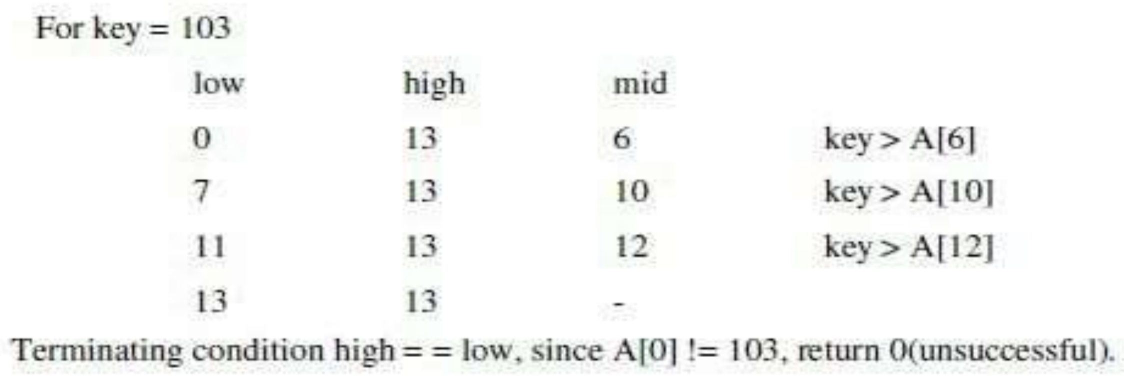


**Take input array a[] = {2 , 5 , 7, 9 ,18, 45 ,53, 59, 67, 72, 88, 95, 101, 104}**

For key = 2

| low | high | mid | |
|---|---|---|---|
| 0 | 13 | 6 | key < A[6] |
| 0 | 5 | 2 | key < A[2] |
| 0 | 1 | 0 | |

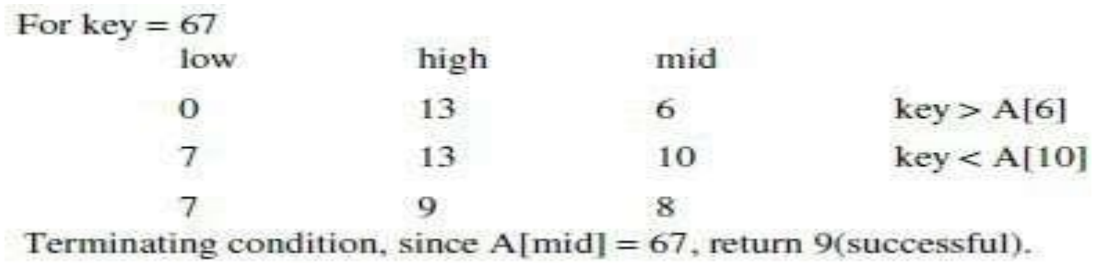
Terminating condition, since A[mid] = = 2, return 1(successful).



For key = 103

| low | high | mid | |
|---|---|---|---|
| 0 | 13 | 6 | key > A[6] |
| 7 | 13 | 10 | key > A[10] |
| 11 | 13 | 12 | key > A[12] |
| 13 | 13 | - | |

Terminating condition high = = low, since A[0] != 103, return 0(unsuccessful).



For key = 67

| | low | high | mid | |
|---|---|---|---|---|
| | 0 | 13 | 6 | key > A[6] |
| | 7 | 13 | 10 | key < A[10] |
| | 7 | 9 | 8 | |

Terminating condition, since A[mid] = 67, return 9(successful).

## Algorithm

```
BinarySearch(A,l,r, key)
{
    if(l= = r)
    {
        if(key = = A[l])
            return l+1; //index starts from 0
        else
            return 0;
    }
    else
    {
        m = (l + r) /2 ; //integer division
        if(key = = A[m]
            return m+1;
        else if (key < A[m])
            return BinarySearch(l, m-1, key) ;
        else return BinarySearch(m+1, r, key) ;
    }
}
```

## Analysis:

From the above algorithm we can say that the running time of the algorithm is:
$T(n) = T(n/2) + Q(1)$
$= O(\log n)$ .

In the best case output is obtained at one run i.e. O(1) time if the key is at middle. In the worst case the output is at the end of the array so running time is O(logn) time. In the average case also running time is O(logn). For unsuccessful search best, worst and average time complexity is O(logn).

## Quick Sort

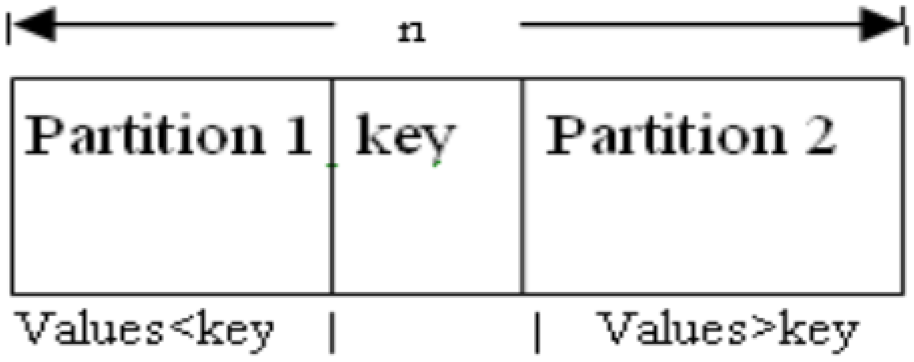
• **Divide**

Partition the array A[l…r] into 2 subarrays A[l..m] and A[m+1..r], such that each element of A[l..m] is smaller than or equal to each element in A[m+1..r]. Need to find index p to partition the array.

**Conquer**

Recursively sort A[p..q] and A[q+1..r] using Quicksort

**Combine**
Trivial: the arrays are sorted in place. No additional work is required to combine them.



## Algorithm:

```
QuickSort(A,l,r)
{
if(l<r)
{
p = Partition(A,l,r);
QuickSort(A,l,p-1);
QuickSort(A,p+1,r);
}
}

 Partition(A,l,r)
{
x =l; y =r ; p = A[l];
while(x<y)
{
do {
x++;
}while(A[x] <= p);
do {
y--;
} while(A[y] >=p);
if(x<y)
swap(A[x],A[y]);
}

A[l] = A[y]; A[y] = p;
return y; //return position of pivot
}
```

## Strassens's Matrix Multiplication

Strassen showed that 2x2 matrix multiplication can be accomplished in 7 multipication and 18 additions or subtractions.
The basic calculation is done for matrix of size 2 x 2 .

$$\begin{bmatrix} C11 & C12 \\ C21 & C22 \end{bmatrix} = \begin{bmatrix} A11 & A12 \\ A21 & A22 \end{bmatrix} \begin{bmatrix} B11 & B12 \\ B21 & B22 \end{bmatrix}$$

Where;

$P_1 = (A_{11}+ A_{22})(B_{11}+B_{22})$
$P_2 = (A_{21} + A_{22}) * B_{11}$
$P_3 = A_{11} * (B_{12} - B_{22})$
$P_4 = A_{22} * (B_{21} - B_{11})$
$P_5 = (A_{11} + A_{12}) * B_{22}$
$P_6 = (A_{21} - A_{11}) * (B_{11} + B_{12})$
$P_7 = (A_{12} - A_{22}) * (B_{21} + B_{22})$

$C_{11} = P_1 + P_4 - P_5 + P_7$
$C_{12} = P_3 + P_5$
$C_{21} = P_2 + P_4$
$C_{22} = P_1 + P_3 - P_2 + P_6$

**Greedy Algorithms**

Greedy method is the simple straightforward way of algorithm design. The general class of problems solved by greedy approach is optimization problems. In this approach the input elements are exposed to some constraints to get feasible solution and the feasible solution that meets some objective function best among all the solutions is called optimal solution. Greedy algorithms always makes optimal choice that is local to generate globally optimal solution however, it is not guaranteed that all greedy algorithms yield optimal solution. We generally cannot tell whether the given optimization problem is solved by using greedy method or not, but most of the problems that can be solved using greedy approach have two parts:

<u>Greedy choice property</u>

Globally optimal solution can be obtained by making locally optimal choice and the choice at present cannot reflect possible choices at future.

<u>Optimal substructure</u>

Optimal substructure is exhibited by a problem if an optimal solution to the problem contains optimal solutions to the sub-problems within it.

## Fractional Knapsack Problem

**Statement:** A thief has a bag or knapsack that can contain maximum weight W of his loot. There are n items and the weight of ith item is wi and it worth vi. Any amount of item can be put into the bag i.e. xi fraction of item can be collected, where 0<=xi<=1. Here the objective is to collect the items that maximize the total profit earned.

**Algorithm**:

Take as much of the item with the highest value per weight (vi/wi) as you can. If the item is finished then move on to next item that has highest (vi/wi), continue this until the knapsack is full. v[1… n] and w[1 … n] contain the values and weights respectively of the n objects sorted in non-increasing ordered of v[i]/w[i] . W is the capacity of the knapsack, x[1 … n] is the solution vector that includes fractional amount of items and n is the number of items.

```
GreedyFracKnapsack(W,n)
{
for(i=1; i<=n; i++)
x[i] = 0.0;
tempW = W;
for(i=1; i<=n; i++)
{
if(w[i] > tempW) then break;
x[i] = 1.0;
tempW -= w[i];
}
if(i<=n)
x[i] = tempW/w[i];
}
```

**Analysis:**

We can see that the above algorithm just contain a single loop i.e. no nested loops the running time for above algorithm is O(n). However our requirement is that v[1 … n] and w[1 … n] are sorted, so we can use sorting method to sort it in O(nlogn) time such that the complexity of the algorithm above including sorting becomes O(nlogn).

# Job sequencing with deadline

## Problem Statement

In job sequencing problem, the objective is to find a sequence of jobs, which is completed within their deadlines and gives maximum profit.

## Solution

Let us consider, a set of ***n*** given jobs which are associated with deadlines and profit is earned, if a job is completed by its deadline. These jobs need to be ordered in such a way that there is maximum profit.

It may happen that all of the given jobs may not be completed within their deadlines.

Assume, deadline of $\mathbf{i^{th}}$ job $J_i$ is $d_i$ and the profit received from this job is $p_i$. Hence, the optimal solution of this algorithm is a feasible solution with maximum profit.

Thus, $D(i)>0$ for $1\leqslant i\leqslant n$
Initially, these jobs are ordered according to profit, i.e. $p_1\geqslant p_2\geqslant p_3\geqslant\ldots\geqslant p_n$

Example
Let us consider a set of given jobs as shown in the following table. We have to find a sequence of jobs, which will be completed within their deadlines and will give maximum profit. Each job is associated with a deadline and profit.

| Job | $J_1$ | $J_2$ | $J_3$ | $J_4$ | $J_5$ |
|---|---|---|---|---|---|
| Deadline | 2 | 1 | 3 | 2 | 1 |
| Profit | 60 | 100 | 20 | 40 | 20 |

To solve this problem, the given jobs are sorted according to their profit in a descending order. Hence, after sorting, the jobs are ordered as shown in the following table.

| Job | $J_2$ | $J_1$ | $J_4$ | $J_3$ | $J_5$ |
|---|---|---|---|---|---|
| Deadline | 1 | 2 | 2 | 3 | 1 |
| Profit | 100 | 60 | 40 | 20 | 20 |

From this set of jobs, first we select $J_2$, as it can be completed within its deadline and contributes maximum profit.

- Next, $J_1$ is selected as it gives more profit compared to $J_4$.
- In the next clock, $J_4$ cannot be selected as its deadline is over, hence $J_3$ is selected as it executes within its deadline.
- The job $J_5$ is discarded as it cannot be executed within its deadline.

Thus, the solution is the sequence of jobs ($J_2$, $J_1$, $J_3$), which are being executed within their deadline and gives maximum profit.

Total profit of this sequence is **100 + 60 + 20 = 180**.

Algorithm

**Algorithm: Job-Sequencing-With-Deadline (D, J, n, k)**

```
D(0) := J(0) := 0
k := 1
J(1) := 1   // means first job is selected
for i = 2 … n do
  r := k
  while D(J(r)) > D(i) and D(J(r)) ≠ r do
    r := r – 1
  if D(J(r)) ≤ D(i) and D(i) > r then
    for l = k … r + 1 by -1 do
      J(l + 1) := J(l)
      J(r + 1) := i
      k := k + 1
```

## Analysis

In this algorithm, we are using two loops, one is within another. Hence, the complexity of this algorithm is $O(n^2)$.

## Tree vertex splitting

• Directed and weighted binary tree

• Consider a network of power line transmission

• The transmission of power from one node to the other results in some loss, such as drop in voltage

• Each edge is labeled with the loss that occurs (edge weight)

• Network may not be able to tolerate losses beyond a certain level

• You can place boosters in the nodes to account for the losses

- Let $T = (V, E, w)$ be a weighted directed tree
  - $V$ is the set of vertices
  - $E$ is the set of edges
  - $w$ is the weight function for the edges
  - $w_{ij}$ is the weight of the edge $\langle i, j \rangle \in E$
    We say that $w_{ij} = \infty$ if $\langle i, j \rangle \notin E$
  - A vertex with in-degree zero is called a *source vertex*
  - A vertex with out-degree zero is called a *sink vertex*
  - For any path $P \in T$, its *delay* $d(P)$ is defined to be the sum of the weights $(w_{ij})$ of that path, or

$$d(P) = \sum_{\langle i,j \rangle \in P} w_{ij}$$

Greedy solution for TVSP

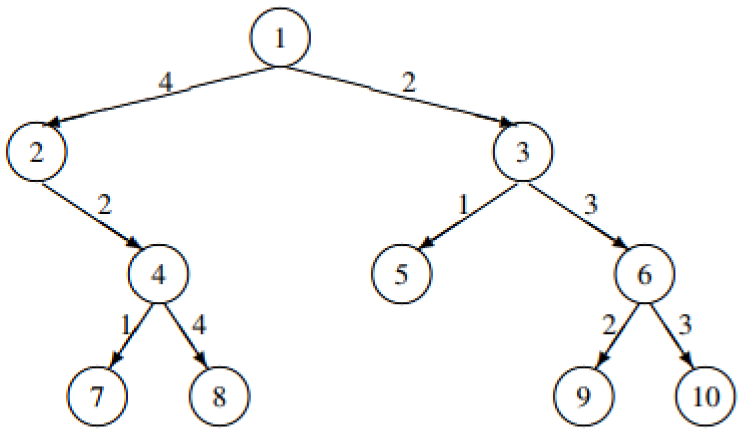
- We want to minimize the number of booster stations $(X)$
- For each node $u \in V$, compute the maximum delay $d(u)$ from $u$ to any other node in its subtree
- If $u$ has a parent $v$ such that $d(u) + w(v, u) > \delta$, split $u$ and set $d(u)$ to zero
- Computation proceeds from leaves to root
- Delay for each leaf node is zero
- The delay for each node $v$ is computed from the delay for the set of its children $C(v)$

$$d(v) = \max_{u \in C(v)} \{d(u) + w(v, u)\}$$

  If $d(v) > \delta$, split $v$
- The above algorithm computes the delay by visiting each node using post-order traversal

```
int tvs ( tree T, int delta )
{
if ( T == NULL ) return ( 0 ); // Leaf node
d_l = tvs ( T.left(), delta ); // Delay in left subtree
d_r = tvs ( T.right(), delta ); // Delay in right subtree
current_delay = max ( w_l + d_l, w_r + d_r );// Weight of left edge // Weight of right edge
if ( current_delay > delta )
{
if ( w_l + d_l > delta )
{
write ( T.left().info() );
d_l = 0;
}
if ( w_r + d_r > delta )
{
write ( T.right().info() );
d_r = 0;
}
}
current_delay = max ( w_l + d_l, w_r + d_r );
return ( current_delay );
}
```

Example:



**Solve the above tree with** $\delta = 5$

## Optimal Storage on Tapes – Solving using Greedy Approach

Optimal Storage on Tapes Problem: Given n programs P1, P2, …, Pn of length L1, L2, …, Ln respectively, store them on a tap of length L such that Mean Retrieval Time (MRT) is a minimum. The retrieval time of the jth program is a summation of the length of first j programs on tap. Let Tj be the time to retrieve program Pj. The retrieval time of Pj is computed as,

$$T_j = \sum_{k=1}^{j} L_k$$

Length of kth program

Mean retrieval time of n programs is the average time required to retrieve any program. It is required to store programs in an order such that their Mean Retrieval Time is minimum. MRT is computed as,

Average retrieval time over n programs

$$MRT = \frac{1}{n}\sum_{i=1}^{N} T_j = \frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{j} L_k$$

Time to retrieve jth program $P_j$

Length of kth program

Optimal storage on tape is minimization problem which,

$$\text{Minimize} \sum_{i=1}^{n} \sum_{k=1}^{i} L_k$$

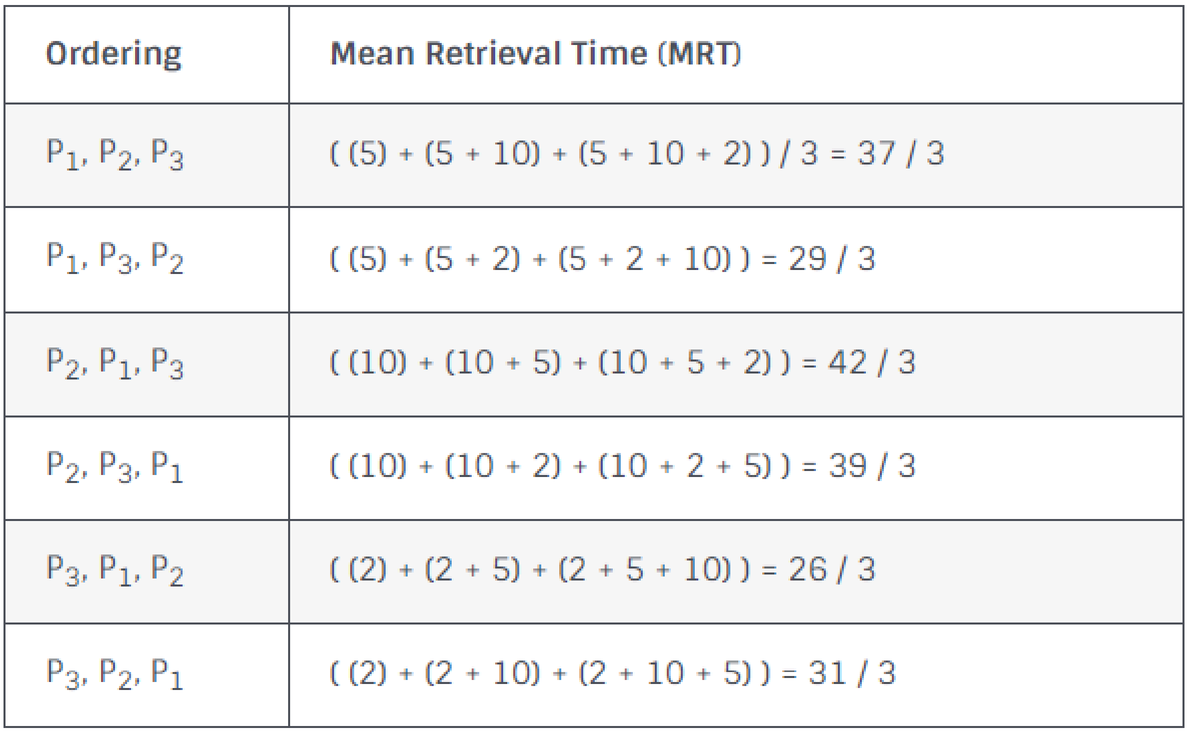
$$\text{Subjected to} \sum_{i=1}^{n} \underbrace{L_i}_{\text{Length of } i^{th} \text{ program}} \le \underbrace{L}_{\text{Length of tape}}$$

Storage on Single Tape

In this case, we have to find the permutation of the program order which minimizes the MRT after storing all programs on single tape only.

There are many permutations of programs. Each gives a different MRT. Consider three programs (P1, P2, P3) with a length of (L1, L2, L3) = (5, 10, 2).

Let’s find the MRT for different permutations. 6 permutations are possible for 3 items. The Mean Retrieval Time for each permutation is listed in the following table.



| Ordering | Mean Retrieval Time (MRT) |
|---|---|
| $P_1, P_2, P_3$ | ( (5) + (5 + 10) + (5 + 10 + 2) ) / 3 = 37 / 3 |
| $P_1, P_3, P_2$ | ( (5) + (5 + 2) + (5 + 2 + 10) ) = 29 / 3 |
| $P_2, P_1, P_3$ | ( (10) + (10 + 5) + (10 + 5 + 2) ) = 42 / 3 |
| $P_2, P_3, P_1$ | ( (10) + (10 + 2) + (10 + 2 + 5) ) = 39 / 3 |
| $P_3, P_1, P_2$ | ( (2) + (2 + 5) + (2 + 5 + 10) ) = 26 / 3 |
| $P_3, P_2, P_1$ | ( (2) + (2 + 10) + (2 + 10 + 5) ) = 31 / 3 |

It should be observed from the above table that the MRT is 26/3, which is achieved by storing the programs in ascending order of their length.

Thus, greedy algorithm stores the programs on tape in non-decreasing order of their length, which ensures the minimum MRT.

## Algorithm for **Optimal Storage on Tapes**

Let L be the array of program length in ascending order. The greedy algorithm finds the MRT as following

Algorithm MRT_SINGLE_TAPE(L)

// Description : Find storage order of n programs to such that mean retrieval time is minimum

// Input : L is array of program length sorted in ascending order

// Output : Minimum Mean Retrieval Time

```
Tj ← 0
for i ← 1 to n do
  for j ← 1 to i do
    Tj ← Tj + L[j]
  end
end
MRT ← Tj / n
```

## Complexity analysis of **Optimal Storage on Tapes**

$$T(n) = \underbrace{\sum_{i=1}^{n}}_{\text{Loop over n programs}} \underbrace{\sum_{j=1}^{i}}_{\text{Retrieval time for } j^{th} \text{ program}} \overbrace{\theta(1)}^{\text{Cost of addition}}$$

$$= \sum_{i=1}^{n} \sum_{j=1}^{i} 1$$

$$= \sum_{i=1}^{n} i = 1 + 2 + 3 + \ldots + n$$

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

$T(n) = O(n^2)$

This algorithm runs in $O(n^2)$ time.

# Optimal Merge Pattern

Optimal Merge Pattern Problem: “Merge n sorted sequences of different lengths into one sequence while minimizing reads”. Any two sequences can be merged at a time. At each step, the two shortest sequences are merged.

Consider three sorted lists L1, L2 and L3 of size 30, 20 and 10 respectively. Two way merge compares elements of two sorted lists and put them in new sorted list. If we first merge list L1 and L2, it does 30 + 20 = 50 comparisons and creates a new array L’ of size 50. L’ and L3 can be merged with 60 + 10 = 70 comparisons that forms a sorted list L of size 60. Thus total number of comparisons required to merge lists L1, L2 and L3 would be 50 + 70 = 120.

Alternatively, first merging L2 and L3 does 20 + 10 = 30 comparisons, which creates sorted list L’ of size 30. Now merge L1 and L’, which does 30 + 30 = 60 comparisons to form a final sorted list L of size 60. Total comparisons required to create L is thus 30 + 60 = 90.

In both the cases, final output is identical but first approach does 120 comparisons whereas second does only 90. Goal of optimal merge pattern is to find the merging sequence which results into minimum number of comparisons.

Let S = {s1, s2, …, sn} be the set of sequences to be merged. Greedy approach selects minimum length sequences si and sj from S. The new set S’ is defined as, S’ = (S – {si, sj}) ∪ {si + sj}. This procedure is repeated until only one sequence is left.

The working principle of optimal merge patterns is similar to Huffman coding tree construction. This can easily be done with min-heap data structure. In min heap, parent is always smaller than its child’s. Thus root of min-heap always contains minimum element. Using min heap we can retrieve minimum element in constant time. And insertion in min heap takes O(logn) time.

## Algorithm for Optimal Merge Pattern

```
Algorithm OPTIMAL_MERGE_PATTERNS(S)
// S is set of sequences

Create min heap H from S

while H.length > 1 do
  min1 ← minDel(H)
  // minDel function returns minimum element from H and delete it from H
  min2 ← minDel(H)
  NewNode.Data ← min1 + min2
  NewNoode.LeftChild ← min1
  NewNode.RightChild ← min2
  Insert(NewNode, H) // Insert node NewNode in heap H
End
```

## Complexity Analysis

In every iteration, two delete minimum and one insert operation is performed. Construction of heap takes O(logn) time. The total running time of this algorithm is O(nlogn).

T(n) = O(n – 1) * max(O(findmin), O(insert))

Case 1 : If list is not sorted :

O(findmin)  = O(n)

O(insert) = O(1)

So, T(n) = (n – 1) * n = O($n^2$)

Case 2.1 : List is represented as an array

O(findmin) = O(1)

O(insert) = O(n)

So, T(n) = (n – 1) * n = O($n^2$)

Case 2.2 : List is represented as min-heap

O(findmin)  =  O(1)

O(insert)  =   O(logn)

So, T(n) = (n – 1) * logn = O(nlogn)

## Example

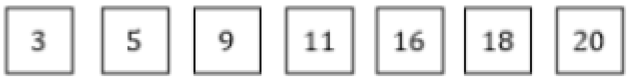
Example: Consider the sequence {3, 5, 9, 11, 16, 18, 20}. Find optimal merge patter for this data

Solution:

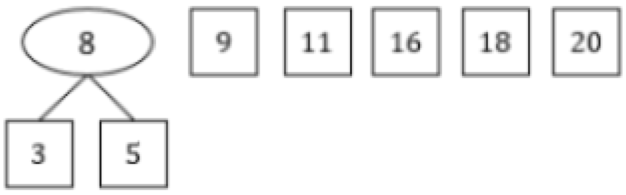
At each step, merge the two smallest sequences

Step 1:

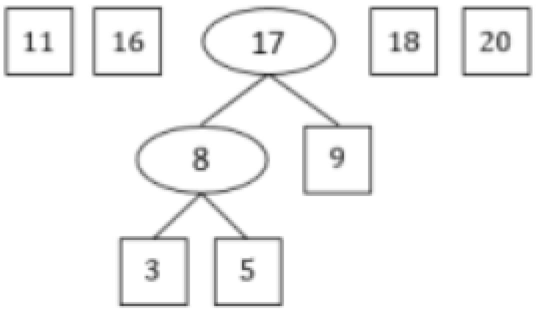
Given sequence is,



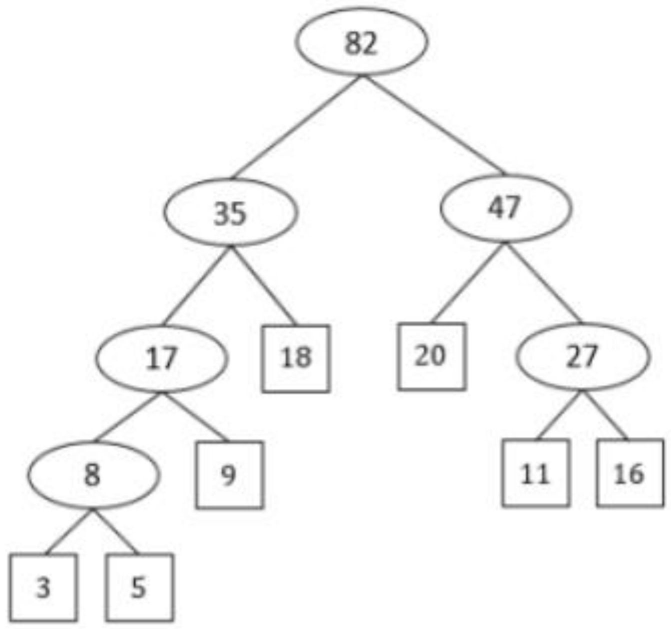
Step 2: Merge the two smallest sequences and sort in ascending order



Step 3: Merge the two smallest sequences and sort in ascending order



Follow same process …finally we will get



Total time = 8 + 17 + 27 + 35 + 47 + 82 = 216