Design and Analysis of Algorithms

Unit - II

Dr. R. Bhuvaneswari
Assistant Professor
Department of Computer Science
Periyar Govt. Arts College, Cuddalore.

Periyar Govt. Arts College
Cuddalore

Divide and Conquer

Syllabus

UNIT-II

Divide and Conquer: General Method – Binary Search – Finding Maximum and Minimum – Merge Sort – Greedy Algorithms: General Method – Container Loading – Knapsack Problem.

Text Book:

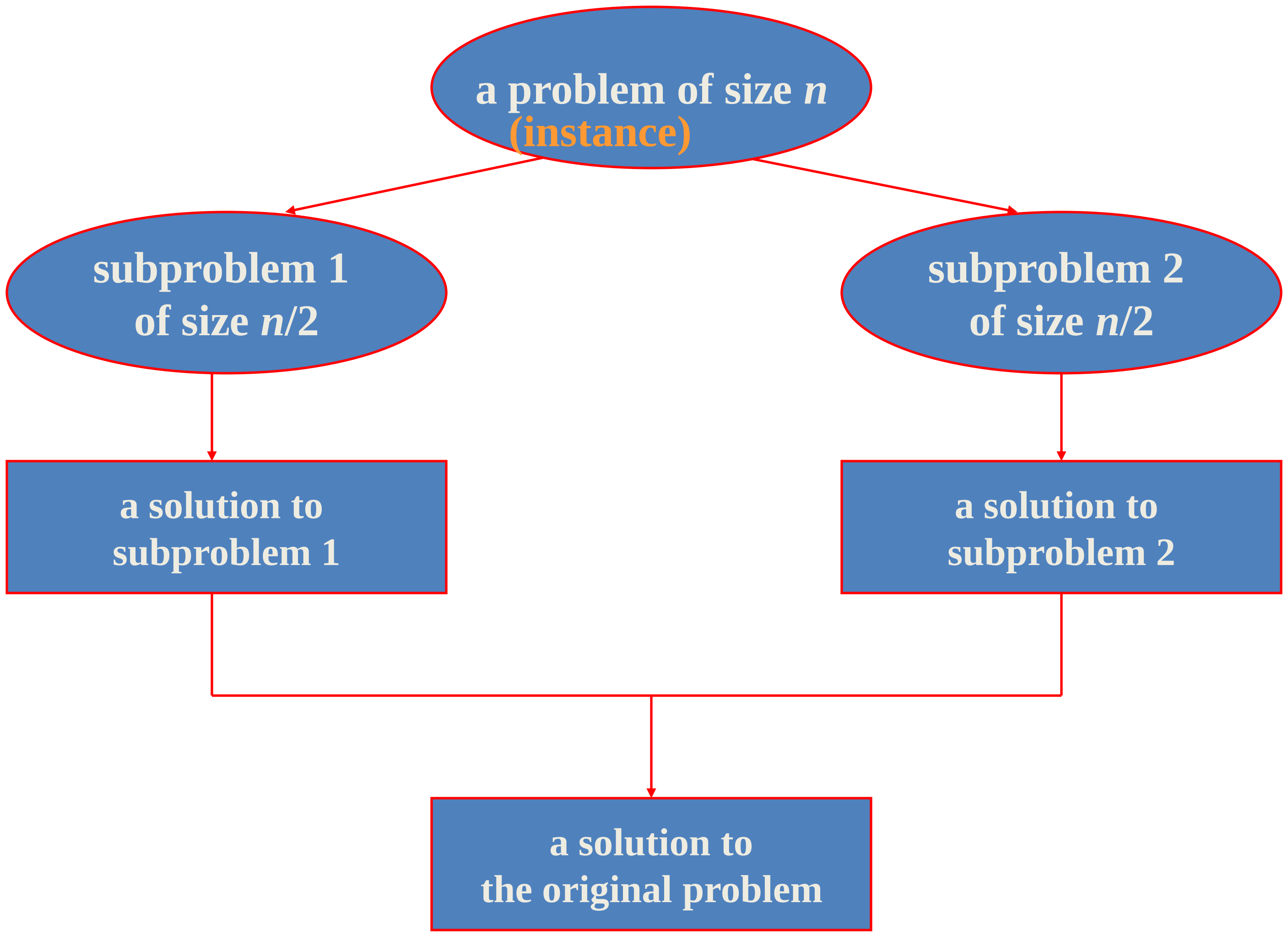
Ellis Horowitz, Sartaj Sahni and Sanguthevar Rajasekaran, Computer Algorithms C++, Second Edition, Universities Press, 2007. (For Units II to V)

Divide and Conquer

General Method:

- Given a function to compute on ‘n’ inputs, divide-and-conquer strategy suggests splitting the inputs into ‘k’ distinct subsets, $1<k\leq n$, yielding ‘k’ **subproblems**.
- These subproblems must be solved, and then a method must be found to combine sub solutions into a solution of the whole.
- If the subproblems are still relatively large, then the divide-and-conquer strategy can possibly be reapplied.
- For those cases the re-application of the divide-and-conquer principle is naturally expressed by a recursive algorithm.

Divide and Conquer



Divide and Conquer

Control Abstraction of Divide and Conquer

```
Algorithm DAndC(P)
{
if small(P) then
     return S(P);
else
{
    divide P into smaller instance P1, P2…….., Pk, k≥1;
    apply DAndC to each of these subproblems;
    return combine(DAndC(P1), DAndC(P2), ……., DAndC(Pk));
}
}
```

Divide and Conquer

Computing time of DAndC is:

$$T(n) = \begin{cases} g(n) & \text{n small} \\ T(n1) + T(n2) + \ldots . + T(nk) + f(n) & \text{otherwise} \end{cases}$$

where

T(n) is the time for DAndC on any input of size n

g(n) is the time to compute the answer directly for small inputs

f(n) is the time for dividing P and combining the solutions to subproblems

Binary Search

- Let a_i be a list of elements that are in non-decreasing order. $1 \le i \le n$.
- It is a problem of determining whether a given element x is present in the list.

1	2	3	4	5	6	7	8	9	10
10	20	30	40	50	60	70	80	90	100

mid = $\lfloor (low+high)/2 \rfloor$

x = 60

1. low = 1, high = 10
   mid = (1+10)/2 = 5, 60 > 50, low = 6
2. low = 6, high = 10
   mid = (6 + 10)/2 = 8, 60 < 80, high = 7
3. low = 6, high = 7
   mid = (6 + 7)/2 = 6

```
1.(x<a[mid] ) then
     high = mid-1
2.else if (x>a[mid]) then
     low = mid+1
3.else return mid;
```

Binary Search

1	2	3	4	5	6	7	8	9	10
10	20	30	40	50	60	70	80	90	100

```
Algorithm BinSearch(a,n,x)
//Given an array a[1:n] of elements in
//nondecreasing order, n ≥ 0
{
   low = 1; high = n;
   while (low ≤ high) do
   {
       mid := ⌊(low+high)/2⌋;
       if (x < a[mid]) then high = mid-1;
       else if (x > a[mid]) then low := mid+1;
              else return mid;
   }
   return 0;
}
```

mid = (low+high)/2 **x = 30**

1. low = 1, high = 10
   mid = (1+10)/2 = 5, 30 < 50, high = 4
2. low = 1, high = 4
   mid = (1 + 4)/2 = 2, 30 > 20, low = 3
3. low = 3, high = 4
   mid = (3 + 4)/2 = 3

Binary Search using recursion

1	2	3	4	5	6	7	8	9	10
10	20	30	40	50	60	70	80	90	100

```
Algorithm BinSrch(a,i,l,x)
{
if (l = i) then
{
   if (x = a[i]) then return i;
   else return 0;
}
else
{
   mid = ⌊(i+l)/2⌋;
   if (x = a[mid]) then return mid;
   else if (x < a[mid]) then return BinSrch(a,i,mid-1,x);
          else return BinSrch(a,mid+1,l,x);
}
}
```

mid = (i+l)/2 **x = 30**

1. i = 1, l = 10
   mid = (1+10)/2 = 5, 30 < 50, l = 4
2. i = 1, l = 4
   mid = (1 + 4)/2 = 2, 30 > 20, i = 3
3. i = 3, l = 4
   mid = (3 + 4)/2 = 3

Binary Search

Time Complexity

1. If the search element is the middle element of the array, **in this case, time complexity will be O(1), the best case.**
2. Otherwise, binary search algorithm breaks the array into half in each iteration.

The array is divided by 2 until the array has only one element:

$$\frac{n}{2^k} = 1$$

we can rewrite it as:

$$n = 2^k$$

by taking log both side, we get

$$\log_2{}^n = \log_2 2^k$$

$$\log_2{}^n = k\log_2{}^2$$

$$k = \log_2{}^n \text{ (since } \log_a{}^a = 1)$$

The time complexity of binary search is $\log_2{}^n$

Finding the maximum and minimum

- The problem to find the maximum and minimum items in a set of n elements.

```
Algorithm StraightMaxMin(a,n,max,min)
// set max to maximim and min to the
// minimum of a[1:n]
{
    max := min := a[1];
    for i := 2 to n do
    {
      if (a[i] > max) then max := a[i];
      if (a[i] < min) then min := a[i];
    }
}
```

- StraightMaxMin requires 2(n-1) element comparisons in the best, average and worst cases.

1	2	3	4	5
37	78	45	12	92

```
max = min = 37
i = 2
max = 78; min = 37
i = 3
max = 78; min = 37
i = 4
max = 78; min =12
i = 5
max = 92; min = 12
```

Finding the maximum and minimum

- An immediate improvement is possible by realizing that the comparison a[i] < min is necessary only when a[i] > max is false.

  Hence we can replace the contents of the for loop by

  ```
  if (a[i] > max) then max := a[i];
  else if (a[i] < min) then min := a[i];
  ```
- When the elements are in the increasing order the number of element comparisons is n-1.
- When the elements are in the decreasing order the number of element comparisons is 2(n-1).

Finding the maximum and minimum

Divide and Conquer Algorithm

- Let P = (n, a [i],……,a [j]) denote an arbitrary instance of the problem.
- Here 'n' is the no. of elements in the list (a[i],….,a[j]) and we are interested in finding the maximum and minimum of the list.
- If the list has more than 2 elements, P has to be divided into smaller instances.
- We divide 'P' into the 2 instances,
  - P1=([n/2], a[1],……..a[n/2]) and
  - P2= (n-[n/2], a[[n/2]+1],….., a[n])
- After having divided 'P' into 2 smaller sub problems, we can solve them by recursively invoking the same divide-and-conquer algorithm.
- max(P) is the maximum of max(P1) and max(P2)
- min(P) is the minimum of min(P1) and min(P2)

Finding the maximum and minimum

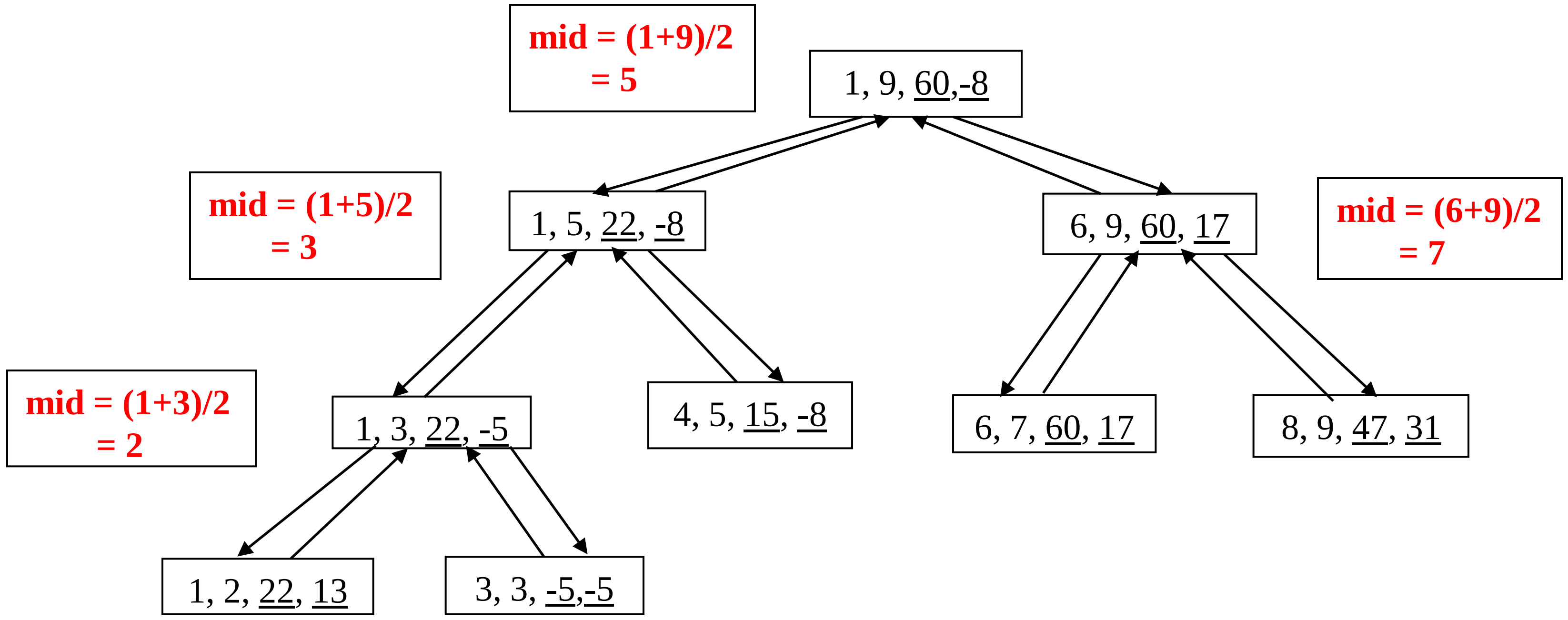
```
Algorithm MaxMin(i,j,max,min)
//a[1:n] is a global array.
{
if (i = j) then max = min = a[i];
else if (i = j-1) then
{
   if (a[i] < a[j]) then
  {
      max = a[j]; min = a[i];
   }
   else
  {
      max = a[i]; min = a[j];
   }
 }
 else
{
     mid = ⌊(i+j)/2⌋;
     MaxMin(i,mid,max,min);
     MaxMin(mid+1,j,max1,min1);
     if(max < max1) then max = max1;
     if (min > min1) then min = min1;
   }
}
```

Finding the maximum and minimum

Example: find max and min in the array:

22, 13, -5, -8, 15, 60, 17, 31, 47 (n = 9)

Index:	1	2	3	4	5	6	7	8	9
Array:	22	13	-5	-8	15	60	17	31	47



Finding the maximum and minimum

The number of element comparisons T(n) is represented as recurrence relation

$$T(n) = \begin{cases} T\left(\frac{n}{2}\right) + T\left(\frac{n}{n}\right) + 2 & n > 2 \\ 1 & n = 2 \\ 0 & n = 1 \end{cases}$$

When n is a power of two, $n = 2^k$ for some positive integer k, then

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

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

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

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

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

…….

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

Finding the maximum and minimum

Taking T(2) = 1

ie. $\frac{n}{2^k} = 2$

T(n) = $2^k + 2^k + 2^{k-1} + 2^{k-2} + \ldots\ldots. + 2$

$= 2^k + \sum_{j=1}^{k} 2^j$

$= 2^k + 2 * \frac{(2^k - 1)}{2-1}$

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

$= \frac{n}{2} + n - 2$

$= \frac{3n}{2} - 2$

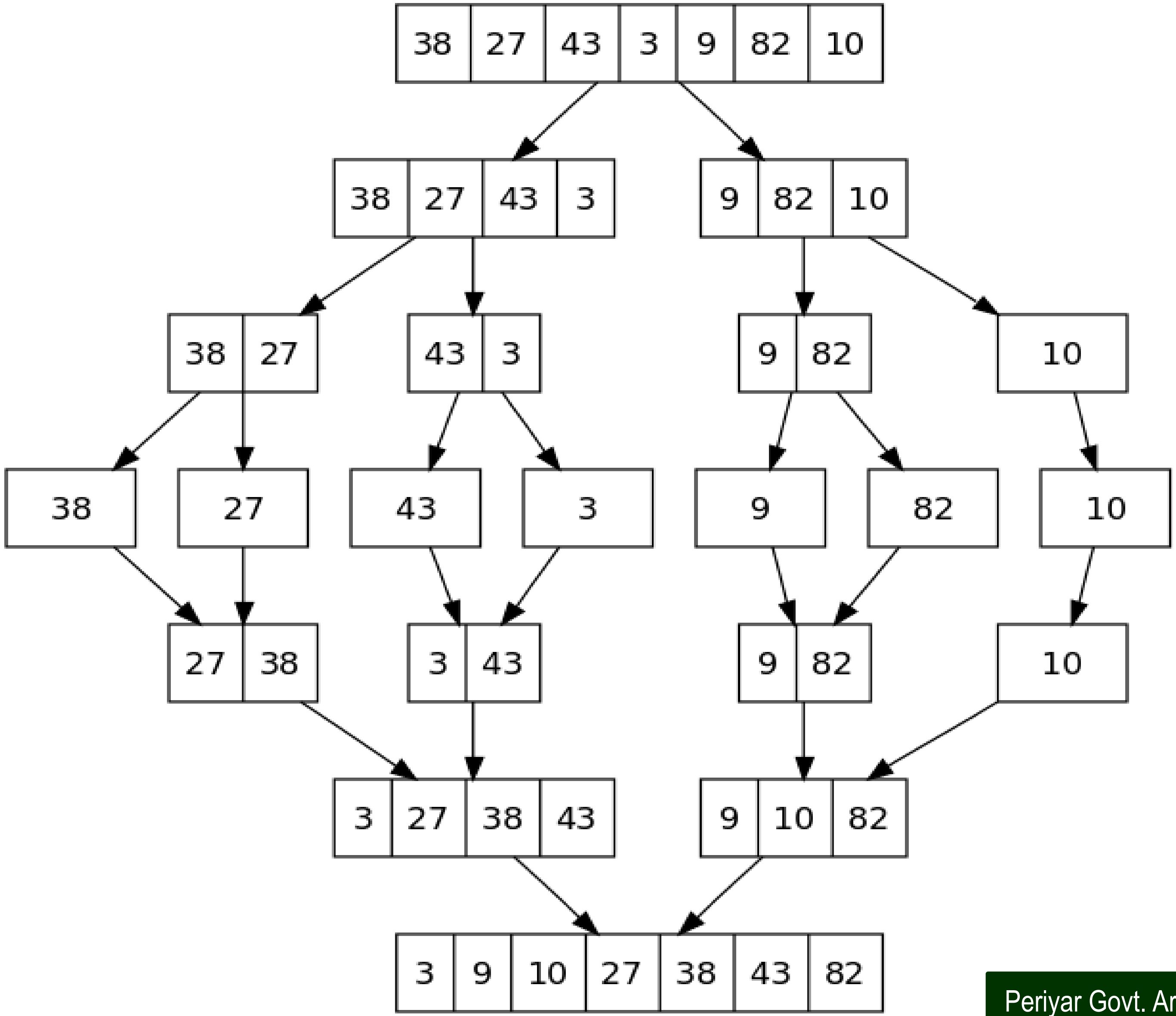
$$Since, \sum_{j=1}^{n} x^j = x * \frac{x^n - 1}{x - 1}$$

Therefore, 3n/2- 2 is the best, average and worst case number of comparisons where n is power of 2.

Merge Sort

- **Sort** a sequence of n elements into non-decreasing order.
- Merge sort is a sorting technique based on divide and conquer technique.
- Merge sort first divides the unsorted list into two equal halves.
- Sort each of the two sub lists recursively until we have list size of length 1, in which case the list itself is returned.
- Merge the two sorted sub lists back into one sorted list.

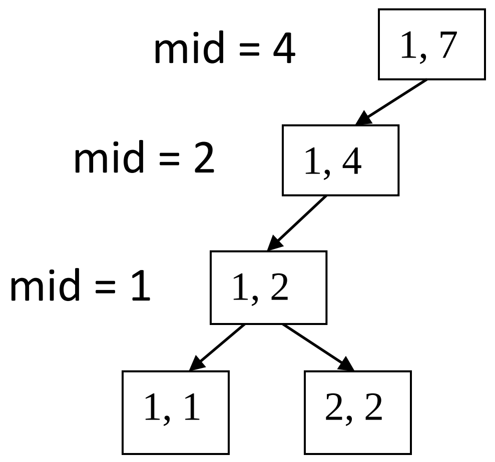
Merge Sort


38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

Merge Sort

```
Algorithm MergeSort(low,high)
{
  If (low < high) then
  {
    mid = $\lfloor$(low+high)/2$\rfloor$;
    MergeSort(low,mid);
    MergeSort(mid+1,high);
    Merge(low,mid,high);
  }
}
```

38	27	43	3	9	82	10



Merge Sort

```
Algorithm Merge(low,mid,high)
//b[] is an auxiliary global array.
{
  h=low; i=low; j=mid+1;
  while((h≤mid) and (j≤high)) do
  {
     if(a[h] ≤ a[j]) then
    {
       b[i] = a[h]; h = h+1;
    }
    else
    {
       b[i] = a[j]; j = j+1;
    }
    i = i+1;
  }
  if(h > mid) then
  {
    for k = j to high do
    {
       b[i] = a[k]; i = i+1;
    }
  }
  else
  {
    for k = h to mid do
    {
       b[i] = a[k]; i = i+1;
    }
  }
  for k = low to high do
       a[k] = b[k];
}
```

Merge Sort

Computing time for merge sort is described by the recurrence relation,

$$T(n) = \begin{cases} a & n = 1, a \text{ is a constant} \\ 2T\left(\frac{n}{2}\right) + cn & n > 1, c \text{ is a constant} \end{cases}$$

when $n = 2^k$

$$\begin{aligned} T(n) &= 2T(n/2) + cn \\ &= 2[2T(n/4) + cn/2] + cn \\ &= 4T(n/4) + cn + cn \\ &= 4T(n/4) + 2cn \\ &= 4[2T(n/8) + cn/4] + 2cn \\ &= 8T(n/8) + cn + 2cn \\ &= 8T(n/8) + 3cn \\ &\quad \ldots\ldots\ldots\ldots\ldots \\ &= 2^kT(n/2^k) + kcn \\ &= 2^kT(1) + kcn \\ &= an + cn\log n \end{aligned}$$

Since,
$T(n/2^k = 1)$
$n = 2^k$
$$\begin{aligned} \log_2{}^n &= \log_2 2^k \\ &= k\log_2{}^2 \\ &= k \end{aligned}$$

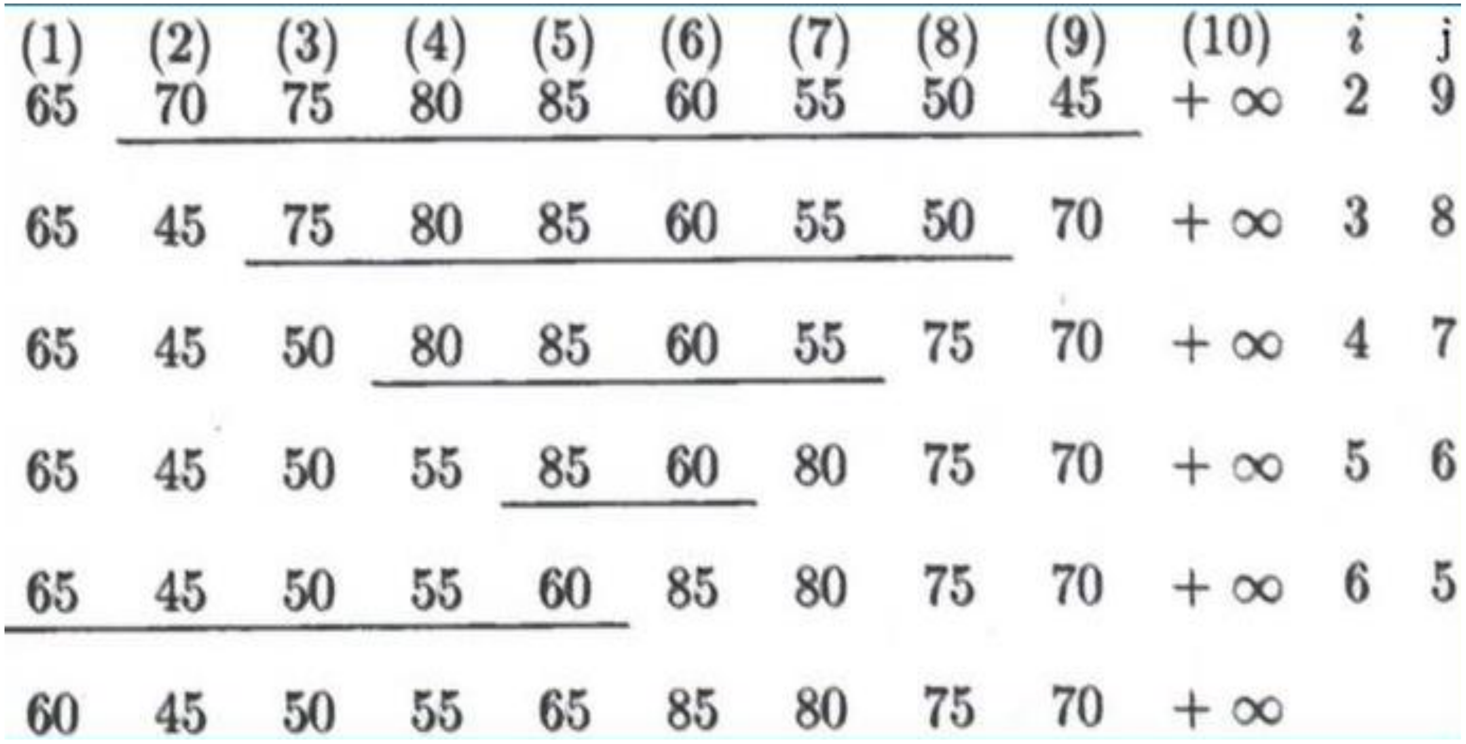
Quick Sort

- In merge sort, the array a[1:n] was divided at its midpoint into sub arrays which were independently sorted and later merged.
- In quick sort, the division into 2 sub arrays is made so that the sorted sub arrays do not need to be merged later.
- This is accomplished by rearranging the elements in a[1:n] such that $a[i] \le a[j]$ for all i between 1 and m and all j between (m+1) and n for some m, $1 \le m \le n$.
- Thus the elements in a[1:m] and a[m+1:n] can be independently sorted.
- No merging is needed.
- This rearranging is referred to as partitioning.

Quick Sort

- Quick sort picks an element as pivot element and partitions the given array around the picked pivot.
- There are many different versions of quick sort that pick pivot in different ways.
  - ➢ pick first element as pivot.
  - ➢ pick last element as pivot.
  - ➢ Pick a random element as pivot.
  - ➢ Pick median as pivot.
- The role of the pivot value is to assist with splitting the list.
- The actual position where the pivot value belongs in the final sorted list, commonly called the **split point**, will be used to divide the list for subsequent calls to the quick sort.

Quick Sort Example



(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	*i*	j
65	70	75	80	85	60	55	50	45	$+\infty$	2	9
65	45	75	80	85	60	55	50	70	$+\infty$	3	8
65	45	50	80	85	60	55	75	70	$+\infty$	4	7
65	45	50	55	85	60	80	75	70	$+\infty$	5	6
65	45	50	55	60	85	80	75	70	$+\infty$	6	5
60	45	50	55	65	85	80	75	70	$+\infty$		

Quick Sort

```
Algorithm Quicksort(p,q)
{
if (p<q) then
  {
     j:= Partititon (a,p,q+1);
     Quicksort(p,j-1);
     Quicksort(j+1,q);
  }
}

Algorithm Partition(a,m,p)
{
  v:=a[m]; i:=m; j:=p;
  repeat
 {
    repeat
          i:=i+1;
    until (a[i] ≥ v);
    repeat
          j := j-1;
    until (a[j] ≤ v);
    if (i < j) then Interchange(a, i, j);
 }until ( i ≥j);
 a[m] := a[j];
 a[j] := v;
 return j;
}

Algorithm Interchange(a, i, j)
{
   p := a[i];
   a[i] := a[j];
   a[j] := p;
}
```

Quick Sort

Computing time for Quick sort

$T(n) = 2T(n/2) + n$ for $n > 1$, $T(1) = 0$

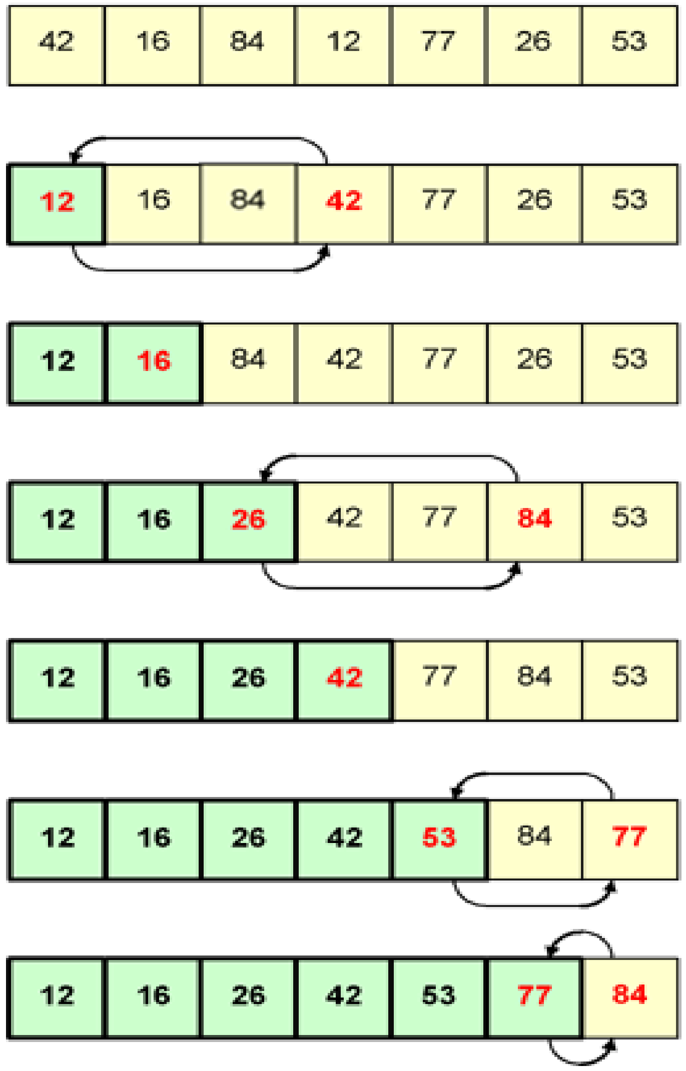
$$
\begin{aligned}
T(n) &= 2T(n/2) + n \\
&= 2[2T(n/4) + n/2] + n \\
&= 4T(n/4) + n + n \\
&= 4T(n/4) + 2n \\
&= 4[2T(n/8) + n/4] + 2n \\
&= 8T(n/8) + n + 2n \\
&= 8T(n/8) + 3n \\
&\quad \ldots\ldots\ldots\ldots\ldots \\
&= 2^kT(n/2^k) + kn \\
&= nT(1) + \mathbf{k}n \\
&= n\log n
\end{aligned}
$$

$$
\boxed{
\begin{aligned}
&\text{Since,} \\
&T(n/2^k = 1) \\
&n = 2^k \\
&\log_2{}^n = \log_2 2^k \\
&\qquad = k\log_2{}^2 \\
&\qquad = k
\end{aligned}
}
$$

Selection Sort

- Selection sort is the most simplest sorting algorithm.
- Following are the steps involved in selection sort(for sorting a given array in ascending order):
  - ➢ Starting from the first element, search the smallest element in the array, and replace it with the element in the first position.
  - ➢ Then move on to the second position, and look for smallest element present in the subarray, starting from index 2 till the last index.
  - ➢ Replace the element at the **second** position in the original array with the second smallest element.
  - ➢ This is repeated, until the array is completely sorted.

Selection Sort Example



The array, before the selection sort operation begins.

The smallest number (**12**) is swapped into the first element in the structure.

In the data that remains, **16** is the smallest; and it does not need to be moved.

26 is the next smallest number, and it is swapped into the third position.

42 is the next smallest number; it is already in the correct position.

53 is the smallest number in the data that remains; and it is swapped to the appropriate position.

Of the two remaining data items, **77** is the smaller; the items are swapped. *The selection sort is now complete.*

Selection Sort

```
Algorithm Selection(a, n)
{
   for i := 1 to n-1 do
  {
     min := a[i];
     loc := i;
     for j := i+1 to n do
     {
         if (min > a[j] ) then
        {
              min := a[j];
              loc :=j;
         }
     }
     temp := a[i]; a[i] := a[loc]; a[loc] := temp;
  }
}
```

Selection Sort

Number of comparisons in selection sort:

$(n-1) + (n-2) + (n-3) + \ldots\ldots. + 2 + 1$

$n(n-1)/2$ comparisons

Greedy Method

General Method:

- In the method, problems have n inputs and requires to obtain a subset that satisfies some constraints.
- Any subset that satisfies these constraints is called feasible solution.
- A feasible solution should either maximizes or minimizes a given objective function is called an optimal solution.
- The greedy technique in which selection of input leads to optimal solution is called subset paradigm.
- If the selection does not lead to optimal subset, then the decisions are made by considering the inputs in some order. This type of greedy method is called ordering paradigm.

Greedy Method

Control Abstraction of Greedy Method

```
Algorithm Greedy(a,n)
// a[1:n] contains n inputs
{
  solution := 0;
  for i :=1 to n do
 {
    x := select(a);
    if feasible(solution, x) then
             solution := Union(solution,x);
  }
  return solution;
}
```

Container Loading

- Large ship to be loaded with cargo.
- All containers are of the same size but may be of different weights.
- Container i has weight w_i.
- The capacity of the ship is C.
- Load the ship with maximum number of containers without exceeding the cargo weight capacity.
- Find values $x_i \in \{0, 1\}$ such that

$$\sum_{i=1}^{n} w_i x_i \leq C \qquad 1 \leq i \leq n$$

- And the optimum function $\sum_{i=1}^{n} x_i$ is maximized.
- Every set of x_i's that satisfy the constraints is a feasible solution.
- Every feasible solution that maximizes $\sum_{i=1}^{n} x_i$ is an optimal solution.

Container Loading

- Ship may be loaded in stages.
- Greedy criterion: From the remaining containers, select the one with least weight.

Example:

$n = 8$

$[w_1, w_2, w_3, w_4, w_5, w_6, w_7, w_8] = [100, 200, 50, 90, 150, 50, 20, 80]$

$C = 400$

$[x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8] = [1, 0, 1, 1, 0, 1, 1, 1]$

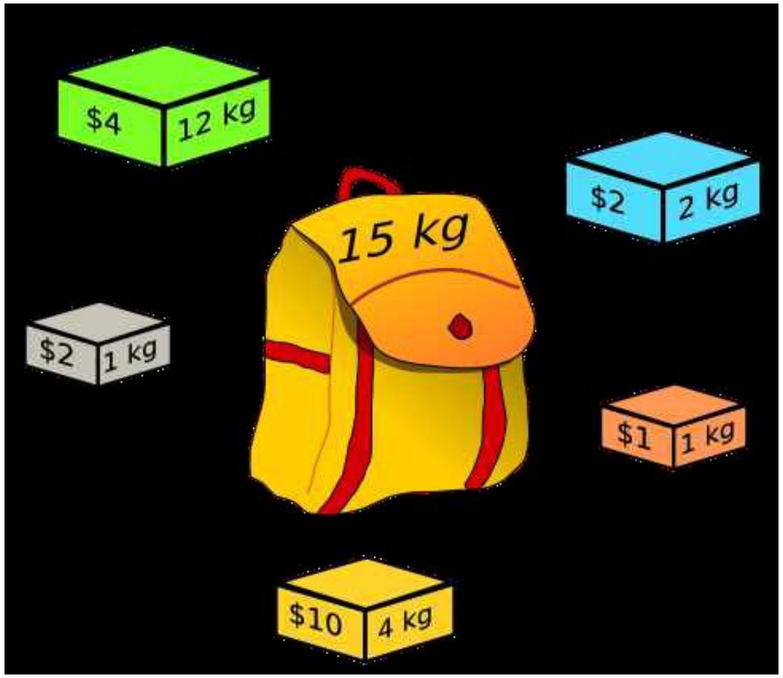
$$\sum x_i = 6$$

Container Loading

```
Algorithm ContainerLoading(c, capacity, numberofContainers, x)
// set x[i] = 1 if and only if container c[i], i ≥ 1 is loaded.
{
// sort into increasing order of weights.
  Sort(C, numberofContainers);
  n = numberofContainers;
  for i = 1 to n do
            x[i] = 0;
  i = 1;
  while ((i ≤ n) && (c[i].weight ≤ capacity))
  {
     x[c[i].id] = 1;
     capacity = capacity – c[i].weight;
     i++;
  }
}
```

Knapsack Problem

- Given a set of items, each with a weight and a profit, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total profit is as large as possible.
- Items are divisible; you can take any fraction of an item.
- And it is solved using greedy method.



Knapsack Problem

- Given n objects and a knapsack or bag.
- $w_i \rightarrow$ weight of object i.
- $m \rightarrow$ knapsack capacity.
- If a fraction x_i, $0 \leq x_i \leq 1$ of object i is placed into the knapsack, then a profit of $p_i x_i$ is earned.
- Objective is to fill the knapsack that maximizes the total profit earned.
- Problem can be stated as

$$\text{maximize} \sum_{1 \leq i \leq n} p_i x_i \quad - - - - - \text{①}$$

$$\text{subject to} \sum_{1 \leq i \leq n} w_i x_i \leq m \quad - - - - - \text{②}$$

$$0 \leq x_i \leq 1, 1 \leq i \leq n \quad - - - - - \text{③}$$

- A feasible solution is any set $(x_1 \ldots . x_n)$ satisfying equations ② and
- An optimal solution is a feasible solution for which equation ① is maximized.

Knapsack Problem

Example: n = 3, m = 20

Weight w_i	18	15	10
Profits p_i	25	24	15

	(x_1, x_2, x_3)	$\Sigma w_i x_i$	$\Sigma p_i x_i$
1.	(1/2, 1/3, 1/4)	16.5	24.25
2.	(1, 2/15, 0)	20	28.2
3.	(0, 2/3, 1)	20	31
4.	(0, 1, 1/2)	20	31.5
5.	(2/3, 8/15, 0)	20	29.5
6.	(5/6, 1/3, 0)	20	28.8

Among all the feasible solutions ④ yields the maximum profit

Knapsack Problem

The greedy algorithm:

Step 1: Sort p_i/w_i into nonincreasing order.

Step 2: Put the objects into the knapsack according to the sorted sequence as possible as we can.

e. g.

Weight w_i	15	10	18
Profits p_i	24	15	25

$n = 3, M = 20$

$(w_1, w_2, w_3) = (18, 15, 10)$

$(p_1, p_2, p_3) = (25, 24, 15)$

Sol: $p_1/w_1 = 25/18 = 1.39$

$p_2/w_2 = 24/15 = 1.6$

$p_3/w_3 = 15/10 = 1.5$

Optimal solution: $x_1 = 0, x_2 = 1, x_3 = 1/2$

Knapsack Problem

```
Algorithm GreedyKnapsack(m, n)
//n objects are ordered such that p[i]/w[i] ≥ p[i+1]/w[i+1].
{
   for i:= 1 to n do x[i] := 0.0;
   U := m;
   for i := 1 to n do
   {
       if (w[i] > U) then break;
       x[i] :=1.0;
       U := U-w[i];
   }
   if (i ≤ n) then
       x[i] = U/w[i];
}
```

Weight w_i	15	10	18
Profits p_i	24	15	25

x[i] = 0.0 **m = 20, n = 3**
x[2] = 0.0
x[3] = 0.0
U = 20
i = 1
x[1] = 1; U = 5
i = 2, 10 > 5
x[2] = 5/10 = 1/2
x[1] = 1, x[2] = 1/2, x[3] = 0