Unit 1: Advance Algorithm Analysis and Design Techniques

1.1 Advanced Algorithm Analysis Techniques:

[Not just consider one operation, but a sequence of operation on a given data structure.]

[This method calculates the average cost over the sequence of operations.] [No involvement of probabili

Amortized Analysis

In Amortized cost analysis, we analyze a sequence of operation and guarantee a worst case average time which is lower than the worst case time of a particular expensive operation. It is an average cost analysis. It is required because all operations within a set is not worst. Therefore, if an algorithm's average cost is better then it is worth while to implement

verage performance on a sequence of operations, even some operation is expensive]

uarantee average performance of each operation among the sequence in worst case.]

Example: Hash Tables, Disjoint Sets, Splay Trees.

Let us consider an example of a simple hash table insertions. How do we decide table size? There is a trade-off between space and time, if we make hash-table size big, search time becomes low, but space required becomes high.

The solution of this trade-off problem is to use Dynamic Table. The idea is to increase size of table whenever it becomes full.

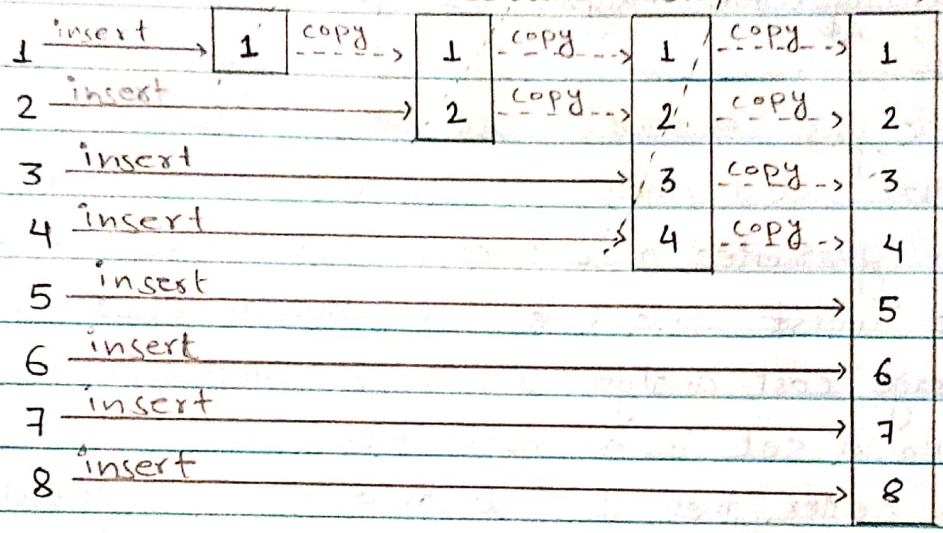
Following are the steps to follow when table becomes full.

1. Allocate memory for a larger table of size, typically twice the old table.
2. Copy the contents of old table to new table.
3. Free the old table.

If the table has space available, we simply insert new item in available space.

Insert Table Doubling

free ---> free (when table is double)



Note: C_i (copy + Insert, Excluding Table Doubling)

i	1	2	3	4	5	6	7	8	9	10
size	1	2	4	4	8	8	8	8	16	16
C_i	1	2	3	1	5	1	1	1	9	1
Actual cost		i	i		i				i	

C_i { 1 1 1 1 1 1 1 1 1 1 - Insert
1 2 4 8 - copy

There are three ways of Amortized cost Analysis

[v. Imp]

a. Aggregate Method
b. Accounting Method
c. Potential Method

ii) Accounting Method

- Idea
  - Assign differing charges to differen operations.
  - The amount of charge is called amortized cost.
  - Amortized cost is more or less than actual cost
  - When amortized cost > actual cost, the difference is saved in specific object as credits.
  - The credits can be used by later operation whose amortized cost < actual cost.

- As a comparision, in aggregate analysis, all operations have same amortized costs.

Consider the cost is added or reduced from the bank.

initially Bank >= 0

Assign $\hat{C}_i = 3$, considering every one operation cost as 3.

i	1	2	3	4	5	6	7	8	9	10
Actual cost (C_i)	1	2	3	1	5	1	1	1	9	1
$\hat{C}_i$	3	3	3	3	3	3	3	3	3	3
Bank	2	3	3	5	3	5	7	9	3	5

if every one operation cost is 3, then n operation is 3n.

$\therefore \Theta n = \frac{3n}{n} = 3$ Hence $O(n) = 1$

iii) Potential Method

The potential Method Works as follows. We will perform n operations, starting with an initial data structure D_0. For each $i = 1, 2, 3 \ldots n$, we let c_i be the actual cost of the i^{th} operation and D_i be the data structure. That results after applying the i^{th} operation to data structure D_{i-1}. A potential function (Φ) maps each data structure D_i to a real number $\Phi(D_i)$, which is the potential associated with data structure D_i.

The amortized cost $\hat{c}_i$ of the i^{th} operation with respect to potential function is defined by:

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

Let's assume

$$\Phi(D_i) = 2i - 2^{\lceil \log i \rceil}$$

$$\Phi(D_{i-1}) = 2(i-1) - 2^{\lceil \log (i-1) \rceil}$$

$$\hat{c}_i = \begin{cases} i & \text{if } i-1 \text{ is an exact power of 2} \\ 1 & \text{otherwise} \end{cases} + 2i - \left[2^{\lceil \log i \rceil} - 2(i-1) - 2^{\lceil \log (i-1) \rceil}\right]$$

Case 1: Taking i

$$= i + 2i - 2^{\lceil \log i \rceil} - \left[2(i-1) - 2^{\lceil \log (i-1) \rceil}\right]$$

$$= i + 2i - 2^{\lceil \log i \rceil} - 2i + 2 + 2^{\lceil \log (i-1) \rceil}$$

$$= i - 2(i-1) + 2 + i - 1$$

$$= i - 2i + 2 + 2 + i - 1$$

$$= 4 - 1$$

$$= 3$$

Case 2: Taking 1

$$= 1 - 2^{\lceil \log i \rceil} + 2 + 2^{\lceil \log (i-1) \rceil}$$

$$= 1 - (i-1) + 2 + (i-1)$$

$$= 3$$

Example

Maintenance Contract

In January, you buy a new car from a dealer who offers you the following maintenance contract: $50 each month other than March, June, September and December (this covers oil change and general inspection), $100 every March, June and September (this covers an oil change, a minor tune-up and a general inspection), and $200 every December (this covers an oil change, a major tune-up and a general inspection). We are to obtain an upper bound on the cost of this maintenance contract as a function of the number of months.

=> Worst-Case Method

We can bound the contract cost for the first n months by taking the product of n and the maximum cost charge in any month (i.e., $200). This would be analagous to the traditional way to estimate the complexity - take the product of number of operations and the worst-case complexity of an operation. Using this approach, we get $200n an upper bound on the contract cost. The upper bound is correct because the actual cost for the n months does not exceed $200n.

=> Aggregate Method

To use the aggregate method for amortized complexity we first determine an upper bound on the sum of the cost for the first n months. As tight a bound as is possible is desired. The sum of the actual monthly costs of the contract for the first n months is.

$$200\lfloor n/12 \rfloor + 100 \times (\lfloor n/3 \rfloor - \lfloor n/12 \rfloor) + 50 \times (n - \lfloor n/3 \rfloor)$$

$$= 200(n/12) + 100 \times (3n/12) + 50 \times (2n/3)$$

$$= \frac{50n}{3} + 25n + 100n/3$$

$$= \frac{50n + 75n + 100n}{3}$$

$$= \frac{225n}{3}$$

$= 75n$ Hence one operation cost 75n and n operation will cost $75n/n = 75$ ie. constant.

Cost

$$= 50n + 50n/3 + 100n/12$$

$$= \frac{600n + 200n + 100n}{12}$$

$$= \frac{900n}{12}$$

$= 75n$

Amortized cost for n operation is.

$75n/n = 75$ ie constant.

⇒Accounting Method.

To use the accounting Method, we must first assign an amortized cost for each month.

month	1	2	3	4	5	6	7	8	9	10	11	12
actual cost	50	50	100	50	50	100	50	50	100	50	50	200
amortized cost	75	75	75	75	75	75	75	75	75	75	75	75
P(i)	25	50	25	50	75	50	75	100	75	100	125	0

If every operation cost is 75 then n of operation cost 75n

$\therefore O(n) = \frac{75n}{n} = 75$ i.e. constant $O(1)$

⇒ Potential Method

We first define the potential function for the analysis. The only guideline you have in defining this function is that the potential function represents the cumulative difference between the amortized and actual costs.

$$P(n) = \begin{cases} 0 & n \bmod 12 = 0 \\ 25 & n \bmod 12 = 1 \text{ or } 3 \\ 50 & n \bmod 12 = 2, \text{or } 4, \text{or } 6 \\ 75 & n \bmod 12 = 5 \text{or } 7 \text{or } 9 \\ 100 & n \bmod 12 = 8, \text{or } 10 \\ 125 & n \bmod 12 = 11 \end{cases}$$

Here amortized cost = actual(i) + P(i) - P(i-1)

Therefore

amortized (1) = actual (1) + P(1) - P(0) = 50 + 25 - 0 = 75

amortized (2) = actual (2) + P(2) - P(1) = 50 + 50 - 25 = 75

amortized (3) = actual(3) + P(3) - P(2) = 100 + 25 - 50 = 75

and so on. Therefore, the amortize cost for each month is $75. i.e. constant O(1).

Probabilistic Analysis

Probabilistic analysis is the use of probability in the analysis of problems. Most commonly, we use probalistic analysis to analyze the running time of an algorithm. Sometimes we use it to analyze other quantities, such as the hiring cost in procedure HIRE-ASSISTANT. In order to perform a probablistic analysis, we must use knowledge of ~~our~~ or make assumption about the distribution of the inputs. Then we analyze our algorithm, computing an average-case running time, where we take the average over the distribution of the possible inputs. When reporting such a running time, we will refer to it as the average-case running time.

Randomized Algorithms

- uses random technique in the logic of the algorithm in some part of the process.
- decision made in algorithms depends on output of the randomizer (i.e. random number generator.)
- the execution time of the algorithm may vary from run to run for same input.

Two methods.

1) Las Vegas Algorithm
2) Monte Carlo Algorithm

1) Las Vegas Algorithm:

- produces always same output for same inputs.
- execution time depends on the output of the randomizer.
- finite running time (it may terminate fast or takes longer time which is characterized by random number)
- If we are lucky, the algorithm might terminate fast, and if not it might take longer period of time. In general, the execution

time of a Las Vegas algorithm is characterized as a random variable.

2) Monte Carlo Algorithm.

- Output may differ from run to run for same input.
- It may generate wrong answer sometimes that depends on the output of the randomizer.
- Run time is fixed.

Problem Statement:

Give an array with n elements A[1....n]. Half of the array contains 0s, the other half contains 1s.

Goal:

Finding an index that contains a 1

Las Vegas

```
repeat:
    k = RandInt(n)
    if A[k] = 1, return k
```

MonteCarlo

```
repeat 300 times:
    k = RandInt(n)
    if A[k] = 1, return k
return "Failed"
```

Las Vegas: Output is always correct. Ex: Randomized Quicksort.

Monte Carlo: Output may be incorrect, Randomized Median

Another Example

General Search for Repeating numbers:

```
AlgoSearchRepeat (A)
{
for i = 0 to n-1
    for j = i+1 to n
        if A[i] == A[j]
        return true
}
```

Las Vegas Search:

```
AlgoLasVegasSearchRepeat (A)
{
  while (true) do
      i = random () mod n+1
      j = random () mod n+1
      if (i != j and A[i] == A[j])
          return True
}
```

General Search:

```
Algo Search (A, a)
{
  for i=0 to n-1
      if A[i] == a
       return True
}
```

Monte Carlo Search:

```
Algo Monte Carlo Search (A, a, x)
{
  for i=0 to n-1
      j= random() mod n+1
      if (A[j] == a)
      return True
}
```

Primality Testing

Any integer greator than one is said to be a prime if its only divisor are 1 and the integer itself. By convention, we take 1 to be a non-prime. Then 2, 3, 5, 7, 11 and 13 are the first six primes. Given an integer n, the problem of deciding whether n is a prime is known as primality testing.

eg:

$a^n \% n = 1 \; ; a < n$

where, n is the number to be tested (input)

if n=3

Then

a=1, 2

a = 1

$1^3 \% 3 = 1$

a = 2

$2^3 \% 3 = 8 \% 3 = 2$

```
LasVegas( )
{
  while(true) do
  {
   i = Random( ) mod 2;
   if (i>=1) then return;
  }
}
```

1.2 Advance Algorithm Design Techniques

1. Greedy Method
2. Dynamic Programming
3. Back Tracking

These methods are used to solve optimization problem.

1. Greedy Method

- It is a straight forward design technique that can be applied to wide variety of problems in which problems have 'n' input that requires feasible solution with optimum (max/min) objective function known as optimal solution.

- It works on stages considering one input at a time

- At each stage a decision is made regarding whether particular input is an optimal solution.

```
Greedy (A[], n)
{
  for (i=1 to n)
  {
    x= select (A)
    if (feasible x)
```

```
{
  solution = solution + x
}
return solution
}
```

Job Scheduling with Deadline

- Given: set of n jobs
- di and Pi are deadline and profit associated with job Ji, where i is an integer.
- for any job, profit Pi is earned iff Ji is finished by its deadline.
- Ji takes one unit time
- Single machine is available for processing.
- Feasible solution = [subset J of jobs such that each job in subset can be completed by its deadline]
- Value of feasible solution = $\sum_{i=J}$ Pi [ie. sum of profits of the jobs in J]

Algorithm

1. Arrange Ji in descending order with respect to Pi
2. Allocate Time slot with respect to maximum deadline
3. K= min(max(deadline), di)
4. Allocate the Time slot for Ji w.r.t. value of k.
5. If the time slot is already occupied then search for the positions before it.
6. If all the timeslot is occupied publish the set and sequence and max profit.

For example:

Jobs :	J_1	J_2	J_3
deadline :	1	1	1
Profit :	10	15	30

Arrange the Jobs in descending order according to Pi

Jobs :	J_3	J_2	J_1
deadline:	1	1	1
Profit :	30	15	10
i :	1	2	3

for i = 1

k = min(max(deadline), di)

= min(1, 1)

= 1

Time slot = 1

J_3

k = 1

set = {J_3}

sequence = {J_3}

maxProfit = 30

2# Ji: J_1 J_2 J_3 J_4

Pi: 100 10 15 27

di: 2 1 2 1

Arrange the jobs in descending order according to Pi

Ji: J_1 J_4 J_3 J_2

Pi: 100 27 15 10

di: 2 1 2 1

qi: 1 2 3 4

Here

Time slot=2

J_4	J_1
K=1	K=2

for i=1

K= min(max(deadline), di)

= min (2, 2)

= 2

set = {J_1, J_4}

seq^n = {J_4, J_1}

max profit = 127

for i=2

K= min(max(deadline), di)

= min(2, 1)

= 1

3#

Ji	J1	J2	J3	J4
Pi	30	15	75	80
di	1	2	3	2

Arrange the jobs in descending order according to Pi

Ji	J4	J3	J1	J2
Pi	80	75	30	15
di	2	3	1	2
i	1	2	3	4

Time slot = 3

for i = 1

k = min(max(deadline), di)

= min(3, 2)

= 2

J4	J1	J3
k=1	k=2	k=3

Set = {J1, J3, J4}

Seqn = {J1, J4, J3}

maxprofit = 185

for i = 2

k = min(max(deadline), di)

= min(3, 3)

= 3

for i = 3

k = min(max(deadline), di)

= min(3, 1)

= 1

4. | Ji | J1 | J2 | J3 | J4 |
|---|---|---|---|---|
| Pi | 30 | 15 | 75 | 80 |
| di | 1 | 2 | 2 | 2 |

Arrange the jobs in descending order according to Pi

Ji	J4	J3	J2	J1
Pi	80	75	30	15
di	2	2	2	1
i	1	2	3	4

Time slot = 2

J3	J4
k=1	k=2

for i = 1

k = min(max(deadline), di)
= min(2, 2)
= 2

for i = 2

k = min(max(deadline), di)
= min(2, 2)
= 2

set = {J3, J4}
seqn = {J3, J4}
max profit = 155

Analysis

Case I:

Each Time slot calculated is vaccant so just insert is required which is equal to the number of iterations.

$\therefore O(n) = n$

Case II:

Time slot calculated is not vacant. So required another iteration to find the vacant slot.

$$O(n) = n(n-1)$$
$$= n^2$$

```
Greedy Job (d, J, n)
{ J = {1}
  for i: 2 to n do
  { if all deadlines is J ∪ {i} are feasible then,
    J = J ∪ {i}
  }
  return J:
}
```

Tree Vertex Splitting

- Consider directed binary tree each edge is levelled with weight
- It is used to model a distributed network in which electric signals / oil are transmitted.
- Nodes represents receiving stations and edges represents transmission lines.
- Some loss may occur in the process of transmission eg. drop of voltage in case of electric signals.
- Each edge is lebelled with respective loss during traversing that edge.
- The network may not be able to tolerate loss beyond certain level.
- In places where loss exceeds the tolerance level boosters have to be installed

TVSP can be defined as:

Let $T = (V, E, W)$ be a weighted directed tree where,

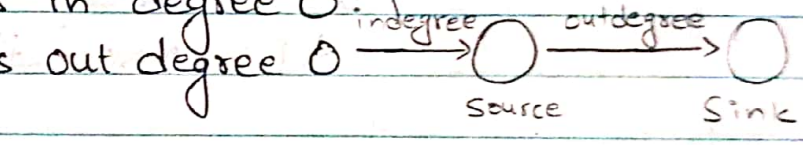
$V \rightarrow$ Set of vertices

$E \rightarrow$ Set of edges

$W(i,j) \rightarrow$ Weight of edge $\langle i, j \rangle \in E$

And,

source vertex has in degree 0. and Sink vertex has out degree 0



We define delay of path P denoted as

$d(P)$ = Sum of weight of path

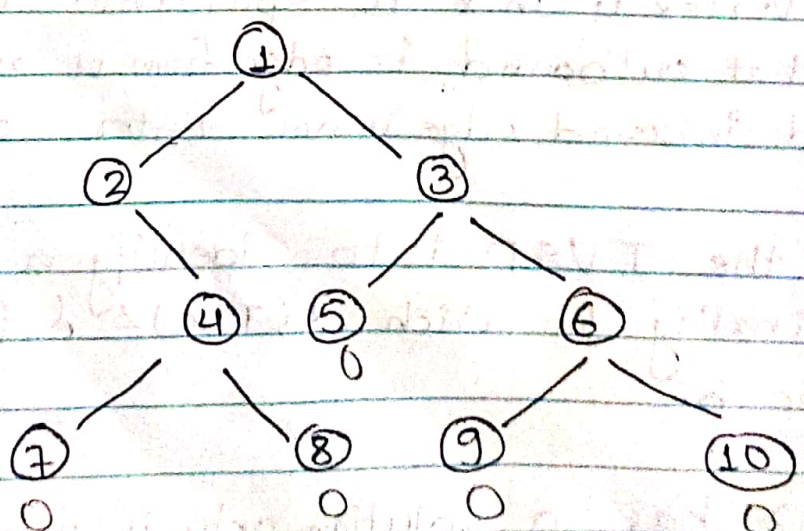
Delay of Tree, $d(T) = \max(d(P))$

- Let Tree splitting is a forest that results when each vertex u in X is split into two nodes u^o and u^i such that outbound is edge from u now leaves from u^o and inbound edge u now enters at u^i
- Now, the TVSP is to identify a set $X \subseteq V$ of cardinality for which $d(T/X) \leq \delta$ for some loss-tolerance δ.
- TVSP has a solution only if max (edge weight $\leq \delta$)

Algorithm of Tree Vertex Splitting

```
void TVS (T, delta)
  {
   if (T != NULL)
     {
      d(T) = 0;
      for (each child V of T)
       {
        TVS (V, delta)
          d(T) = max {d(T), d(V) + w (T, V)};
       }
      if ((T is not root) && (d(T) + w (Parent (T), T) > d
       {
        cout << T << end;
          d(T) = 0;
       }
     }
  }
```

Example:



Delay of leaf Node = 0

All leaf node delay is 0

ie. $d[7], d[8], d[5], d[9], d[10] = 0$

Assume, $\delta = 5$ [Tolerance Limit = 5]

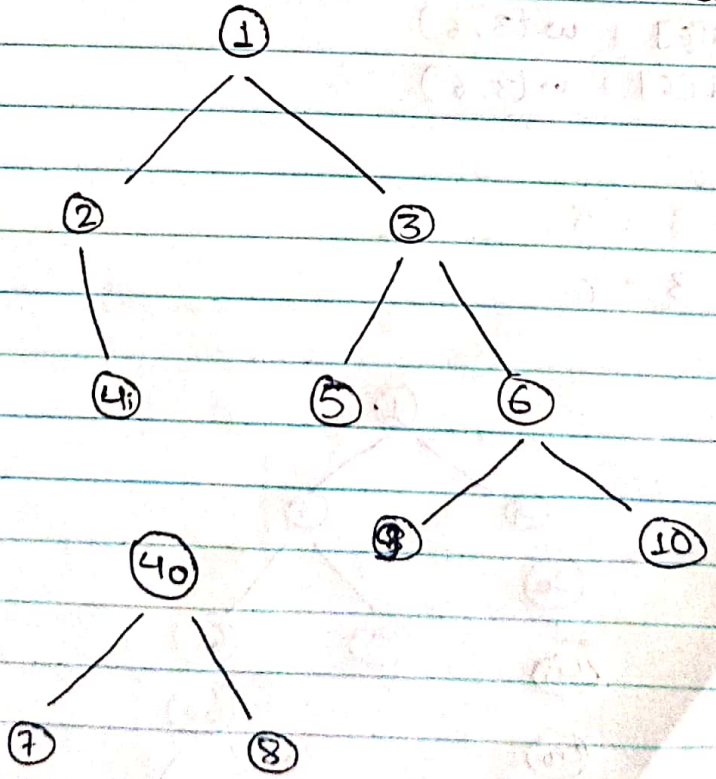
$$d[4] = \max \begin{cases} d[4] + w(2,4) \\ d[4] + w(2,4) \end{cases} \begin{bmatrix} + w(7,4) \\ + w(8,4) \end{bmatrix}$$

$$= \max \begin{cases} 2 + 1 = 3 \\ 2 + 4 = 6 \end{cases}$$

$= 6 >$ Tolerance limit so split.

Where i = inbound

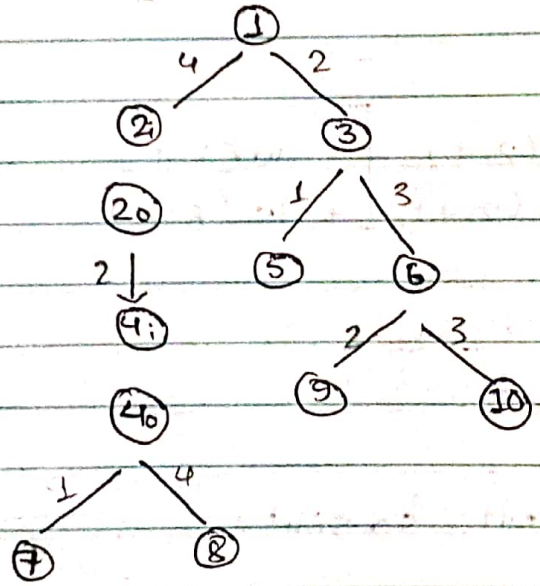
O = Outbound



$d[2] = \max\{ d[2] + w(1,2)$

$= \max\{ 4+2 = 6$

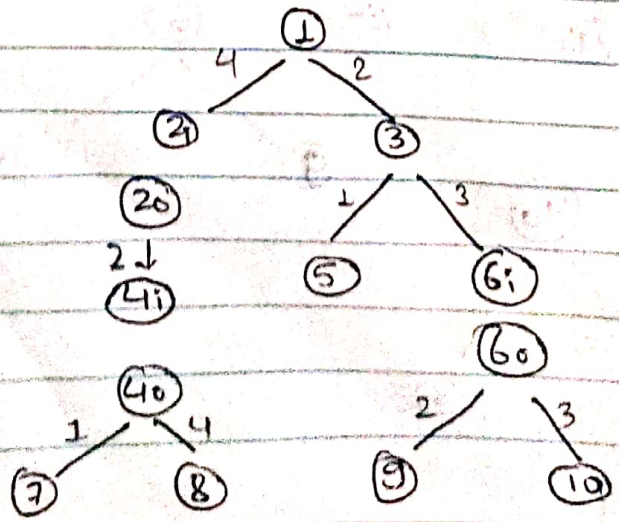
$= 6$



$d[6] = \max\begin{cases} d[6] + w(3,6) \\ d[6] + w(3,6) \end{cases}$

$= \max\begin{cases} 2 + 3 = 5 \\ 3 + 3 = 6 \end{cases}$

$= 6$



$$d[3] = \max\begin{cases} d[3] + w(1,3) \\ d[3] + w(1,3) \end{cases}$$

$$= \max\begin{cases} 1 + 2 = 4 \\ 3 + 2 = 5 \end{cases}$$

$\therefore 5 = \delta$ [i.e. 5 not less than δ (Tolerance limit)]

so no split

Analysis

- Visit each vertex n.
- In each vertex there is costant time required for calculating and comparing it with δ

$\therefore O(n) = n$

Dynamic Programming

Dynamic Programming is an algorithm design method that can be used when the solution to a problem can be viewed as the result of a sequence of decisions.

Greedy Vs Dynamic

Feature	Greedymethod	Dynamic programming
Feasibility	In a greedy algorithm, we make whatever choice seems best at the moment in the hope that it will lead to global optimal solution.	In Dynamic programming we make decision at each step considering current problem and solution to previously solved sub problem to calculate optimal solution.
Optimality	In Greedy method, sometimes there is no such guarantee of getting Optimal Solution	It is guaranteed that Dynamic programming will generate an optimal solution as it generally consider all possible cases and choose the best.
Recursion	A greedy method follows the problem solving heuristic of making the locally optimal choice at each stage	A Dynamic programming is an algorithmic technique which is usually based on a recurrent formula that uses some previously calculated states

Memoization (optimizing technique to speed up computer)	It is more efficient in terms of memory as it never look back or revise previous choices	It requires dp table for memoization and it increases it's memory complexity.
Time Complexity	Greedy methods are generally faster.	Dynamic Programming is generally slower
Fashion	The greedy method computes its solution by making its choices in a serial forward fashion, never looking back or revising previous choices.	Dynamic programming computes its solution bottom up or top down by synthesizing them from smaller optimal sub solution
Example	Fractional knapsack	0/1 Knapsack problem

String Editing

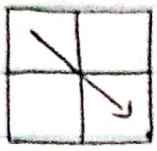
We are give two strings $X = x_1, x_2, \ldots, x_n$ and $Y = y_1, y_2, \ldots, y_m$, where x_i, $1 \le i \le n$ and y_j, $1 \le j \le m$ are members of finite set of symbols known as the alphabets. We want to transform X into Y using a sequence of edit operations on X. The permissible edit operations are insert, delete, and Replace(change), and there is a cost associated with performing each. The problem of string editing is to identify a minimum-cost sequence of edit operations that will transform X into Y.

Let $D(x_i)$ be the cost of deleting the symbol x_i from X, $I(y_i)$ be the cost of inserting the symbol y_i into X and $C(x_i, y_i)$ $(or R(x_i, y_i))$ be the cost of changing or replacing the symbol x_i of X into y_i.

Terminologies

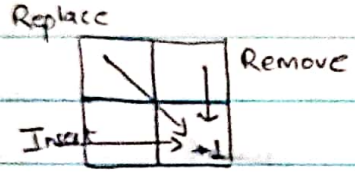
Cost	Operation
0	Copy C_i
1	Remove (Delete) D_i
1	Insert I_i
2	Replace (change) R_i

When character is equal



When character is unequal

Either Remove or Insert or Replace



[check the minimum and add 1 to it]

Example

Given two string

X = a a b a b

Y = b a b b

Convert X to Y

Let

String 1 (x) : a a b a b (source)

⇓ edit

String 2 (Y) : b a b b (distance)

Technique 1 :

- Remove all characters within string 1 (x) = 5
- Insert the required characters in String 2 (Y) = 4

9 - cost

Technique 2:

Insert 'b' at last and Remove 'a' 'a'

D D
a a b a b b
↑
I

Here,

Delete 2 character = 2 cost (a, a)

& Insert 1 character = 1 cost (b)

Total cost = 3 cost

Technique 3:

Replace 'a' with 'b' and remove 'a'.

R (a → b)
a a b a b
D

=> b a b b

Here

Replace 1 character = 2 cost (a → b)

Delete 1 character = 1 cost (a)

Total cost = 3 cost.

Minimum Edit Distance (string editing) of given problem is.

i↓ j→	Null	b	a	b	b
Null	0	1	2	3	4
a	1	~~2~~	1	2	3
a	2	3	2	3	4
b	3	2	3	2	3
a	4	3	2	3	3
b	5	4	3	2	3

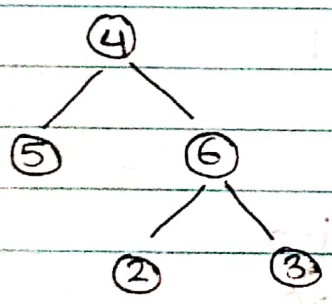
$$\text{cost}(i,j) = \begin{cases} 0, & i=j=0 \\ \text{cost}(i-1,0) + D(x_i); & i>0,\ j=0 \\ \text{cost}(0,j-1) + I(y_i), & i=0,\ j>0 \end{cases}$$

$$= \begin{cases} \text{cost}(i-1,j) + D(x_i) \\ \text{cost}(i,j-1) + I(y_j) \\ \text{cost}(i-1,j-1) + c(x_i,y_j), & i>0,\ j>0 \end{cases}$$

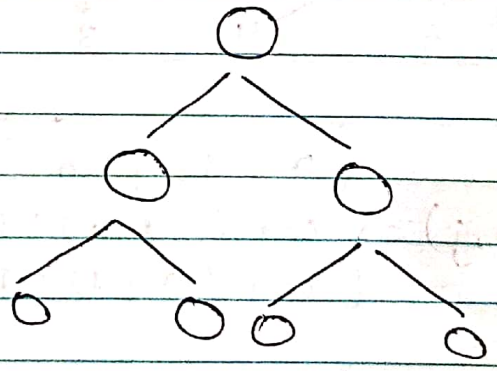
Optimal Binary Search Tree:

Difference between Binary Tree and Binary Search Tree.

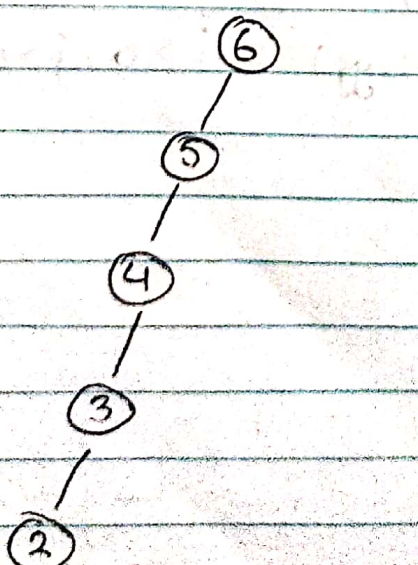
Binary Tree	Binary Search Tree
No constraints on placement of values within child nodes	Strict constraints have to be followed for placement of values within child nodes



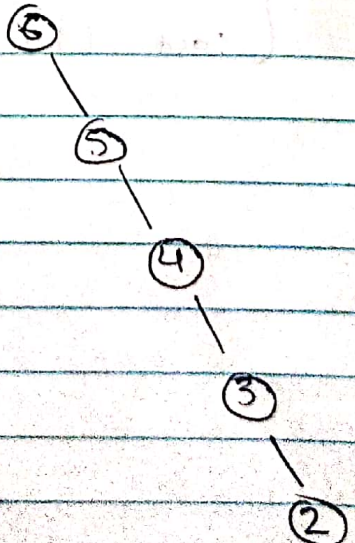
In complete Binary Tree



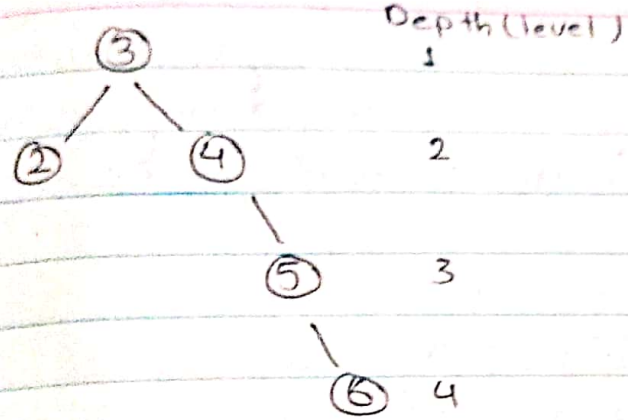
Complete Binary Tree

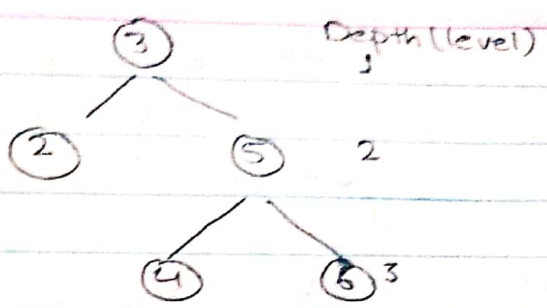


Left-Skewed BST



Right-Skewed BST





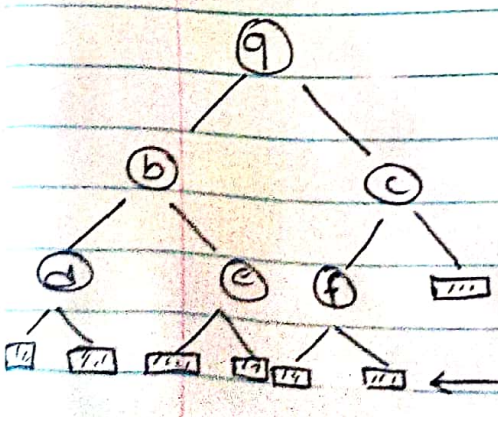
Let us suppose n is the number of keys.
Total no. of Structures (BST) that can be ~~formed~~ formed is

$$\frac{2n!}{n!(n+1)!} \quad \left(\frac{^{2n}C_n}{n+1} \right)$$

Suppose n = 4keys

$$\text{Total Structure} = \frac{8!}{4! \times 5!}$$

$$= 14$$



dummy keys.

p_i = Success probability
q_i = Failure probability

$$\sum p_i + \sum q_i = 1$$

Let k_1, k_2, k_3, k_4, k_5 be the keys provided $k_1 < k_2 < k_3 < k_4 < k_5$. Place the dummy keys yourself. Success and failure probability are as follows

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

Let $W[i,j]$ is the weight and $e[i,j]$ be the expected search cost of OBST

Table 1

$$W[i,j] = W[i,j-1] + p_j + q_j \text{ for } j \geq i$$
$$= q_{i-1} \text{ for } j = i-1$$

Table 2.

$$e[i,j] = p_r + e[i,r-1] + e[r+1,j] + W[i,j]$$
for $j \geq 1$
q_{i-1} for $j < i$

Table 3

Root Table = r

W[i,j] → j / ↓ i	0	1	2	3	4	5
1	q_0 0.05	0.30	0.45	0.55	0.70	1.0
2		q_1 0.10	0.25	0.35	0.50	0.8
3			q_2 0.05	0.15	0.30	0.6
4				q_3 0.05	0.20	0.5
5					q_4 0.05	0.35
6						q_5 0.10

W[1,0]

i = 1

j = 0

i.e. j = i - 1

= 1 - 1

= 0

W[i,j] = q_{i-1}

= q_{1-1}

= q_0

W[1,1] = W[1, 1-1] + P_1 + q_1

= W[1,0] + 0.15 + 0.10

= 0.05 + 0.15 + 0.10

= 0.30

W[2,2] = W[2, 2-1] + P_2 + q_2

= W[2,1] + 0.10 + 0.05

= 0.10 + 0.10 + 0.05

= 0.25

W[1,2] = W[1, 2-1] + P_2 + q_2

= W[1,1] + 0.10 + 0.05

= 0.30 + 0.10 + 0.05

= 0.45

W[3,3] = W[3, 3-1] + P_3 + q_3

= W[3,2] + 0.05 + 0.05

= 0.05 + 0.05 + 0.05

= 0.15

W[4,4] = W[4, 4-1] + P_4 + q_4

= W[4,3] + 0.10 + 0.05

= 0.05 + 0.10 + 0.05

= 0.20

W[5,5] = W[5, 5-1] + P_5 + q_5

= W[5,4] + 0.20 + 0.10

= 0.05 + 0.20 + 0.10

= 0.35

e j → / i ↓	0	1	2	3	4	5
1	q_0 0.05	r=1 0.45	r=1 0.9	r=2 1.25	r=2 1.75	r=2 2.75
2		0.10	r=2 0.4	r=2 0.7	r=2 1.2	r=4 2
3			0.05	r=3 0.25	r=4 0.6	r=5 1.3
4				0.05	r=4 0.3	r=5 0.9
5					0.05	r=5 0.5
6						0.10

$e[1,0]$ $i=1$, $j=0$; $j<i$

$= q_{1-1}$

$= q_0$

$e[1,1]$ $i=1$, $j=1$ $r=1$

$= e[1,1-1] + e[1+1,1] + w[i,j]$

$= e[1,0] + e[2,1] + w[1,1]$

$= 0.05 + 0.10 + 0.30$

$= 0.45$

$e[2,2]$: $i=2$, $j=2$ $r=2$

$= e[2,2-1] + e[2+1,2] + w[2,2]$

$= e[2,1] + e[3,2] + w[2,2]$

$= 0.10 + 0.05 + 0.25$

$= 0.4$

$e[1,2]$ $i=1$, $j=2$ $r=1$, $r=2$

$r=1$

$e[1,2] = e[1,0] + e[2,2] + W[1,2]$

$= [0.05) + 0.4 + 0.45$

$= 0.9$

$r=2$

$e[1,2] = e[1,1] + e[3,2] + W[1,2]$

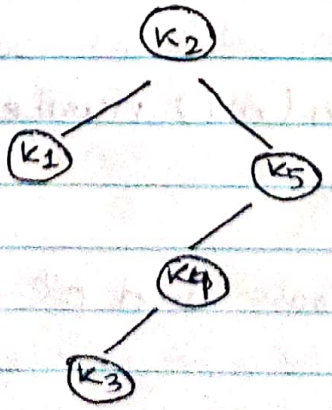
$= 0.45 + 0.05 + 0.45$

$= 0.95$

min = 0.9. ie. $r=1$

Root-Table

	1	2	3	4	5
1	1	1	2	2	2
2		2	3	2	4
3			3	4	5
4				4	5
5					5



Knapsack Problem

0/1 Knapsack Problem:

In 0/1 knapsack problem

- Items are indivisible i.e. we cannot take the fraction of any item.
- We have to either take an item completely or leave it completely.
- It is solved using dynamic programming approach.

Step for solving 0/1 Knapsack Problem Approach

- Consider we are given,
- A knapsack of weight capacity 'w'
- 'n' number of items each having some weight and value

Step 1

- Draw a table say 'T' with (n+1) number of rows and (w+1) number of columns.
- Fill all the boxes of 0^{th} row and 0^{th} column with zero as shown:

	0	1	2	3	..... W
0	0	0	0	0	0
1	0				
2	0				
⋮	0				
	0				
n	0				

Step 2:

- Start filling the table rowwise top to bottom from left to right.
- Use the following formula:

$T(i, j) = \max \{T(i-1, j), (value) + T(i-1, j - weight)\}$

$T(i, j)$ = maxium value of the selected item if we can take item 1 to i and we have weight restrictions of j.

Step 3:

After filling the table completely, value of the last box represents the maximum possible value to put in knapsack

Step 4:

To identify the items that must be put in the knapsack to obtain the maximum profit.

- Considering the last column of the table, start scanning the entries from bottom to top.

- If an entry is encountered whose value is not same as the value which is stored in the entry immediately above it, then mark the row lable of that entry.

- After scanning the entries, the marked label represent the items that must be put in the knapsack.

Q. Describe how dynamic programming strategy can be used to solve 0/1 knapsack problem.

- Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack.

- In ~~order~~ other words, given two integer arrays val[0..n-1] and wt[0....n-1] which represents values and weights associated with n items respectively.

- Also given an integer W which represents knapsack capacity, find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to W.

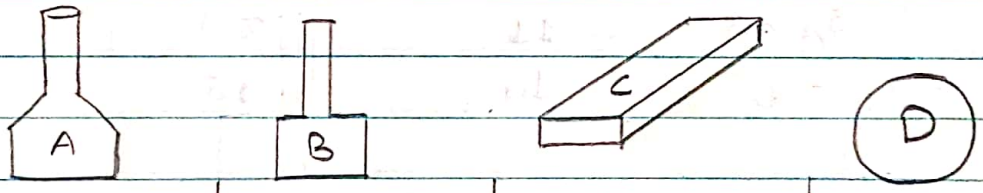
- You cannot break an item, either pick the complete item

or don't pick it (0-1 property)

Lets take an example:

- Suppose there are four different items named 'A', 'B', 'C', and 'D'

-- The maximum load capacity is 15.



Weight	5	10	8	6
Value	15	3	6	9

Maximum Load : 15

To make the concept clear about knapsack let us suppose that a thief enters into a house with his knapsack to steal those mentioned items named A B C and D

The maximum capacity the thief can carry in order to escape safely after stealing those things is 15 (i.e. Wmax = 15)

Now we form the subset as

	Weight	Value
{A}	5	15
{B}	10	3
{C}	8	6
{D}	6	9
{A,B}	15	18
{A,C}	13	21
{A, D}	11	24
{C, D}	14	15

- Every thief tries to take the item having the maximum value. Since the item 'A' has maximum value with minimum weight so the thief grabs the item 'A' and search for other item that meets his maximum load capacity to carry the items and escape safely from the house

- The thief logically forms the above subset in his mind and comes to conclusion that it is profitable to carry the item 'A' and 'D' safely.

Therefore {A, D} is the optimal solution where value is ?

Algorithm

Weight : $W_1, W_2, \ldots, W_n$

Benifit : $B_1, B_2, \ldots, B_n$

Let $X_1, X_2, \ldots, X_n$ be the Boolean variables whose value can be 0/1.

Therefore: $B = \sum_{i=1}^{N} X_i B_i$

$$W = \sum_{i=1}^{n} X_i W_i$$

Then maximize Benefit 'B' within constraints W (Minimum capacity)

Assumptions:

$$B(0, w) = 0$$
$$B(i, 0) = 0$$
$$B(i, w) = \max \begin{cases} B(i-1, w) \\ B_i + B(i, w - w_i) \end{cases}$$

Example

Q. Find the optimal solution for 0/1 knapsack Problem making the use of Dynamic Approach

Given $n = 4$

maximum weight, $W = 5kg$

$(w_1, w_2, w_3, w_4) : (2, 3, 4, 5)$ (Items)

$(b_1, b_2, b_3, b_4) : (3, 4, 5, 6)$ (Value/Probit)

Solution

We have: Knapsack capacity (W) = 5kg

Number of Items (n) = 4

Step 1:

- Draw a table say 'T' with (n+1) = 4+1 = 5 numbers of rows and (W+1) = 6 number of columns.

- Fill the boxes of the 0th rows and 0th column with 0.

Pi	Wi	i \ j	0	1	2	3	4	5	← (Bag's Capaci
		0	0	0	0	0	0	0	
3	2	1	0	0	3	3	3	3	
4	3	2	0	0	3	4	4	7	
5	4	3	0	0	3	4	5	7	
6	5	4	0	0	3	4	5	7	

↗ Item (w+1)

fig: T-Table

Item selected = 1, 2

ie. W_1 & W_2 are items and value is 3, 4 = 7

Step2 :

Start filling the table rowwise top to bottom from left to right

i) $T(1,1)$ $i=1, j=1$

Value = 3

Weight = 2

$T(1,1) = \max\{T(1-1,1), 3 + T(1-1, 1-2)$

$= \max\{T(0,1), 3 + T(0,-1)$ // ignore negative

$= 0$

ii) $T(1,2)$: $i=1, j=2$

Value : 3

Weight : 2

$T(1,2) = \max\{T(1-1,2), 3 + T(1-1, 2-2)\}$

$= \max\{T(0,2), 3 + T(0,0)\}$

$= \max\{(0,3)\}$

$= 3$

iii) $T(1,3)$ $i=1$ $j=3$ Value: 3, Weight = 2

$= \max\{T(1-1,3), 3 + T(1-1, 3-2)\}$

$= \max\{T(0,3), 3 + T(0,1)\}$

$= \max(0, 3+0)$

$= 3$

$T(1,4) = \max\{T(1-1,4), (value) + T(1-1, 4-weight)\}$

$= \max\{T(0,4), 3 + T(0, 4-2)\}$

$= \max\{T,(0,4), 3\}$

$= \max\{0, 3\}$

$= 3$

$T(1,5) = \max\{T\{(1-1,5), (value) + T(1-1, 5-weight)\}$

$= \max\{T(0,5), 3 + T(0, 5-2)\}$

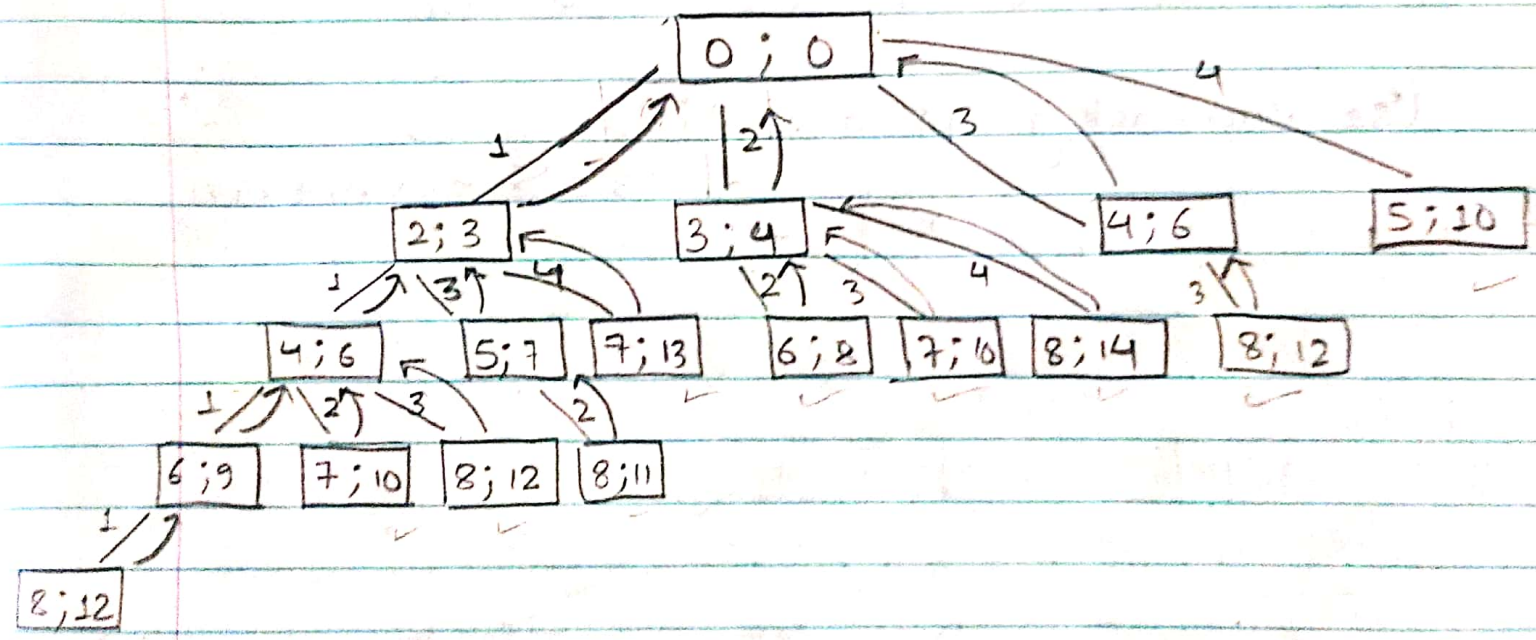
$= \max\{0, 3\}$

$= 3$

Backtracking Approach of Knapsack Problem:

Items	1	2	3	4
Weights	2	3	4	5
Profit	3	4	6	10

$Wt = 8$ $\qquad \sum_{i=1}^{k} w_i \leq wt$



(0, 1, 0, 1)

W (3 + 5) = 8

P (4 + 10) = 14

Backtracking Approach of Subset Sum Problem

- n values are given $\{x_1, x_2, x_3 \ldots x_n\}$
- find subset of n elements such that Sum = S

Example :

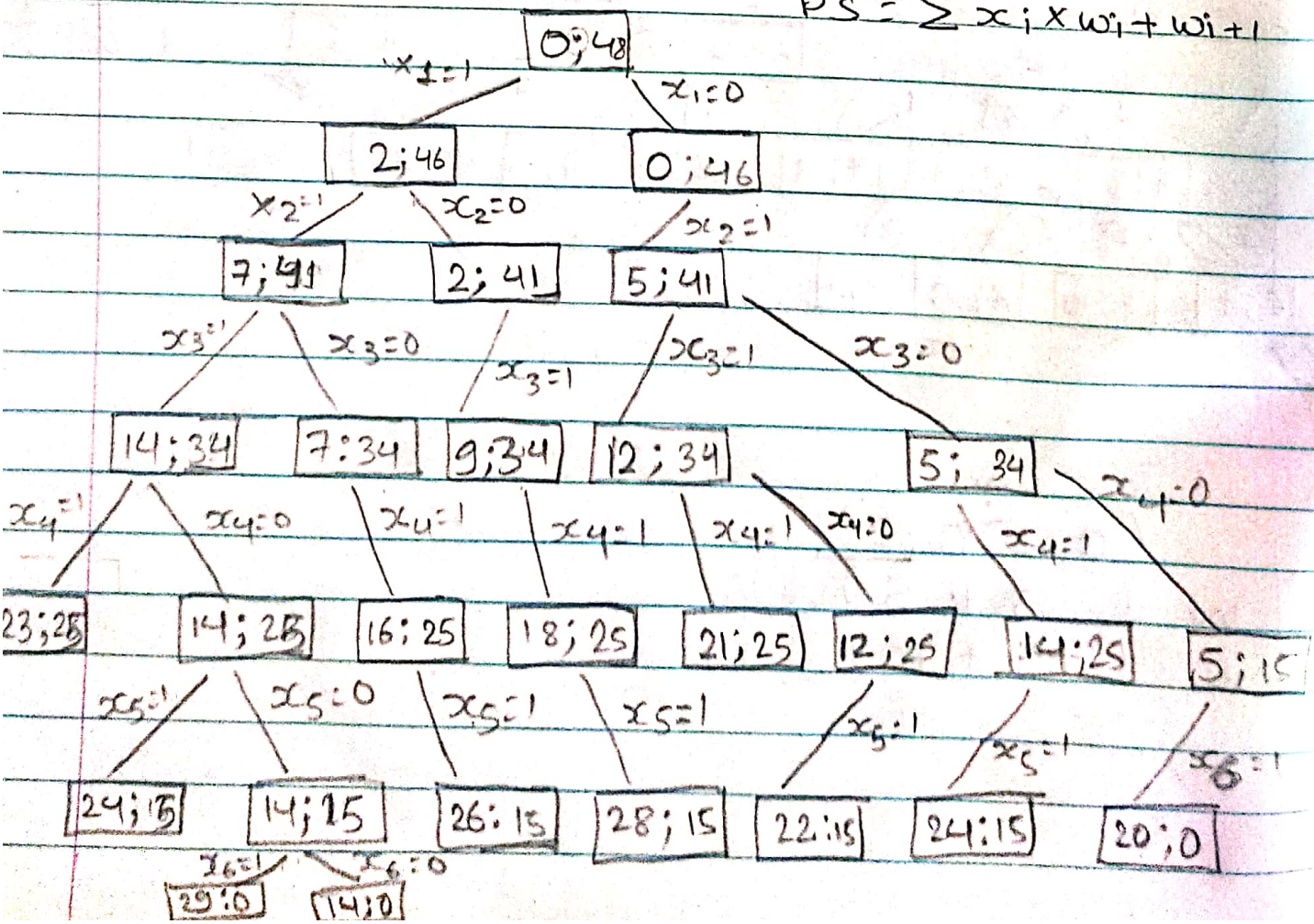
	x_1	x_2	x_3	x_4	x_5	x_6
Wi	2	5	7	9	10	15

S = 20

	x_1	x_2	x_3	x_4	x_5	x_6
	1	2	3	4	5	6
Wi	2	5	7	9	10	15

Use Backtracking Approach

$$PS = \sum x_i \times w_i + w_{i+1}$$



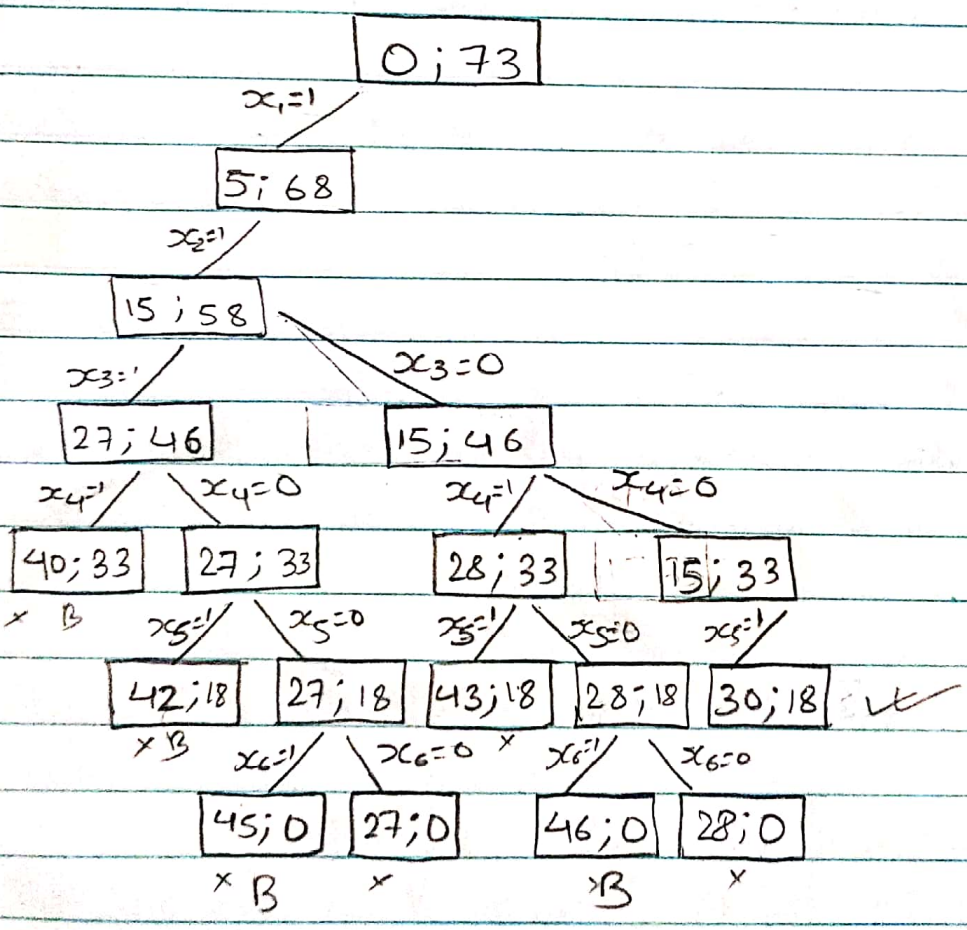
x	0	1	0	0	0	1
	1	2	3	4	5	6

i.e. 5 + 15 = 20

Example

Given, w[1:6] = { 5, 10, 12, 13, 15, 18 }
(1 2 3 4 5 6)

n = 6 S = 30



From above, we get

x	1	1	0	0	1	0
	1	2	3	4	5	6

(1, 2, 5)

i.e $5 + 10 + 15 = 3$

$$\sum_{i=1}^{k} w_i x_i + w_{k+1} \leq S \quad \text{and} \quad \sum_{i=k}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i > S$$

Unit 3

Online and PRAM Algorithms

Online Algorithms

- Which have to make decision 'online' without knowing the entire input.

- formally, an online algorithm receive a sequence of request $\sigma = \sigma(1), \sigma(2), \ldots, \sigma(m)$, these request must be served in the order of occurance. When serving the request $\sigma(t)$, an online algorithm doesnot know request $\sigma(t')$ with $t' > t$

- main measure of quality for online algorithms are their competitive ratio.

Example:

- In finance, when trading stocks we want to be able to predict, whether a stock's price will go up or down based on its history.

- At each moment we can thus characterize an online algorithm as the following:

1) The instance I has value in discrete time t when $t = 1, 2, \ldots, n$.

2) We gain new info at every moment t, eg every minute there is a new price to a particular stock.

3) At every moment t, we make a decision. eg. which stock to buy, sell or hold.

4) The objective is to either maximize/minimize a certain object function eg: maximizing profit.

Ski Rental Problem

- The Ski rental problem is a name given to a class of problem in which there is a choice between continuing to pay a repeating cost or paying a one-time cost which eliminates or reduces the repeating cost.

- Assume that you are taking ski lesson. After each lesson you decide (depending on how much you enjoy it and what is your bones status) whether to continue to ski or to stop totally.

- You have the choice of either renting skis for 1$ a time or buying skis for y$

- If you knew in advance how many times you would ski in your life then the choice of whether to rent or buy is simple: If you will ski more than y times then buy before you start, otherwise always rent.

- The cost of this algorithm min (t, y)

Conclusion to general rule

- When balancing small incremental costs against a big one-time cost, you want to delay spending the big cost until you have accumulated roughly the same amount in small costs.

- This type of strategy, with perfect knowledge of future, is known as an offline strategy.

- In practice you don't know how many times you will ski. What should you do?

- An online strategy will be a number k such that after renting k-1 time you will buy skis (just before your k^{th} visit).

- Claim: Setting k=y guarantees that you never pay more than twice the cost of the offline strategy.

- Example: Assume Y=7$ Thus, after 6 rents you buy. You total payment: 6+7=13$.

Online Strategy

Theorem:

- Setting k=y guarantees that you never pay more than twice the cost of the offline strategy.

- Proof: When you buy skis in your k^{th} visit, even if you quit right after this time $t >= y$.
  - Your total payment is $k-1+y = 2y-1$
  - The offline cost is $\min(t, y)$
  - The ratio is $(2y-1)/y = 2-(1/y)$

- We say that this strategy is $[2-(1/y)]$- competitive.

Load Balancing

- In computing, load balancing improves the distribution of workloads across multiple computing resources such as computers, a computer cluster, network links, central processing units or disk drives.

- Load balancing aims to optimize resource use, maximize throughput, minimize response time, and avoid overload of any single resource.

- Using multiple components with load balancing instead of a single component may increase reliability and availability through redundancy.

- Load balancing usually involves dedicated software or hardware, such as a multilayer switch or a Domain Name System server process.

- Load balancing differs from channel bonding in that load balancing divides traffic between network interfaces on a network socket (OSI Model Layer 4) basis, while channel bonding implies a division of traffic between physical interfaces at a lower level, either per packet (OSI model Layer 3) or on a data link (OSI model layer 2) basis with a protocol like shortest path bridging.

- The distribution of process to multiple check point as nodes or servers so that the traffic of operations can be controlled without stressing the system.

- There are two important requirements of online servers provides:

i) availability
ii) redundancy

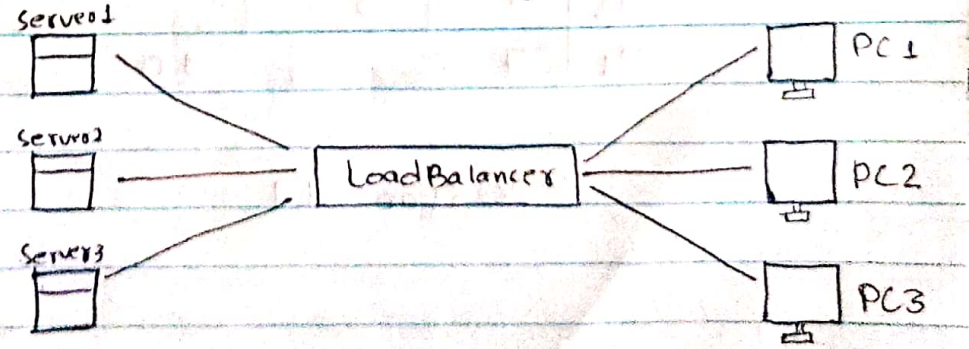
Both can be optimized in effective manner through load balancing.

- The application domain of load balancing are:

i) Servers
ii) Multiplexer
iii) Scheduler

- Following algorithm are commonly used:

i) Round Robin (Weighted Round Robin and Dynamic Round Robin)
ii) Response time
iii) Random



Online Scheduling and Load Balancing

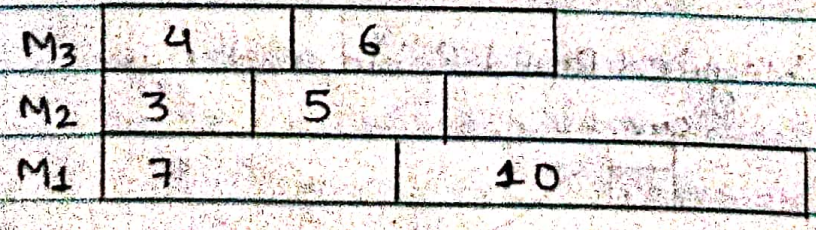
- Problem Statement:
  - A set of m identical machines.
  - A sequence of jobs with processing times $P_1, \ldots$
  - Each job must be assigned to one of the machines.
  - When job j is scheduled, we don't know how many and what are additional jobs we are going to have and what are their processing times.

- Goal: schedule the jobs on machines in a way that
  - minimizes the makespan.

List Scheduling

- A greedy algorithm: always schedule a job on the least loaded machine.

- Example: m=3 σ: 7 3 4 5 6 10



Make span = 17

Paging and Caching

Paging

- In computer operating system, paging is a memory management scheme by which a computer stores and retrieves data from secondary storage for use in main memory.

- In this scheme, the operating system retrieves data from secondary storage in same size blocks called pages.

- Paging is an important part of virtual memory implementations in modern operating systems, using secondary storage to let programs exceed the size of available physical memory.

- For simplicity, main memory is called "RAM" (an acronym of "random-access memory") and secondary storage is called "disk" (a shorthand for "hard disk drive"), but the concepts do not depend on whether these terms apply literally to a specific computer system.

Caching

- In computing, a cache is a hardware or software component that stores data so that further requests for that data can be ~~sever~~ served faster.

- The data stored in a cache might be the result of an earlier

computation or a copy of data stored elsewhere.

- A cache hit occurs when the requested data can be found in a cache, while a cache miss occurs when it cannot.

- Cache hits are served by reading data from the cache which is faster than recomputing a result or reading a slower data store; thus, the more requests that can served from the cache, the faster the system performs.

- To be cost-effective and to enable efficient use of data caches must be relatively small.

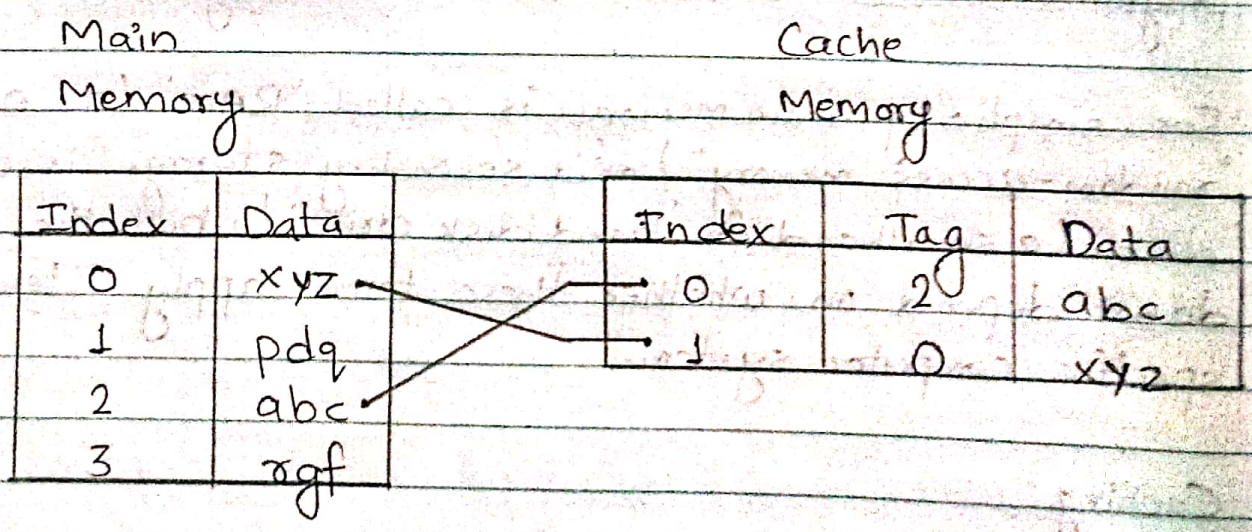


Fig: Diagram of a CPU memory Cache operation

Paging - Cache Replacement Policies

- Problem Statement
  - There are two levels of memory:
    - fast memory M_1 consisting of k pages (cache)
    - Slow memory M_2 consisting of n pages ($k<n$)
  - Pages in M_1 are a strict subset of the pages in M_2
  - Pages are accessible only through M_1
  - Accessing a page contained in M_1 has cost 0.
  - When accessing a page not in M_1 it must first be brought from M_2 at a cost of 1 before in can be accessed.
  - This event is called a page fault.
  - If M_1 is full when a page fault occurs, some page in M_1 must be evicted in order to make room in M_1
  - How to choose a page to evict each time a page fault occurs in a way the that minimizes the total number of page faults over time?

Paging - An Optimal Offline Algorithm

LFD - Longest forward Distance

- LFD is always optimal so also known as optimal page replacement policy.

Proof idea: For any other algorithm A, the cost of A is not increased if in the 1^{st} time that A differs from LFD we evict

in A the page that is requested farthest in the future.

- However, LFD is not practical.
- It is not an online algorithm.