Greedy Algorithm Introduction

"Greedy Method finds out of many options, but you have to choose the best option."

In this method, we have to find out the best method/option out of many present ways.

In this approach/method we focus on the first stage and decide the output, don't think about the future.

This method may or may not give the best output.

Greedy Algorithm solves problems by making the best choice that seems best at the particular moment. Many optimization problems can be determined using a greedy algorithm. Some issues have no efficient solution, but a greedy algorithm may provide a solution that is close to optimal. A greedy algorithm works if a problem exhibits the following two properties:

1. **Greedy Choice Property:** A globally optimal solution can be reached at by creating a locally optimal solution. In other words, an optimal solution can be obtained by creating "greedy" choices.
2. **Optimal substructure:** Optimal solutions contain optimal subsolutions. In other words, answers to subproblems of an optimal solution are optimal.

Example:

1. machine scheduling
2. Fractional Knapsack Problem
3. Minimum Spanning Tree
4. Huffman Code
5. Job Sequencing
6. Activity Selection Problem

Steps for achieving a Greedy Algorithm are:

1. **Feasible:** Here we check whether it satisfies all possible constraints or not, to obtain at least one solution to our problems.
2. **Local Optimal Choice:** In this, the choice should be the optimum which is selected from the currently available
3. **Unalterable:** Once the decision is made, at any subsequence step that option is not altered.

An Activity Selection Problem

The activity selection problem is a mathematical optimization problem. Our first illustration is the problem of scheduling a resource among several challenge activities. We find a greedy algorithm provides a well designed and simple method for selecting a maximum- size set of manually compatible activities.

Suppose S = {1, 2....n} is the set of n proposed activities. The activities share resources which can be used by only one activity at a time, e.g., Tennis Court, Lecture Hall, etc. Each Activity "i" has **start time** s_i and a **finish time** f_i, where $s_i \leq f_i$. If selected activity "i" take place meanwhile the half-open time interval $[s_i,f_i)$. Activities i and j are **compatible** if the intervals (s_i, f_i) and $[s_i, f_i)$ do not overlap (i.e. i and j are compatible if $s_i \geq f_i$ or $s_i \geq f_i$). The activity-selection problem chosen the maximum- size set of mutually consistent activities.

Algorithm Of Greedy- Activity Selector:

GREEDY- ACTIVITY SELECTOR (s, f)

```
1. n ← length [s]
2. A ← {1}
3. j ← 1.
4. for i ← 2 to n
5. do if $s_i \geq f_i$
6. then A ← A ∪ {i}
7. j ← i
8. return A
```

Example: Given 10 activities along with their start and end time as

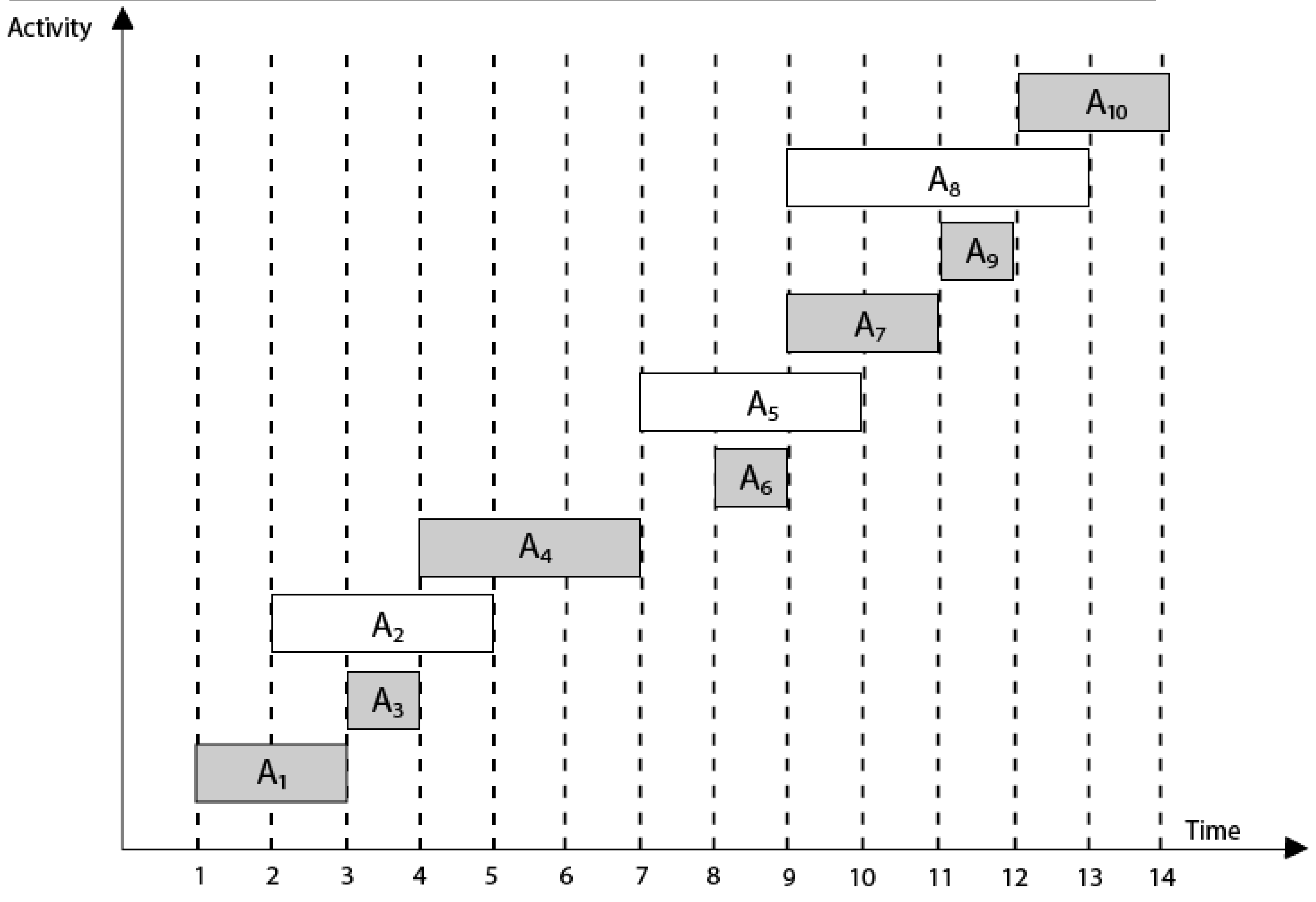
```
S = ($A_1$ $A_2$ $A_3$ $A_4$ $A_5$ $A_6$ $A_7$ $A_8$ A9 A10)
Si = (1,2,3,4,7,8,9,9,11,12)
fi = (3,5,4,7,10,9,11,13,12,14)
```

Compute a schedule where the greatest number of activities takes place.

Solution: The solution to the above Activity scheduling problem using a greedy strategy is illustrated below:

Arranging the activities in increasing order of end time

Activity	A_1	A_3	A_2	A_4	A_6	A_5	A_7	A_9	A_8	A_{10}
Start	1	3	2	4	8	7	9	11	9	12
Finish	3	4	5	7	9	10	11	12	13	14



Now, schedule A_1

Next schedule A_3 as A_1 and A_3 are non-interfering.

Next **skip** A_2 as it is interfering.

Next, schedule A_4 as A_1 A_3 and A_4 are non-interfering, then next, schedule A_6 as A_1 A_3 A_4 and A_6 are non-interfering.

Skip A_5 as it is interfering.

Next, schedule A_7 as A_1 A_3 A_4 A_6 and A_7 are non-interfering.

Next, schedule A_9 as A_1 A_3 A_4 A_6 A_7 and A_9 are non-interfering.

Skip A_8 as it is interfering.

Next, schedule A_{10} as A_1 A_3 A_4 A_6 A_7 A_9 and A_{10} are non-interfering.

Thus the final Activity schedule is:

$$\boxed{(A_1\ A_3\ \ A_4\ A_6\ \ A_7\ \ A_9\ A_{10})}$$

Fractional Knapsack

Fractions of items can be taken rather than having to make binary (0-1) choices for each item.

Fractional Knapsack Problem can be solvable by greedy strategy whereas 0 - 1 problem is not.

Steps to solve the Fractional Problem:

1. Compute the value per pound $^{v_i}/_{w_i}$ for each item.
2. Obeying a Greedy Strategy, we take as possible of the item with the highest value per pound.
3. If the supply of that element is exhausted and we can still carry more, we take as much as possible of the element with the next value per pound.

4. Sorting, the items by value per pound, the greedy algorithm run in O (n log n) time.

```
Fractional Knapsack (Array v, Array w, int W)
1. for i= 1 to size (v)
2. do p [i] = v [i] / w [i]
3. Sort-Descending (p)
4. i ← 1
5. while (W>0)
6. do amount = min (W, w [i])
7. solution [i] = amount
8. W= W-amount
9. i ← i+1
10. return solution
```

Example: Consider 5 items along their respective weights and values: -

I = (I_1,I_2,I_3,I_4,I_5)

w = (5, 10, 20, 30, 40)

v = (30, 20, 100, 90,160)

The capacity of knapsack W = 60

Now fill the knapsack according to the decreasing value of p_i.

First, we choose the item I_i whose weight is 5.

Then choose item I_3 whose weight is 20. Now,the total weight of knapsack is 20 + 5 = 25

Now the next item is I_5, and its weight is 40, but we want only 35, so we chose the fractional part of it,

i.e., $5 \times \frac{5}{5} + 20 \times \frac{20}{20} + 40 \times \frac{35}{40}$

Weight = 5 + 20 + 35 = 60

Maximum Value:-

$$30 \times \frac{5}{5} + 100 \times \frac{20}{20} + 160 \times \frac{35}{40}$$

= 30 + 100 + 140 = 270 (Minimum Cost)

Solution:

ITEM	w_i	v_i
I_1	5	30
I_2	10	20
I_3	20	100
I_4	30	90
I_5	40	160

Taking value per weight ratio i.e. $p_i = \frac{v_i}{w_i}$

ITEM	w_i	v_i	$P_i = \frac{v_i}{w_i}$
I_1	5	30	6.0
I_2	10	20	2.0
I_3	20	100	5.0
I_4	30	90	3.0
I_5	40	160	4.0

Now, arrange the value of p_i in decreasing order.

ITEM	w_i	v_i	$p_i = \frac{v_i}{w_i}$
I_1	5	30	6.0
I_3	20	100	5.0
I_5	40	160	4.0
I_4	30	90	3.0
I_2	10	20	2.0

Huffman Codes

- (i) Data can be encoded efficiently using Huffman Codes.
- (ii) It is a widely used and beneficial technique for compressing data.
- (iii) Huffman's greedy algorithm uses a table of the frequencies of occurrences of each character to build up an optimal way of representing each character as a binary string.

Suppose we have 10^5 characters in a data file. Normal Storage: 8 bits per character (ASCII) - 8 x 10^5 bits in a file. But we want to compress the file and save it compactly. Suppose only six characters appear in the file:

	a	b	c	d	e	f	Total
Frequency	45	13	12	16	9	5	100

How can we represent the data in a Compact way?

(i) Fixed length Code: Each letter represented by an equal number of bits. With a fixed length code, at least 3 bits per character:

For example:

a	000
b	001
c	010
d	011
e	100

f	101

For a file with 10^5 characters, we need 3×10^5 bits.

(ii) A variable-length code: It can do considerably better than a fixed-length code, by giving many characters short code words and infrequent character long codewords.

For example:

a	0
b	101
c	100
d	111
e	1101
f	1100

```
Number of bits = (45 x 1 + 13 x 3 + 12 x 3 + 16 x 3 + 9 x 4 + 5 x 4) x 1000
```

= 2.24 x 10^5bits

Thus, 224,000 bits to represent the file, a saving of approximately 25%.This is an optimal character code for this file.

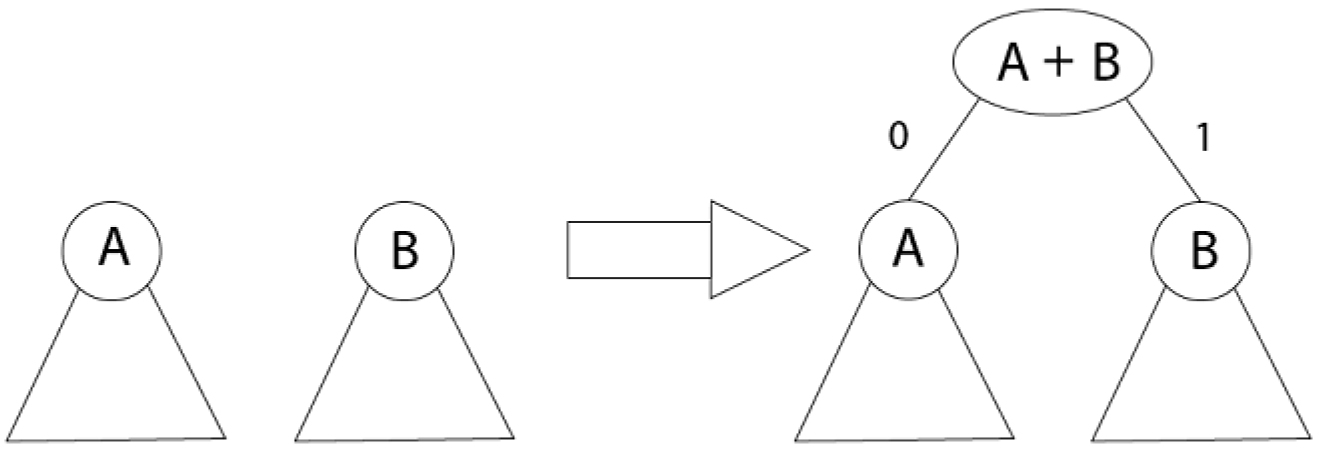
Prefix Codes:

The prefixes of an encoding of one character must not be equal to complete encoding of another character, e.g., 1100 and 11001 are not valid codes because 1100 is a prefix of some other code word is called prefix codes.

Prefix codes are desirable because they clarify encoding and decoding. Encoding is always simple for any binary character code; we concatenate the code words describing each character of the file. Decoding is also quite comfortable with a prefix code. Since no codeword is a prefix of any other, the codeword that starts with an encoded data is unambiguous.

Greedy Algorithm for constructing a Huffman Code:

Huffman invented a greedy algorithm that creates an optimal prefix code called a Huffman Code.



The algorithm builds the tree T analogous to the optimal code in a bottom-up manner. It starts with a set of |C| leaves (C is the number of characters) and performs |C| - 1 'merging' operations to create the final tree. In the Huffman algorithm 'n' denotes the quantity of a set of characters, z indicates the parent node, and x & y are the left & right child of z respectively.

Algorithm of Huffman Code

Huffman (C)

```
1. n=|C|
2. Q ← C
3. for i=1 to n-1
4. do
5. z= allocate-Node ()
6. x=  left[z]=Extract-Min(Q)
7. y= right[z] =Extract-Min(Q)
8. f [z]=f[x]+f[y]
9. Insert (Q, z)
10. return Extract-Min (Q)
```

Example: Find an optimal Huffman Code for the following set of frequencies:

1. a: 50 b: 25 c: 15 d: 40 e: 75

Solution:

Given that: C = {a, b, c, d, e}

f (C) = {50, 25, 15, 40, 75}

n = 5

Q ← c

i.e.

c	15	b	25	d	40	a	50	e	75

for i ← 1 to 4

i = 1 Z ← Allocate node

x ← Extract-Min (Q)

y ← Extract-Min (Q)

c	15	b	25	d	40	a	50	e	75

Left [z] ← x

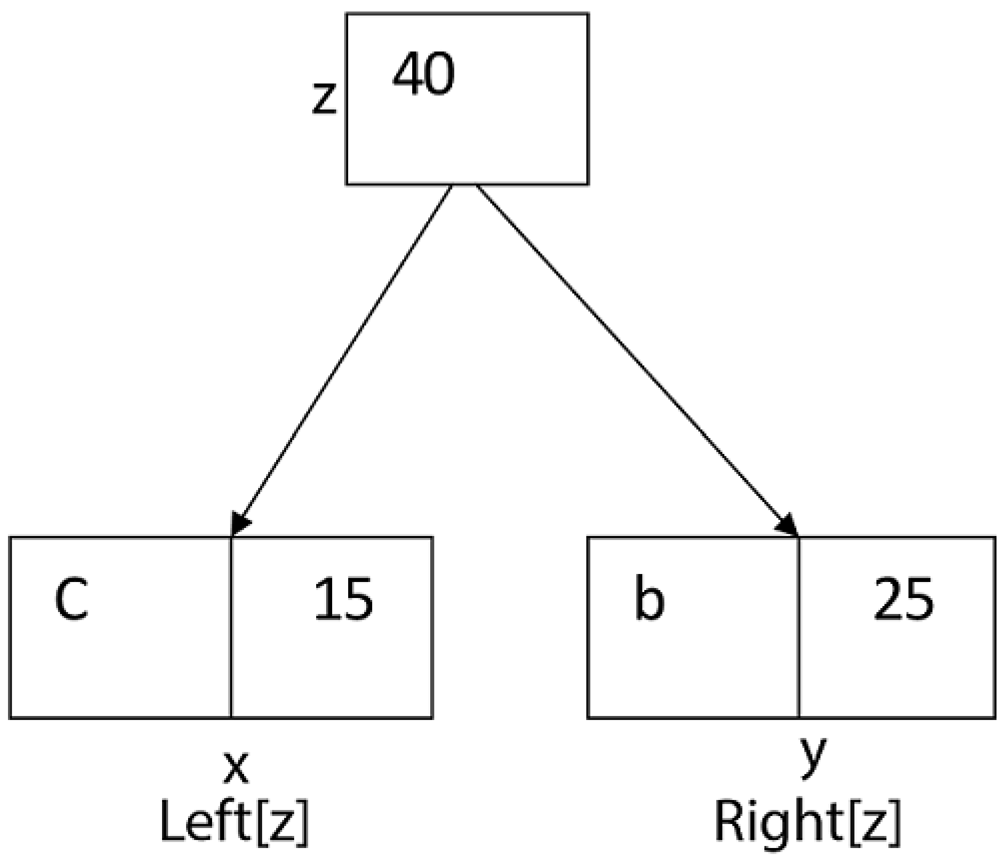
Right [z] ← y

f (z) ← f (x) + f (y) = 15 + 25

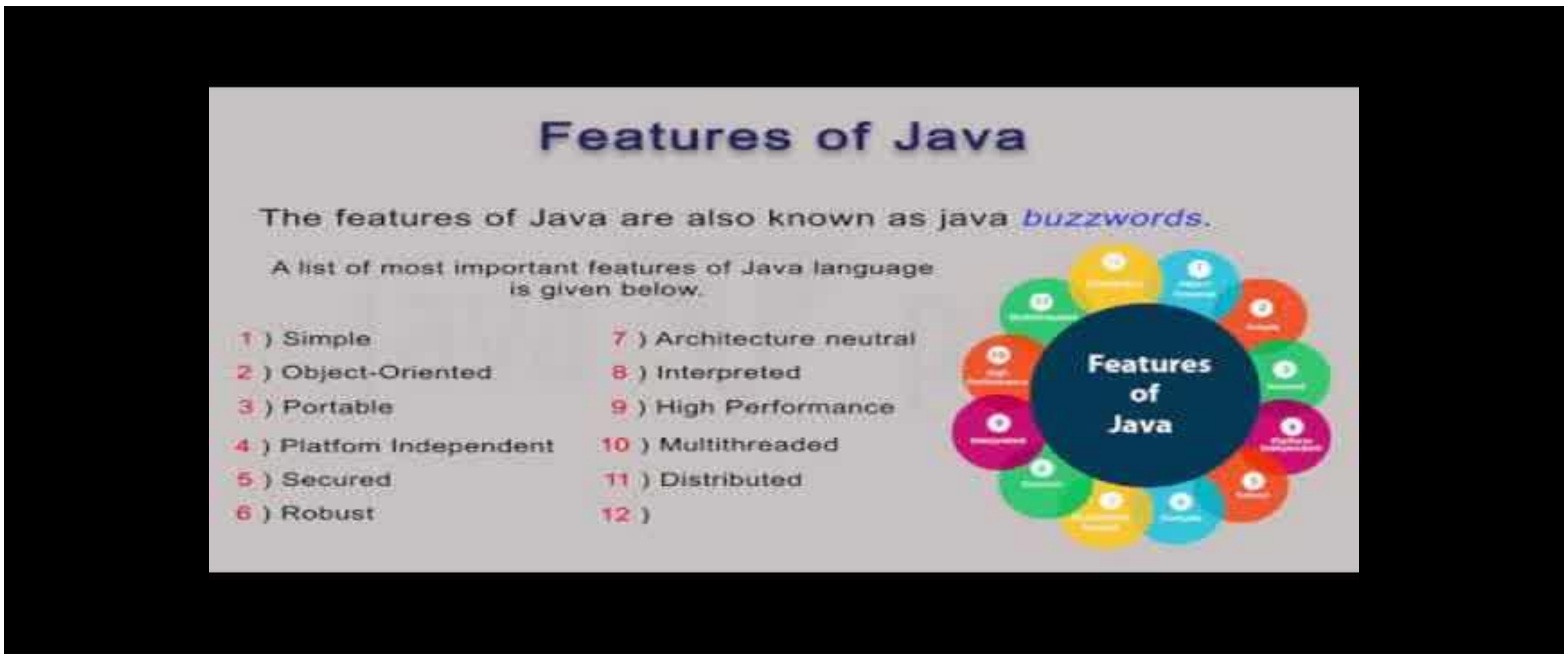
f (z) = 40

d	40	a	50	e	75

i.e.



Again for i=2



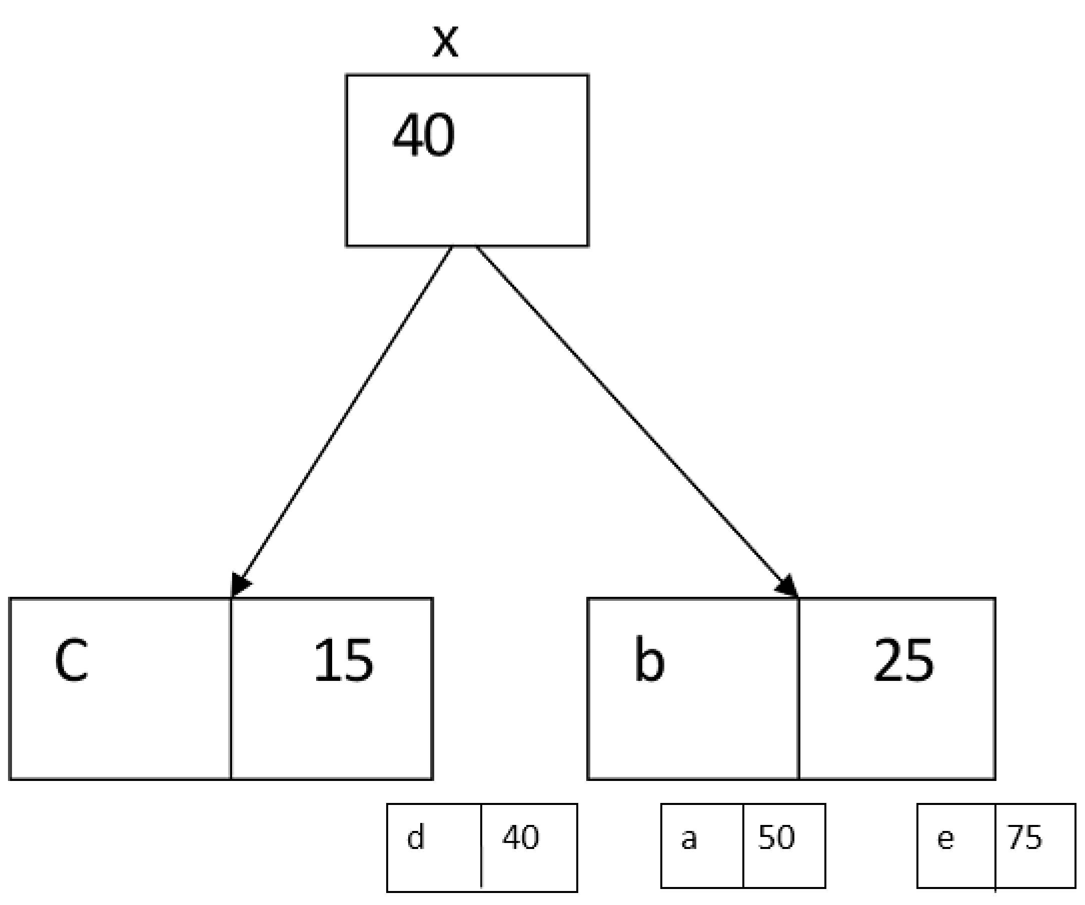
Features of Java

The features of Java are also known as java *buzzwords*.

A list of most important features of Java language is given below.

1) Simple	7) Architecture neutral
2) Object-Oriented	8) Interpreted
3) Portable	9) High Performance
4) Platfom Independent	10) Multithreaded
5) Secured	11) Distributed
6) Robust	12)

Features of Java



$z \leftarrow$ Allocate node

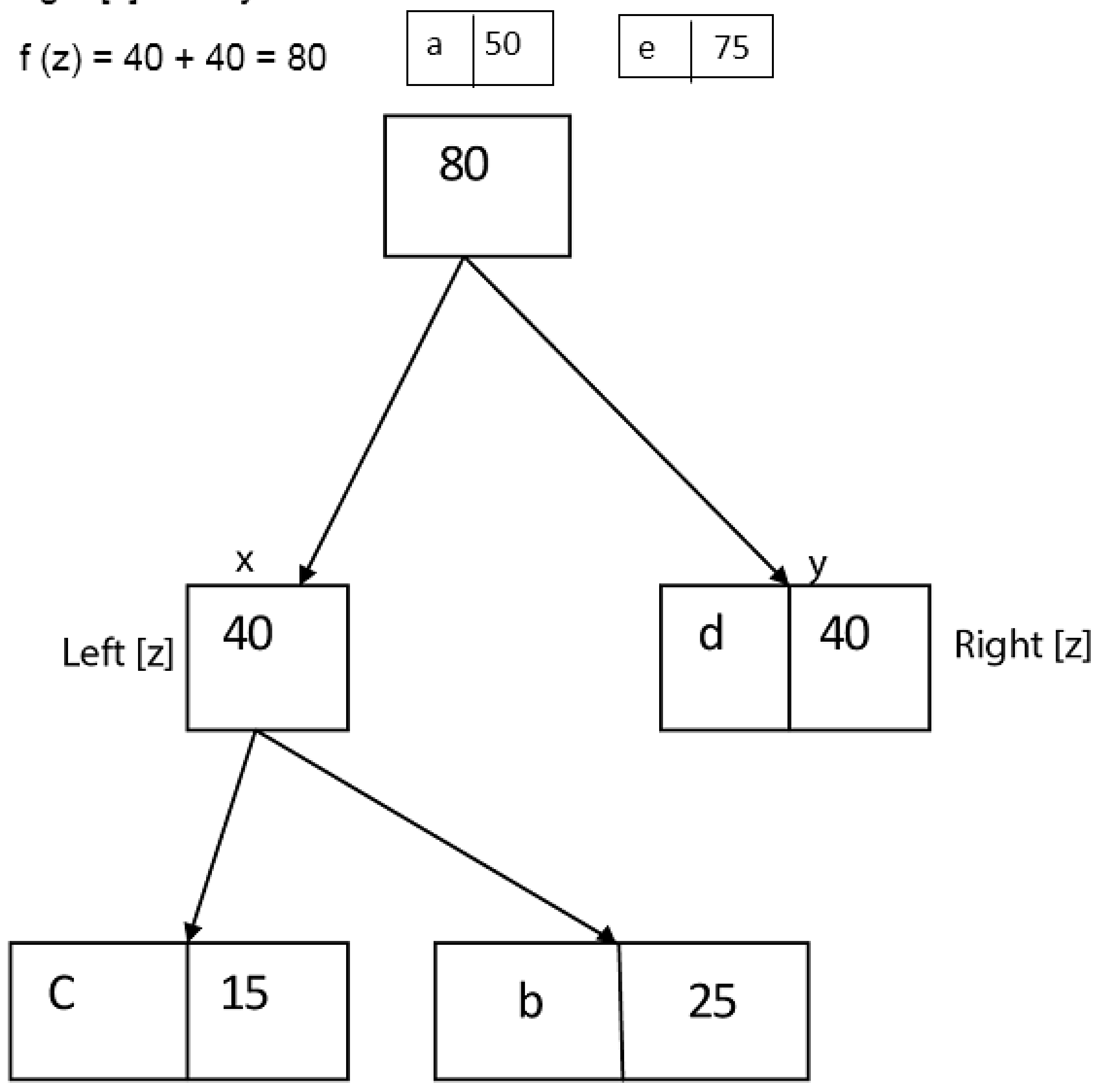
$x \leftarrow 40$

$y \leftarrow 40$

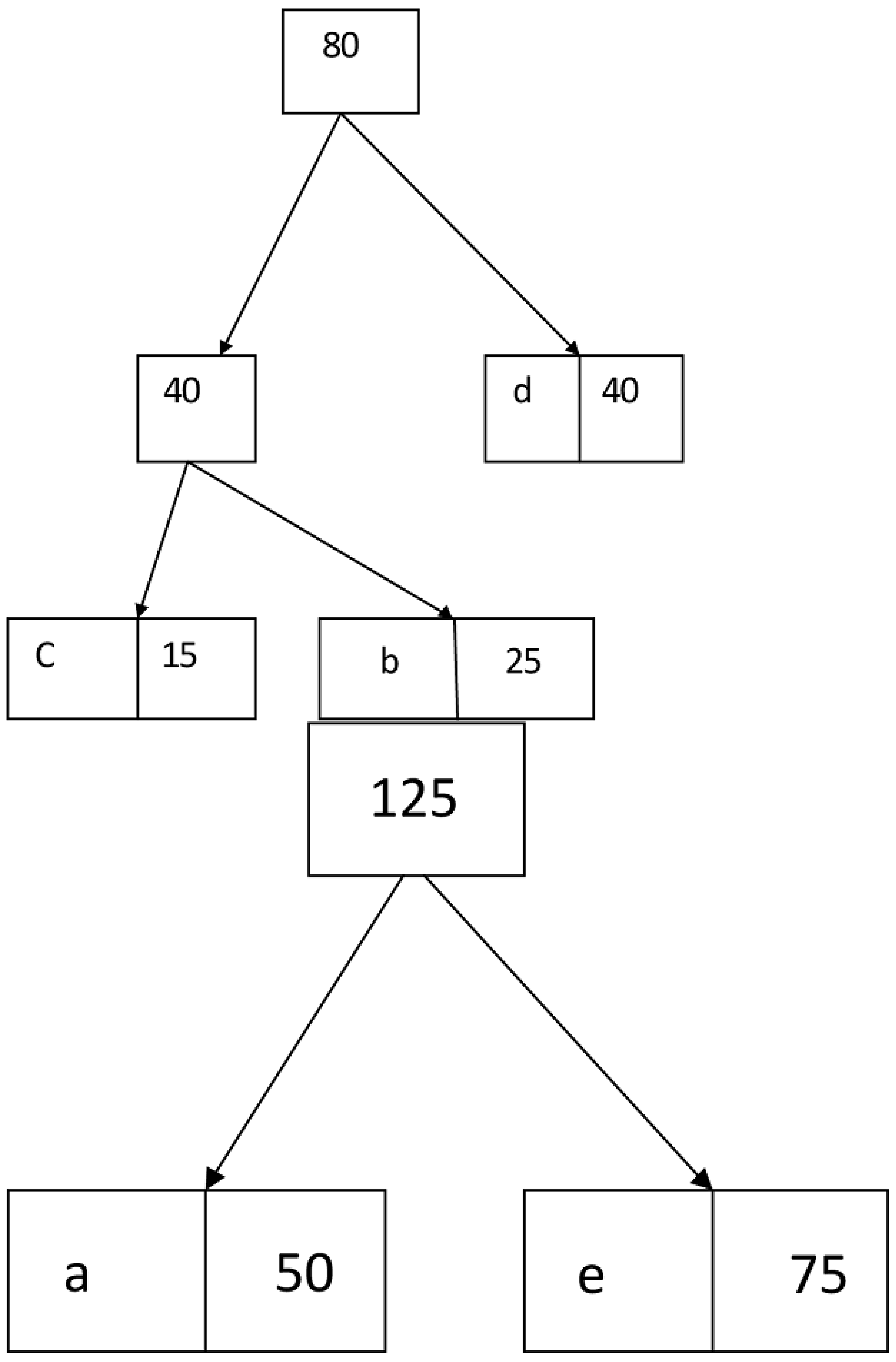
left [z] $\leftarrow$ x

right [z] $\leftarrow$ y

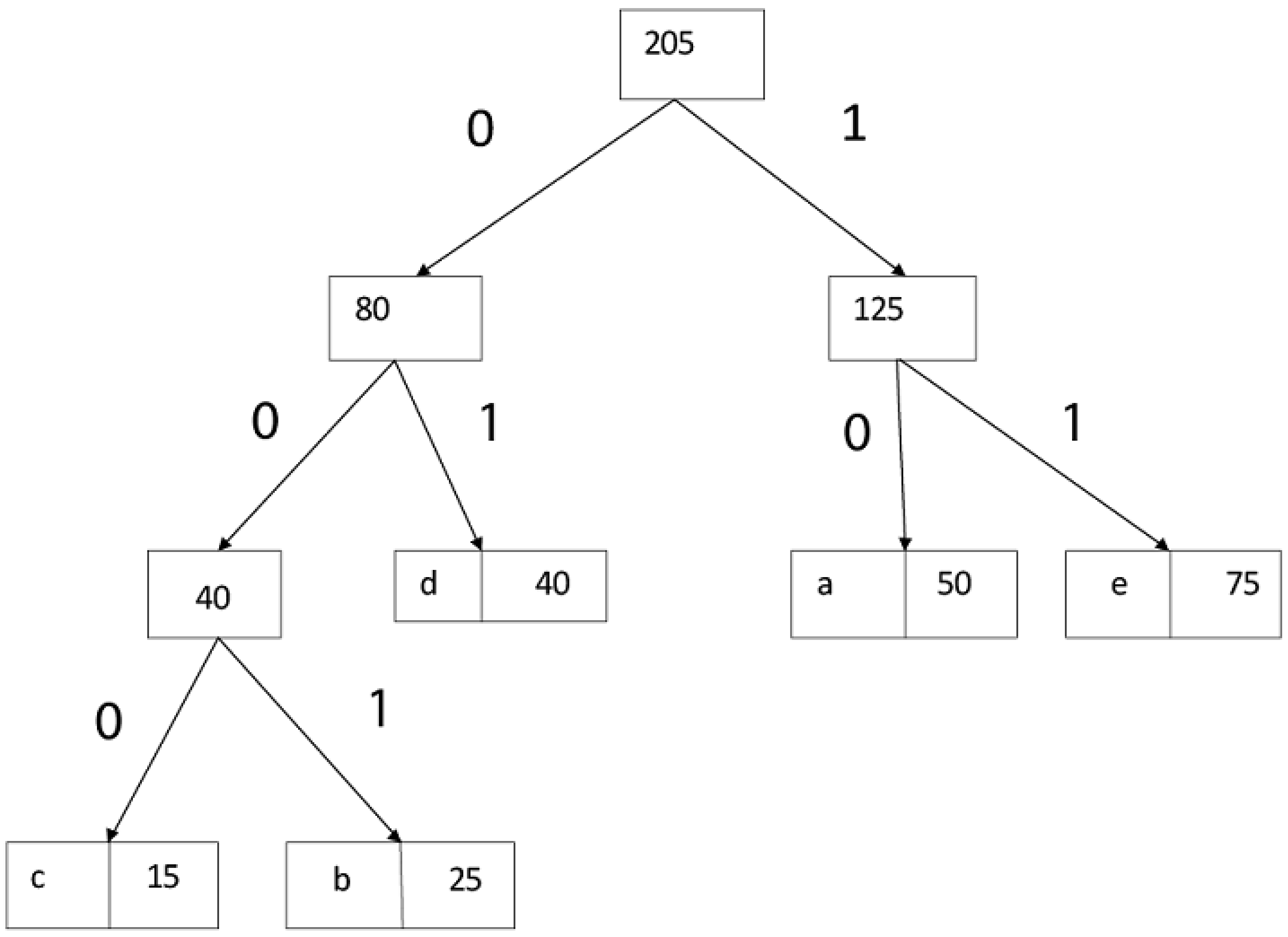
f (z) = 40 + 40 = 80



Similarly, we apply the same process we get



Thus, the final output is:



Activity or Task Scheduling Problem

This is the dispute of optimally scheduling unit-time tasks on a single processor, where each job has a deadline and a penalty that necessary be paid if the deadline is missed.

A unit-time task is a job, such as a program to be rush on a computer that needed precisely one unit of time to complete. Given a finite set S of unit-time tasks, a schedule for S is a permutation of S specifying the order in which to perform these tasks. The first task in the schedule starts at time 0 and ends at time 1; the second task begins at time 1 and finishes at time 2, and so on.

The dispute of scheduling unit-time tasks with deadlines and penalties for each processor has the following inputs:

- a set S = {1, 2, 3.....n} of n unit-time tasks.
- a set of n integer deadlines d_1 d_2 d_3...d_n such that d_i satisfies $1 \le d_i \le n$ and task i is supposed to finish by time d_i and
- a set of n non-negative weights or penalties w_1 w_2....w_n such that we incur a penalty of w_i if task i is not finished by time d_i, and we incurred no penalty if a task finishes by its deadline.

Here we find a schedule for S that minimizes the total penalty incurred for missed deadlines.

A task is **late** in this schedule if it finished after its deadline. Otherwise, the task is early in the schedule. An arbitrary schedule can consistently be put into **early-first form**, in which the first tasks precede the late tasks, i.e., if some new task x follows some late task y, then we can switch the position of x and y without affecting x being early or y being late.

An arbitrary schedule can always be put into a **canonical form** in which first tasks precede the late tasks, and first tasks are scheduled in order of nondecreasing deadlines.

A set A of tasks is **independent** if there exists a schedule for the particular tasks such that no tasks are late. So the set of first tasks for a schedule forms an independent set of tasks 'l' denote the set of all independent set of tasks.

For any set of tasks A, A is independent if for t = 0, 1, 2.....n we have $N_t(A) \leq t$ where $N_t(A)$ denotes the number of tasks in A whose deadline is t or prior, i.e. if the tasks in A are expected in order of monotonically growing deadlines, then no task is late.

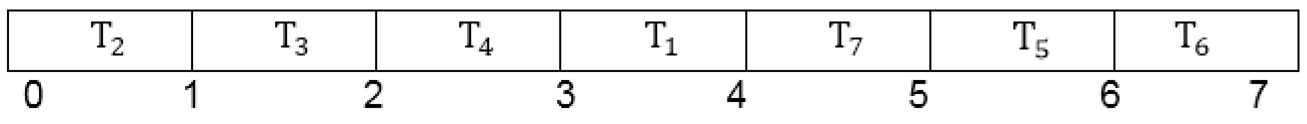
Example: Find the optimal schedule for the following task with given weight (penalties) and deadlines.

	1	**2**	**3**	**4**	**5**	**6**	**7**
d_i	4	2	4	3	1	4	6
w_i	70	60	50	40	30	20	10

Solution: According to the Greedy algorithm we sort the jobs in decreasing order of their penalties so that minimum of penalties will be charged.

In this problem, we can see that the maximum time for which uniprocessor machine will run in 6 units because it is the maximum deadline.

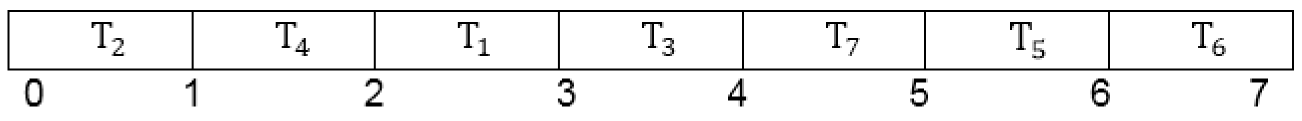
Let T_i represents the tasks where i = 1 to 7



T_5 and T_6 cannot be accepted after T_7 so penalty is

```
w5 + w6 = 30 + 20 = 50 (2 3 4 1 7 5 6)
```

Other schedule is



(2 4 1 3 7 5 6)

There can be many other schedules but (2 4 1 3 7 5 6) is optimal.

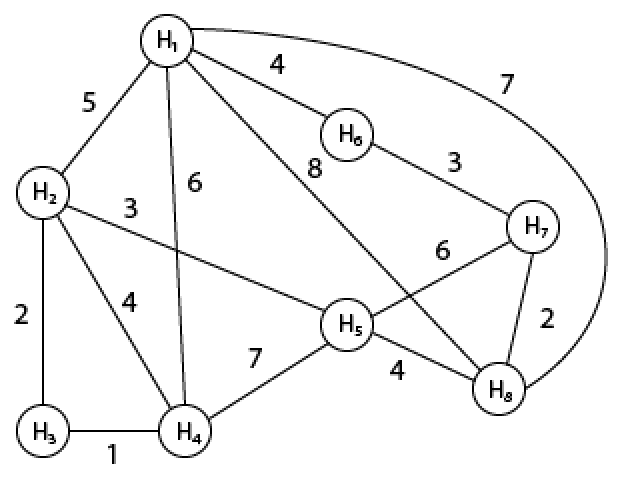
Travelling Sales Person Problem

The traveling salesman problems abide by a salesman and a set of cities. The salesman has to visit every one of the cities starting from a certain one (e.g., the hometown) and to return to the same city. The challenge of the problem is that the traveling salesman needs to minimize the total length of the trip.

Suppose the cities are x_1 x_2..... x_n where cost c_{ij} denotes the cost of travelling from city x_i to x_j. The travelling salesperson problem is to find a route starting and ending at x_1 that will take in all cities with the minimum cost.

Example: A newspaper agent daily drops the newspaper to the area assigned in such a manner that he has to cover all the houses in the respective area with minimum travel cost. Compute the minimum travel cost.

The area assigned to the agent where he has to drop the newspaper is shown in fig:



Solution: The cost- adjacency matrix of graph G is as follows:

$cost_{ij}$ =

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
H_1	0	5	0	6	0	4	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
H_6	4	0	0	0	0	0	3	0
H_7	0	0	0	0	6	3	0	2
H_8	7	0	0	0	4	0	2	0

The tour starts from area H_1 and then select the minimum cost area reachable from H_1.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
$\textcircled{H_1}$	0	5	0	6	0	$\textcircled{4}$	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
H_6	4	0	0	0	0	0	3	0
H_7	0	0	0	0	6	3	0	2
H_8	7	0	0	0	4	0	2	0

Mark area H_6 because it is the minimum cost area reachable from H_1 and then select minimum cost area reachable from H_6.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
$\textcircled{H_1}$	0	5	0	6	0	$\textcircled{4}$	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
$\textcircled{H_6}$	4	0	0	0	0	0	$\textcircled{3}$	0
H_7	0	0	0	0	6	3	0	2
H_8	7	0	0	0	4	0	2	0

Mark area H_7 because it is the minimum cost area reachable from H_6 and then select minimum cost area reachable from H_7.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
$\textcircled{H_1}$	0	5	0	6	0	$\textcircled{4}$	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
$\textcircled{H_6}$	4	0	0	0	0	0	$\textcircled{3}$	0
$\textcircled{H_7}$	0	0	0	0	6	3	0	$\textcircled{2}$
H_8	7	0	0	0	4	0	2	0

Mark area H_8 because it is the minimum cost area reachable from H_8.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
$\textcircled{H_1}$	0	5	0	6	0	$\textcircled{4}$	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
$\textcircled{H_6}$	4	0	0	0	0	0	$\textcircled{3}$	0
$\textcircled{H_7}$	0	0	0	0	6	3	0	$\textcircled{2}$
$\textcircled{H_8}$	7	0	0	0	$\textcircled{4}$	0	2	0

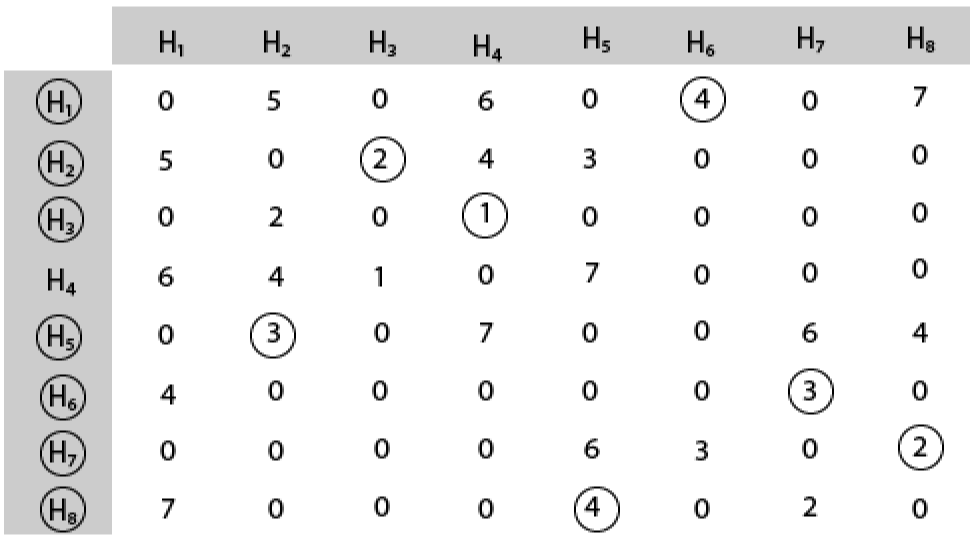
Mark area H_5 because it is the minimum cost area reachable from H_5.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
$\textcircled{H_1}$	0	5	0	6	0	$\textcircled{4}$	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
$\textcircled{H_5}$	0	$\textcircled{3}$	0	7	0	0	6	4
$\textcircled{H_6}$	4	0	0	0	0	0	$\textcircled{3}$	0
$\textcircled{H_7}$	0	0	0	0	6	3	0	$\textcircled{2}$
$\textcircled{H_8}$	7	0	0	0	$\textcircled{4}$	0	2	0

Mark area H_2 because it is the minimum cost area reachable from H_2.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
$\textcircled{H_1}$	0	5	0	6	0	$\textcircled{4}$	0	7
$\textcircled{H_2}$	5	0	$\textcircled{2}$	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
$\textcircled{H_5}$	0	$\textcircled{3}$	0	7	0	0	6	4
$\textcircled{H_6}$	4	0	0	0	0	0	$\textcircled{3}$	0
$\textcircled{H_7}$	0	0	0	0	6	3	0	$\textcircled{2}$
$\textcircled{H_8}$	7	0	0	0	$\textcircled{4}$	0	2	0

Mark area H_3 because it is the minimum cost area reachable from H_3.



	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
H_1	0	5	0	6	0	④	0	7
H_2	5	0	②	4	3	0	0	0
H_3	0	2	0	①	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	③	0	7	0	0	6	4
H_6	4	0	0	0	0	0	③	0
H_7	0	0	0	0	6	3	0	②
H_8	7	0	0	0	④	0	2	0

Mark area H_4 and then select the minimum cost area reachable from H_4 it is H_1.So, using the greedy strategy, we get the following.

$$H_1 \xrightarrow{4} H_6 \xrightarrow{3} H_7 \xrightarrow{2} H_8 \xrightarrow{4} H_5 \xrightarrow{3} H_2 \xrightarrow{2} H_3 \xrightarrow{1} H_4 \xrightarrow{6} H_1.$$

Thus the minimum travel cost = 4 + 3 + 2 + 4 + 3 + 2 + 1 + 6 = 25

Matroids:

A matroid is an ordered pair M(S, I) satisfying the following conditions:

1. S is a finite set.
2. I is a nonempty family of subsets of S, called the independent subsets of S, such that if B ∈ I and A ∈ I. We say that I is hereditary if it satisfies this property. Note that the empty set ∅ is necessarily a member of I.
3. If A ∈ I, B ∈ I and |A| <|B|, then there is some element x ∈ B ? A such that A∪{x}∈I. We say that M satisfies the exchange property.

We say that a matroid M (S, I) is weighted if there is an associated weight function w that assigns a strictly positive weight w (x) to each element x ∈ S. The weight function w extends to a subset of S by summation:

$$w(A) = \sum_{x \in A} w(x)$$

for any A ∈ S.

Differentiate between Dynamic Programming and Greedy Method

Dynamic Programming	**Greedy Method**
1. Dynamic Programming is used to obtain the optimal solution.	1. Greedy Method is also used to get the optimal solution.
2. In Dynamic Programming, we choose at each step, but the choice may depend on the solution to sub-problems.	2. In a greedy Algorithm, we make whatever choice seems best at the moment and then solve the sub-problems arising after the choice is made.
3. Less efficient as compared to a greedy approach	3. More efficient as compared to a greedy approach
4. Example: 0/1 Knapsack	4. Example: Fractional Knapsack
5. It is guaranteed that Dynamic Programming will generate an optimal solution using Principle of Optimality.	5. In Greedy Method, there is no such guarantee of getting Optimal Solution.