Course: Analysis of Algorithms
Code: CS33104
Branch: MCA -3rd Semester

Lecture – 5: Greedy Programming Technique

Faculty & Coordinator : Dr. J Sathish Kumar (JSK)

Department of Computer Science and Engineering

Motilal Nehru National Institute of Technology Allahabad,
Prayagraj-211004

The Greedy Method: Introduction

- ***Humans/People are usually Greedy. It is seldom to find non-greedy people***

- *Greed works!!!!*

  We want to verify whether is it really so ?

- Take a number of computational problems investigate the pros & cons of short-sighted greed

- How to we define a greedy algorithm ?

Basic Greedy paradigm

- Builds up the small solution in small steps
- Choose a decision myopically (Lacking foresight or scope)
- To optimize some underlying criterion
- Choose what is best for the moment
- Typically works in stages

  1.a decision made in one stage **can’t be changed later (can’t be done)**

  2.the choice must lead to the feasible solution

  3.expected to be an optimal solution

Applications

- Optimal solutions
  - Huffman codes
  - Minimum Spanning Tree (MST)
  - Single-source shortest paths
  - Optimal Storage on Tapes
  - Container Loading problem
  - Change making
  - Simple scheduling problems
- Approximations
  - Knapsack problem
  - Traveling Salesman Problem (TSP)
  - other combinatorial optimization problems

Greedy algorithms

- Optimization problems
  - solved through a sequence of choices that are:
    - feasible
    - locally optimal
    - irrevocable
- Not all optimization problems can be approached in this manner!

Huffman Codes

Huffman Codes

- For compressing data (sequence of characters)
- Widely used
- Very efficient (saving 20-90%)
- Use a table to keep frequencies of occurrence of characters.
- Output binary string.

Huffman Codes

Example:

A file of 100,000 characters.
Containing only ‘a’ to ‘e’

	Frequency	Fixed-length codeword	Variable-length codeword
‘a’	45000	000	0
‘b’	13000	001	101
‘c’	12000	010	100
‘d’	16000	011	111
‘e’	9000	100	1101
‘f’	5000	101	1100

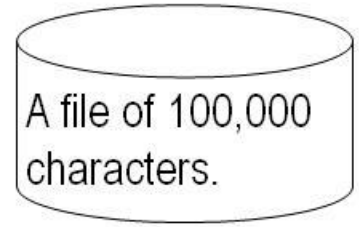
eg. “abc” = “000001010”

bits

eg. “abc” = “0101100”

1*45000 + 3*13000 + 3*12000 +
3*16000 + 4*9000 + 4*5000
= bits

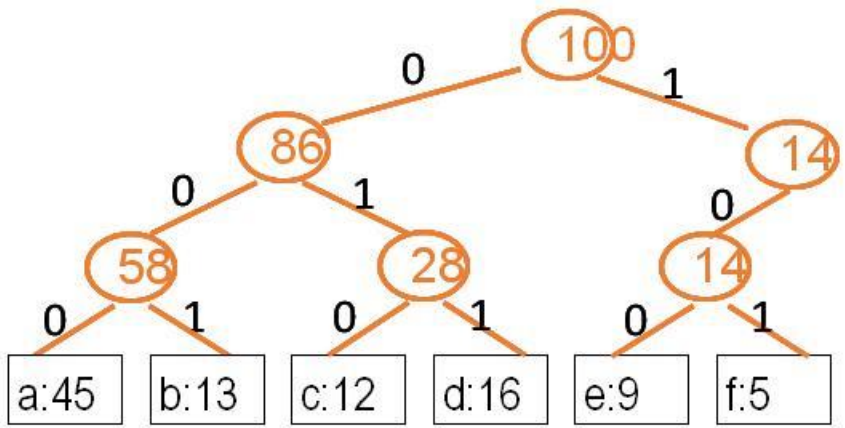
Huffman Codes

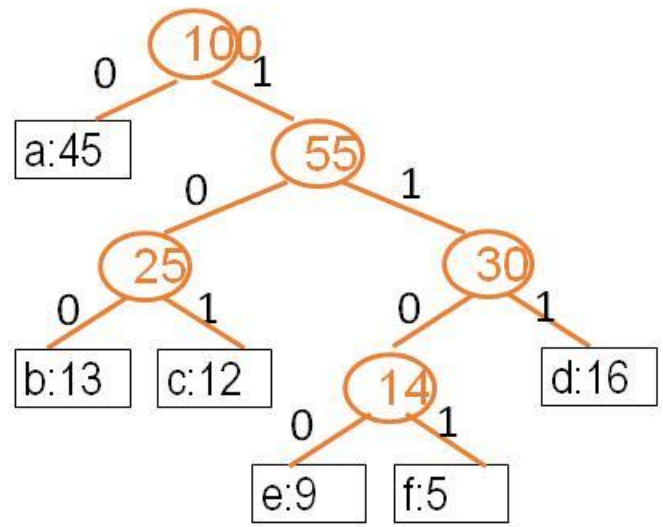


The coding schemes can be represented by trees:

	Frequency (in thousands)	Fixed-length codeword
'a'	45	000
'b'	13	001
'c'	12	010
'd'	16	011
'e'	9	100
'f'	5	101

	Frequency (in thousands)	Variable-length codeword
'a'	45	0
'b'	13	101
'c'	12	100
'd'	16	111
'e'	9	1101
'f'	5	1100





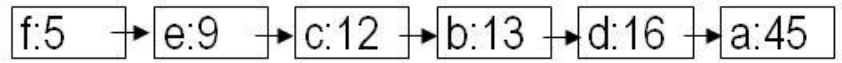
Huffman Codes

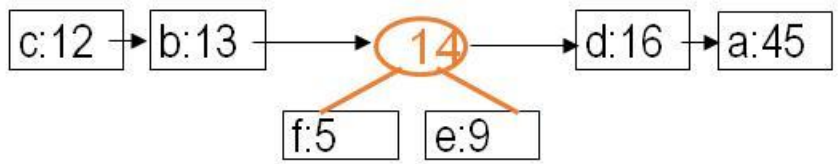
How do we find the optimal prefix code?

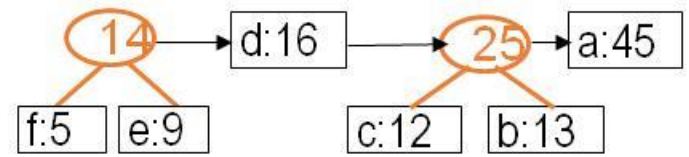
Huffman code (1952) was invented to solve it.

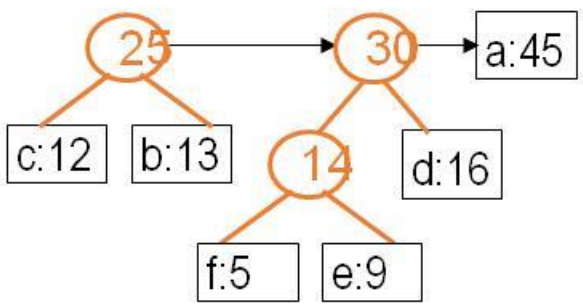
A Greedy Approach.

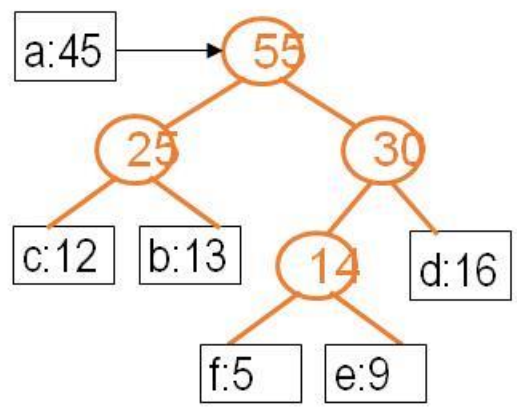
Q: A min-priority queue

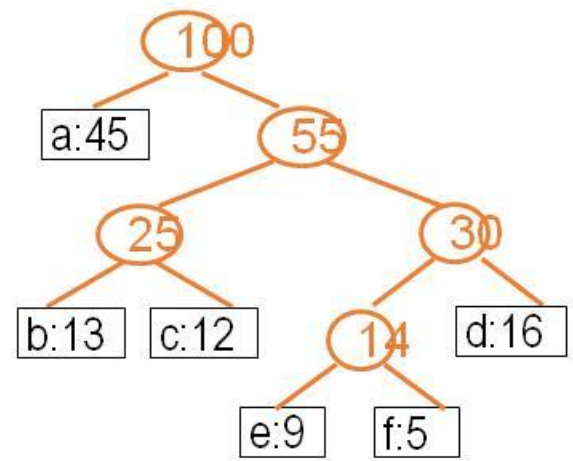






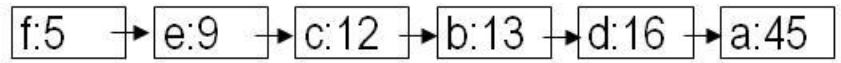


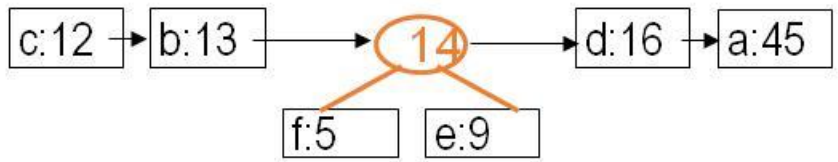


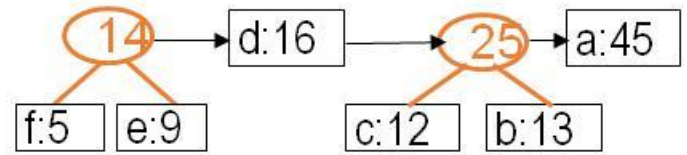


Huffman Codes

Q: A min-priority queue







....

```
HUFFMAN(C)
1  Build Q from C
2  For i = 1 to |C|-1
3          Allocate a new node z
4          z.left = x = EXTRACT_MIN(Q)
5          z.right = y = EXTRACT_MIN(Q)
6          z.freq = x.freq + y.freq
7          Insert z into Q in correct position.
8  Return EXTRACT_MIN(Q)
```

If Q is implemented as a binary min-heap,

"Build Q from C" is O(n)

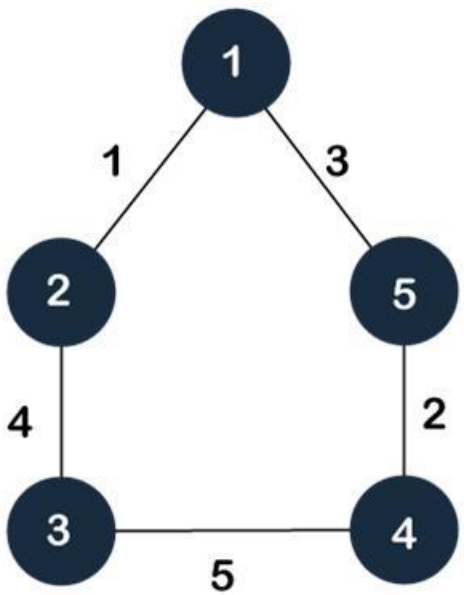
"EXTRACT_MIN(Q)" is O(lg n)

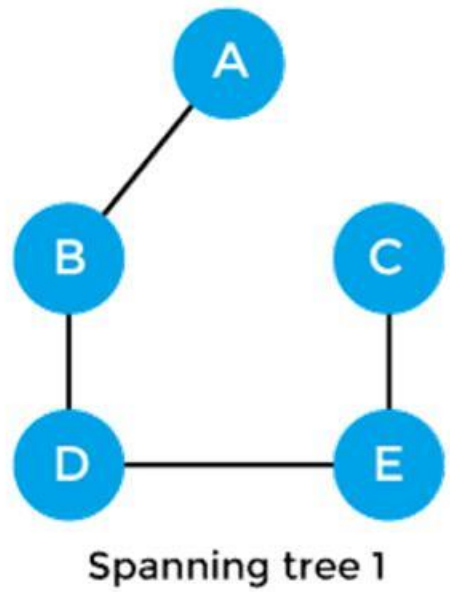
"Insert z into Q" is O(lg n)

Huffman(C) is O(n lg n)

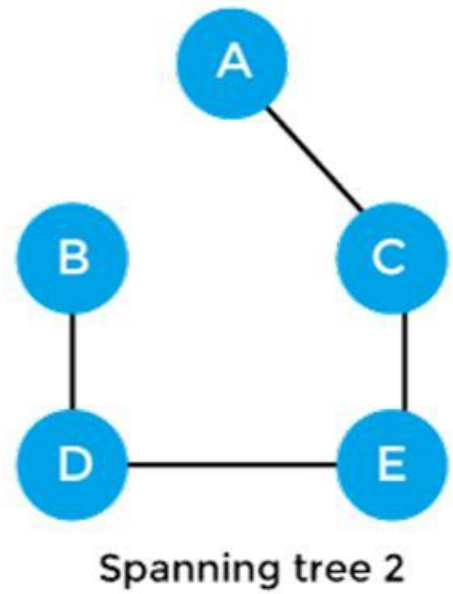
Minimum Spanning Tree

- A spanning tree of an undirected connected graph is its connected acyclic subgraph (i.e., a tree) that contains all the vertices of the graph.

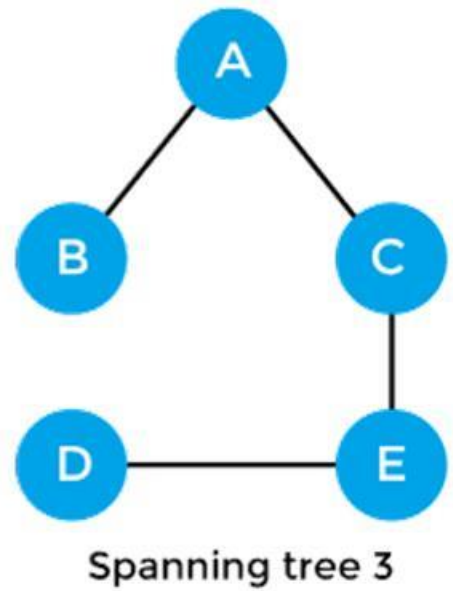




Spanning tree 1



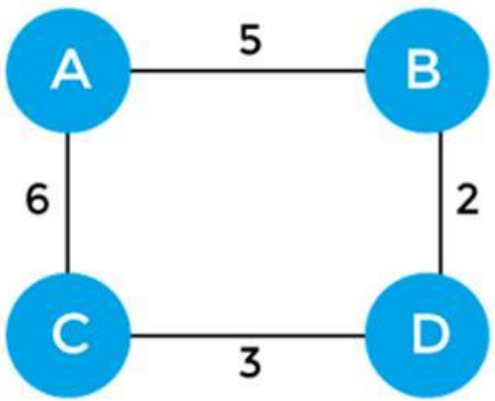
Spanning tree 2



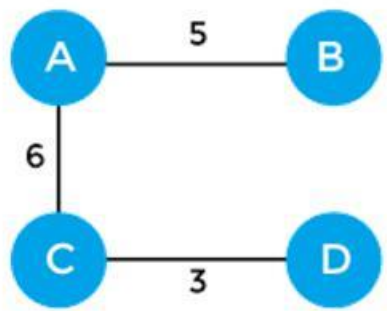
Spanning tree 3

Minimum Spanning Tree

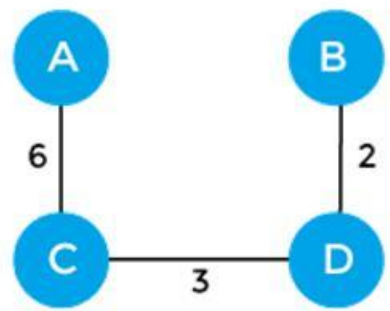
- If such a graph has weights assigned to its edges, a ***minimum spanning tree*** **is its spanning tree of the smallest weight, where the *weight* of a tree is defined as the sum of the weights on all its edges.**



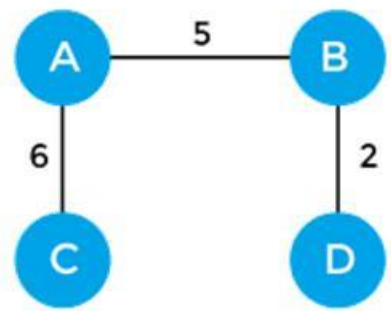
Weighted graph



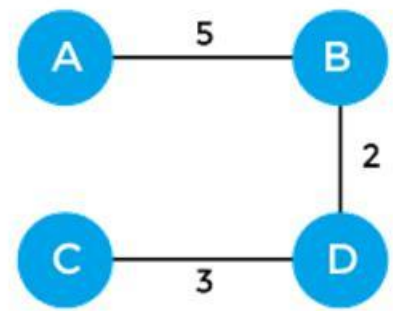
Sum = 14
Minimum spanning tree - 1



Sum = 11
Minimum spanning tree - 2



Sum = 13
Minimum spanning tree - 3

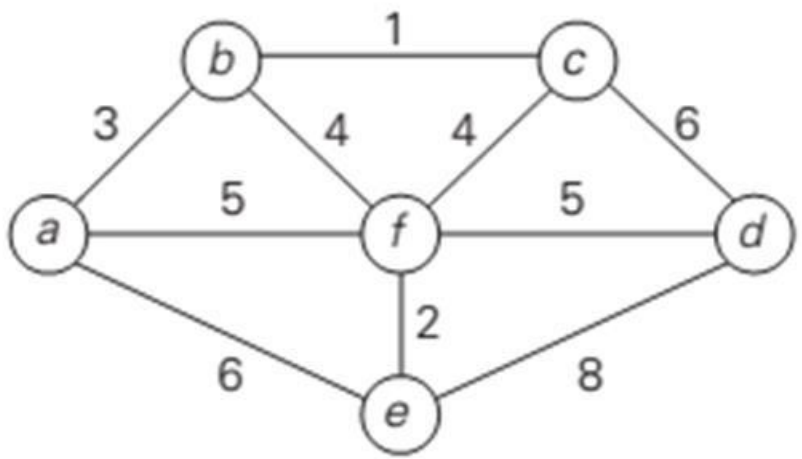


Sum = 10
Minimum spanning tree - 4

Applications of minimum spanning tree

- Minimum spanning tree can be used to design
  - water-supply networks,
  - telecommunication networks, and
  - electrical grids.
- It can be used to find paths in the map.

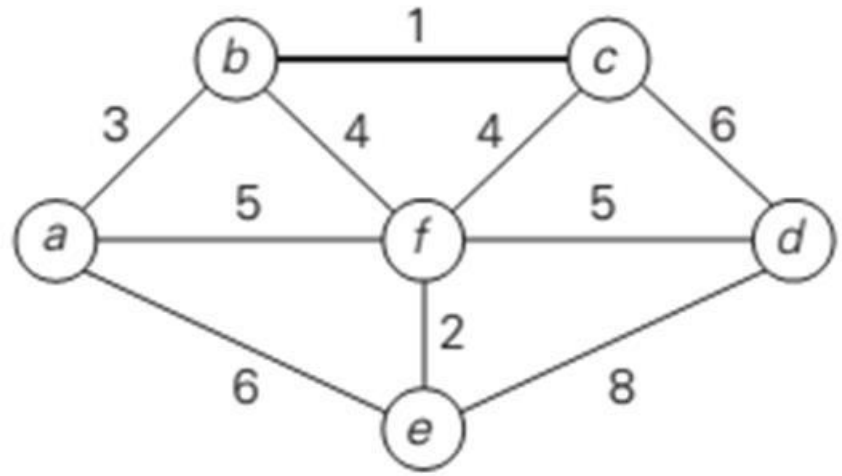
Kruskal's Algorithm



bc	ef	ab	bf	cf	af	df	ae	cd	de
1	2	3	4	4	5	5	6	6	8

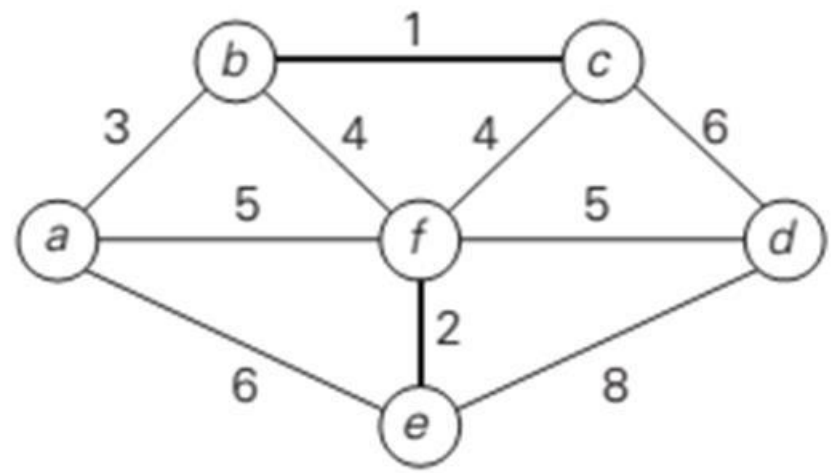
Kruskal's Algorithm

bc	ef	ab	bf	cf	af	df	ae	cd	de
1	2	3	4	4	5	5	6	6	8



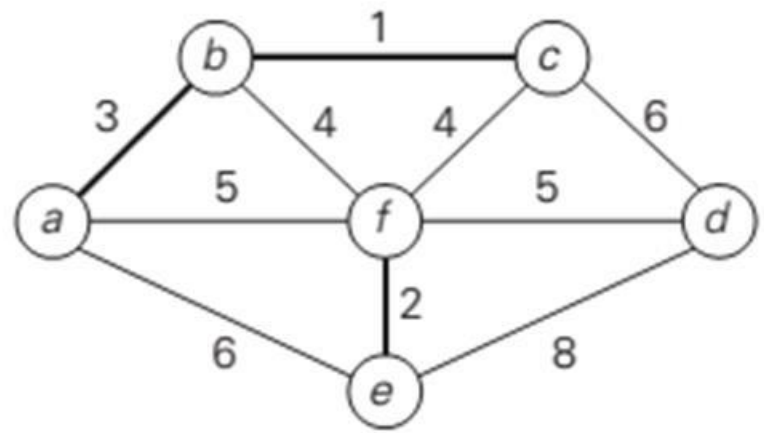
Kruskal's Algorithm

bc	bc	**ef**	ab	bf	cf	af	df	ae	cd	de
1	1	2	3	4	4	5	5	6	6	8



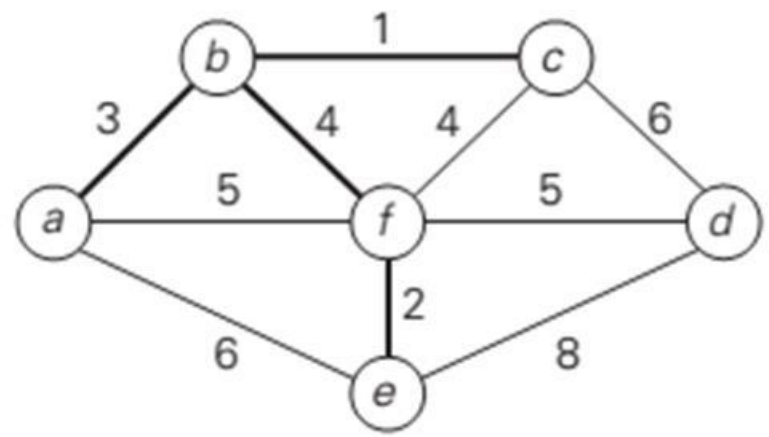
Kruskal's Algorithm

ef		bc	ef	**ab**	bf	cf	af	df	ae	cd	de
2		1	2	3	4	4	5	5	6	6	8



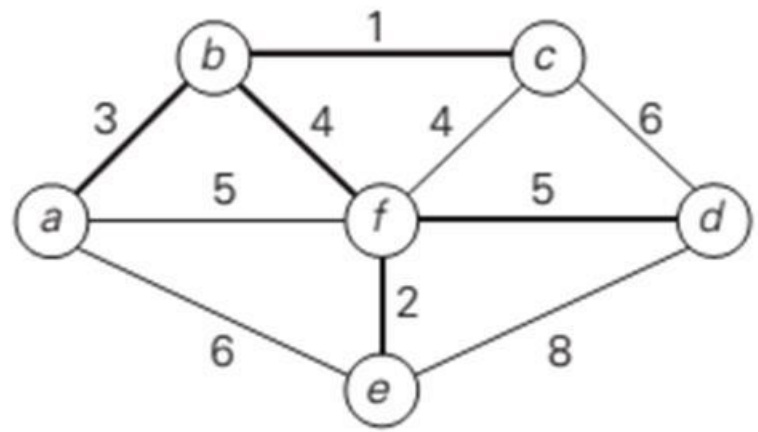
Kruskal's Algorithm

ab		bc	ef	ab	**bf**	cf	af	df	ae	cd	de
3		1	2	3	4	4	5	5	6	6	8



Kruskal's Algorithm

bf		bc	ef	ab	bf	cf	af	**df**	ae	cd	de
4		1	2	3	4	4	5	5	6	6	8



df
5

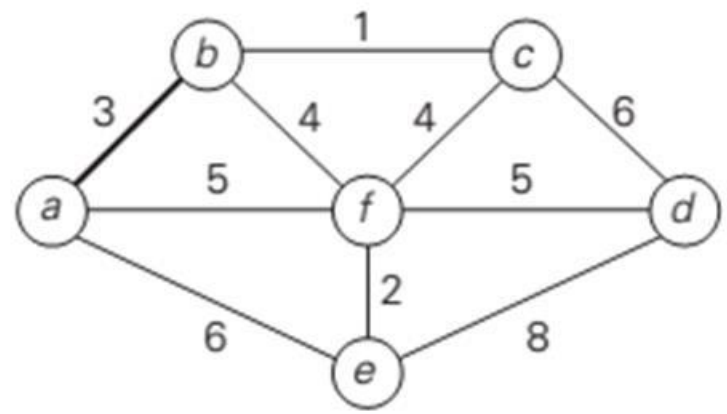
Kruskal's Algorithm

ALGORITHM *Kruskal(G)*

//Kruskal's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph $G = \langle V, E \rangle$
//Output: E_T, the set of edges composing a minimum spanning tree of G
sort E in nondecreasing order of the edge weights $w(e_{i_1}) \leq \cdots \leq w(e_{i_{|E|}})$
$E_T \leftarrow \varnothing$; $ecounter \leftarrow 0$ //initialize the set of tree edges and its size
$k \leftarrow 0$ //initialize the number of processed edges
while $ecounter < |V| - 1$ **do**
    $k \leftarrow k + 1$
    if $E_T \cup \{e_{i_k}\}$ is acyclic
        $E_T \leftarrow E_T \cup \{e_{i_k}\}$; $ecounter \leftarrow ecounter + 1$
return E_T

$O(|E| \log |E|)$

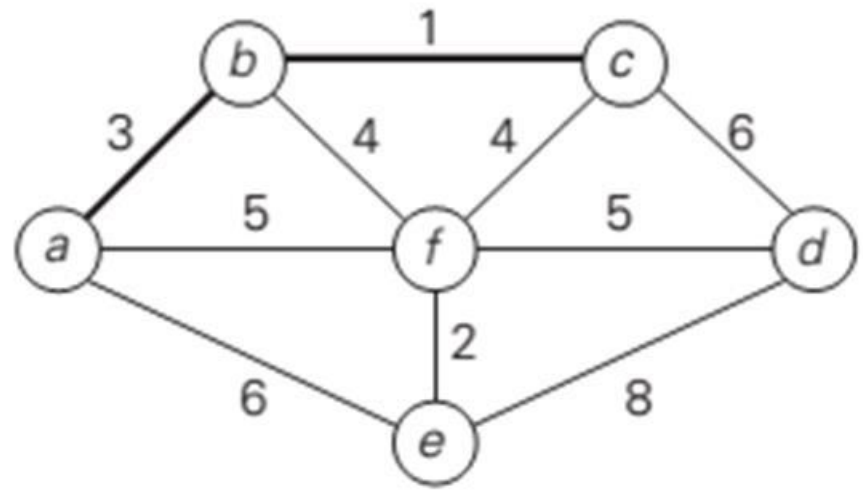
Prim's Algorithm

Tree vertices	Remaining vertices	Illustration
a(−, −)	**b(a, 3)** c(−, ∞) d(−, ∞) e(a, 6) f(a, 5)	

Prim's Algorithm

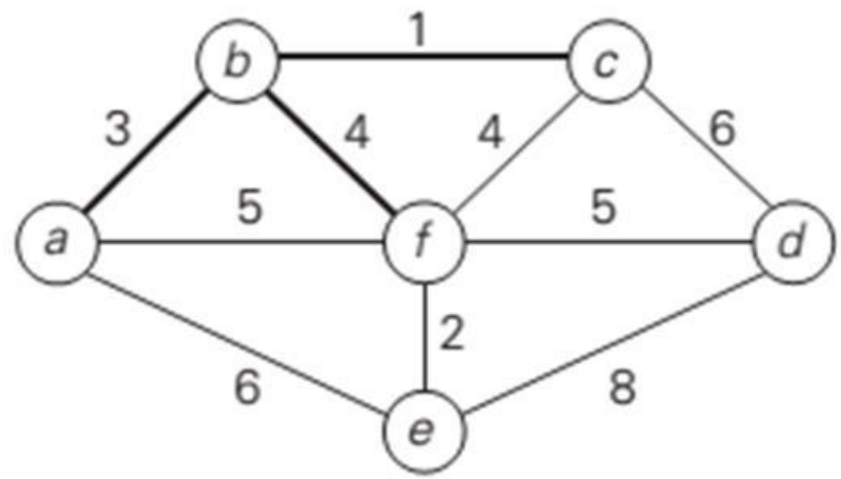
b(a, 3)

c(b, 1) d(−, ∞) e(a, 6)
f(b, 4)



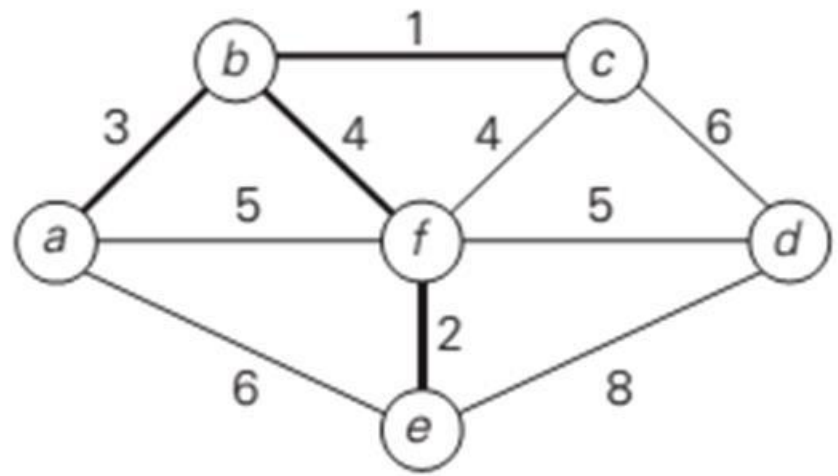
Prim's Algorithm

c(b, 1) d(c, 6) e(a, 6) **f(b, 4)**



Prim's Algorithm

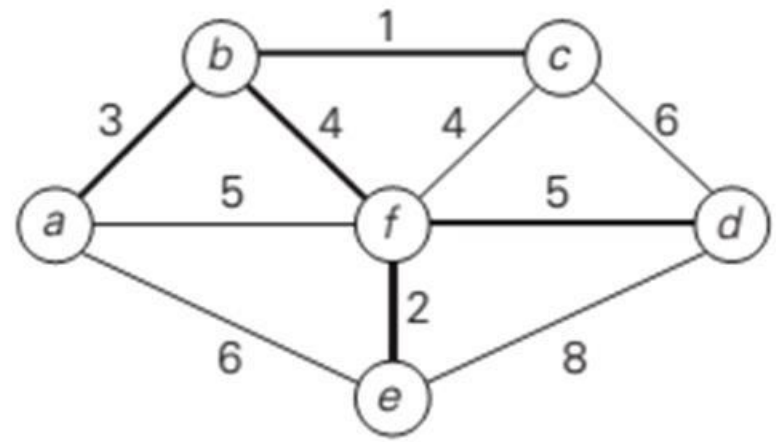
f(b, 4) d(f, 5) **e(f, 2)**



Prim's Algorithm

e(f, 2)

d(f, 5)



d(f, 5)

Prim's Algorithm

ALGORITHM *Prim(G)*

//Prim's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph $G = \langle V, E \rangle$
//Output: E_T, the set of edges composing a minimum spanning tree of G
$V_T \leftarrow \{v_0\}$ //the set of tree vertices can be initialized with any vertex
$E_T \leftarrow \varnothing$
for $i \leftarrow 1$ **to** $|V| - 1$ **do**
    find a minimum-weight edge $e^* = (v^*, u^*)$ among all the edges (v, u)
    such that v is in V_T and u is in $V - V_T$
    $V_T \leftarrow V_T \cup \{u^*\}$
    $E_T \leftarrow E_T \cup \{e^*\}$
return E_T

$O(|E| \log |V|)$

Prim's Algorithm

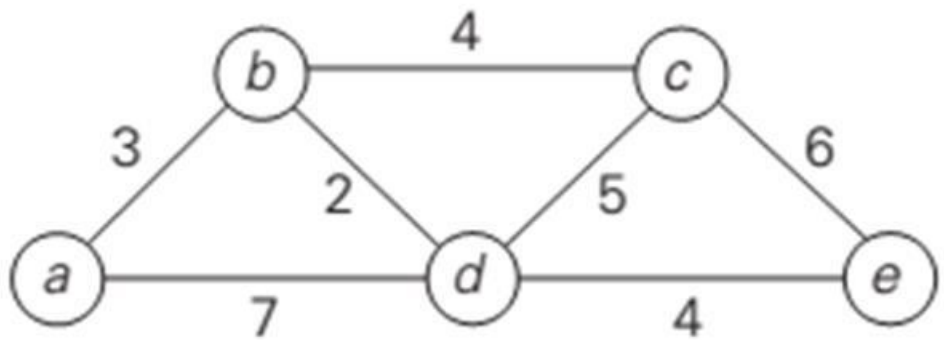
ALGORITHM *Prim(G)*

//Prim's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph $G = \langle V, E \rangle$
//Output: E_T, the set of edges composing a minimum spanning tree of G
$V_T \leftarrow \{v_0\}$ //the set of tree vertices can be initialized with any vertex
$E_T \leftarrow \varnothing$
for $i \leftarrow 1$ **to** $|V| - 1$ **do**
    find a minimum-weight edge $e^* = (v^*, u^*)$ among all the edges (v, u)
    such that v is in V_T and u is in $V - V_T$
    $V_T \leftarrow V_T \cup \{u^*\}$
    $E_T \leftarrow E_T \cup \{e^*\}$
return E_T

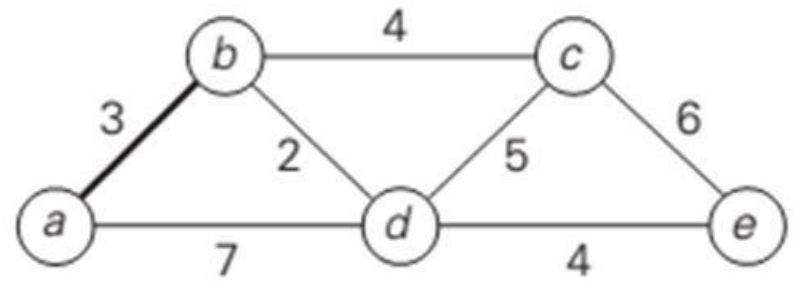
$O(|E| \log |V|)$

Single-source shortest-paths problem

- Single-source shortest-paths problem: for a given vertex called the source in a weighted connected graph, find shortest paths to all its other vertices.
- Dijkstra's algorithm

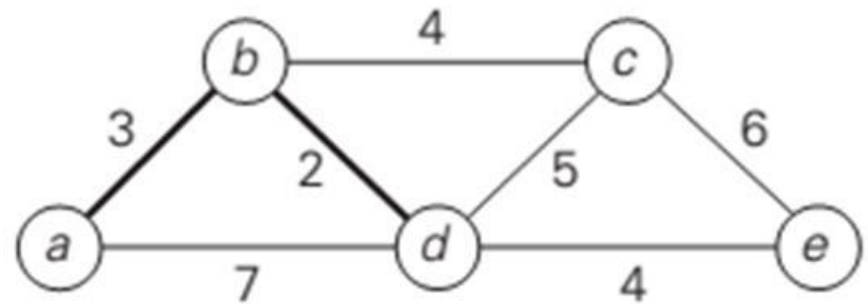


Dijkstra's algorithm

Tree vertices	Remaining vertices	Illustration
a(−, 0)	**b(a, 3)** c(−, ∞) d(a, 7) e(−, ∞)	

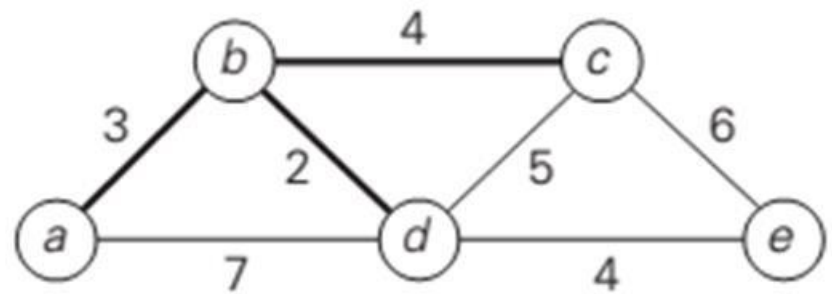
Dijkstra's algorithm

b(a, 3) c(b, $3+4$) **d(b, $3+2$)** e(−, ∞)



Dijkstra’s algorithm

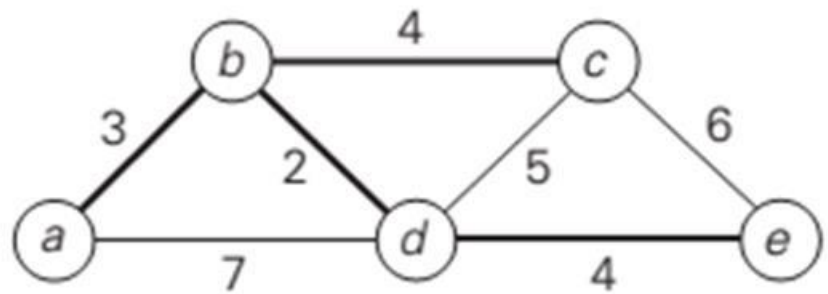
d(b, 5) **c(b, 7)** e(d, 5 + 4)



Dijkstra’s algorithm

c(b, 7)

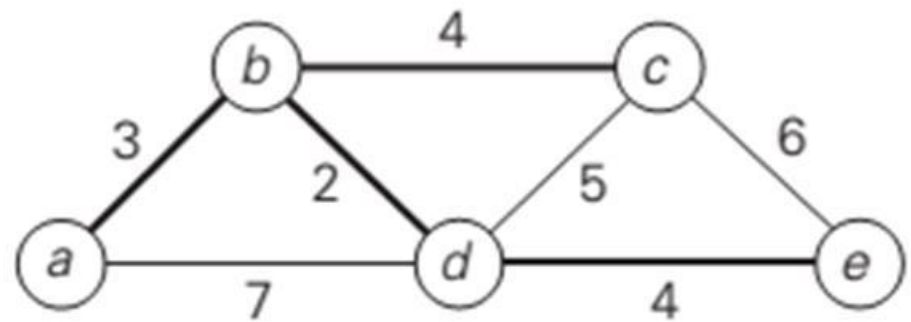
e(d, 9)



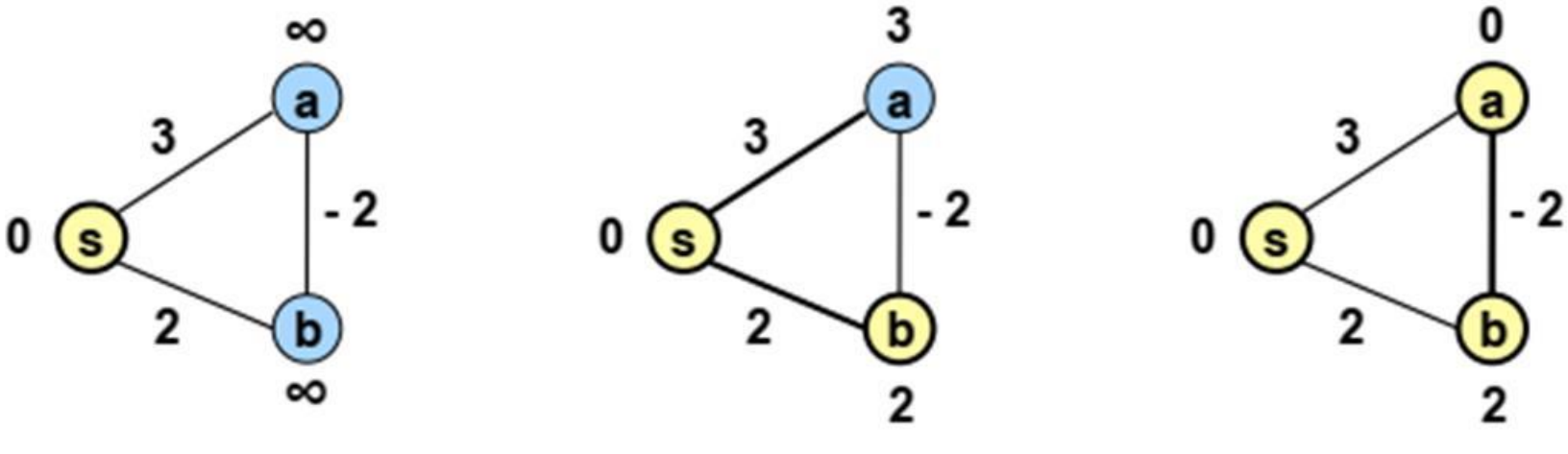
e(d, 9)

Dijkstra's algorithm

from a to b :	$a - b$	of length 3
from a to d :	$a - b - d$	of length 5
from a to c :	$a - b - c$	of length 7
from a to e :	$a - b - d - e$	of length 9



Example of *Dijkstra's property fails with negative edge weights*



Dijkstra's algorithm would visit b then a and leave b with a distance of 2 *instead of the correct distance* 1.
The problem is that when Dijkstra visits b, it fails to consider the possibility of there being a shorter path from a to b (which is impossible with nonnegative edge weights).

ALGORITHM *Dijkstra*(G, s)

```
//Dijkstra's algorithm for single-source shortest paths
//Input: A weighted connected graph G = ⟨V, E⟩ with nonnegative weights
//        and its vertex s
//Output: The length d_v of a shortest path from s to v
//         and its penultimate vertex p_v for every vertex v in V
Initialize(Q)  //initialize priority queue to empty
for every vertex v in V
    d_v ← ∞;   p_v ← null
    Insert(Q, v, d_v)  //initialize vertex priority in the priority queue
d_s ← 0;  Decrease(Q, s, d_s)  //update priority of s with d_s
V_T ← ∅
for i ← 0 to |V| − 1 do
    u* ← DeleteMin(Q)  //delete the minimum priority element
    V_T ← V_T ∪ {u*}
    for every vertex u in V − V_T that is adjacent to u* do
        if d_u* + w(u*, u) < d_u
            d_u ← d_u* + w(u*, u);   p_u ← u*
            Decrease(Q, u, d_u)
```

$O(|E| \log |V|)$

Knapsack problem

A thief breaks into a museum. Fabulous paintings, sculptures, and jewels are everywhere. The thief has a good eye for the value of these objects, and knows that each will fetch hundreds or thousands of dollars on the clandestine art collector's market. But, the thief has only brought a single knapsack to the scene of the robbery, and can take away only what he can carry. What items should the thief take to maximize the haul?

Fractional Knapsack Problem

- In a knapsack problem there is a knapsack or a container of capacity M, n items where, each item I is of weight w_i and is associated with a profit p_i.
- The problem of knapsack is to fill the available items into the knapsack so that the knapsack gets filled up and yields a maximum profit.
- If a fraction x_i of object i is placed into the knapsack, then a profit $p_i x_i$ is earned.
- The constrain is that all chosen objects should sum up to M

Illustration

- Consider a knapsack problem of finding the optimal solution where,

  M=15, (p1,p2,p3…p7) = (10, 5, 15, 7, 6, 18, 3) and (w1, w2, …., w7) = (2, 3, 5, 7, 1, 4, 1).
- In order to find the solution, one can follow three different strategies.

Strategy 1: non-increasing profit values

- Let (a,b,c,d,e,f,g) represent the items with profit (10,5,15,7,6,18,3) then the sequence of objects with non-increasing profit is (f,c,a,d,e,b,g), weights- (2, 3, 5, 7, 1, 4, 1).
- Profit= 47 units
- The solution set is

  (1, 0, 1, 4/7, 0, 1, 0).

Item chosen for inclusion	Quantity of item included	Remaining space in M	P_iX_i
f	1 full unit	15-4=11	18*1=18
C	1 full unit	11-5=6	15*1=15
A	1 full unit	6-2=4	10*1=10
d	4/7 unit	4-4=0	4/7*7=04

Strategy 2: non-decreasing weights

- Let (a,b,c,d,e,f,g) represent the items with profit (10,5,15,7,6,18,3) then the sequence of objects with non-increasing profit is (f,c,a,d,e,b,g), weights- (2, 3, 5, 7, 1, 4, 1).
- The sequence of objects with non-decreasing weights is (e,g,a,b,f,c,d).
- Profit= 54 units
- The solution set is (1,1,4/5,0,1,1,1).

Item chosen for inclusion	Quantity of item included	Remaining space in M	P_iX_I
E	1 full unit	15-1=14	6*1=6
G	1 full unit	14-1=13	3*1=3
A	1 full unit	13-2=11	10*1=10
b	1 full unit	11-3=8	5*1=05
f	1 full unit	8-4=4	18*1=18
c	4/5 unit	4-4=0	4/5*15=12

Strategy 3: non-decreasing weights with increasing profits

- Maximum profit per unit of capacity used
- This means that the objects are considered in decreasing order of the ratio P_i/w_i .
  - a: P1/w1 =10/2 = 5
  - b: P2/w2=5/3=1.66
  - c: P3/w3=15/5 = 3
  - d: P4/w4 =7/7=1
  - e: P5/w5 =6/1=6
  - f: P6/w6=18/4 = 4.5
  - g: P7/w7 =3/1=3
- Hence, the sequence is (e, a, f, c, g, b, d)

Strategy 3: non-decreasing weights with increasing profits

Item chosen for inclusion	Quantity of item included	Remaining space in M	P_iX_I
E	1 full unit	15-1=14	6*1=6
A	1 full unit	14-2=12	10*1=10
F	1 full unit	12-4=8	18*1=18
C	1 full unit	8-5=3	15*1=15
g	1 full unit	3-1=2	3*1=3
b	2/3 unit	2-2=0	2/3*5=3.33

Strategy 3: non-decreasing weights with increasing profits

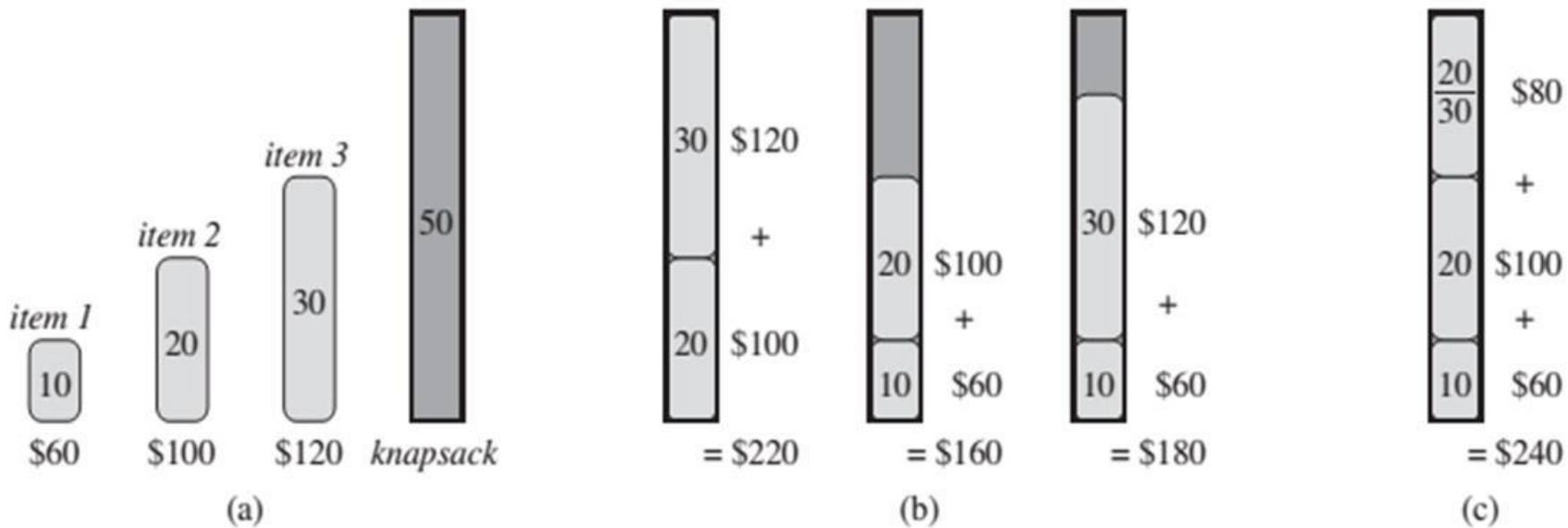
- Profit= 55.33 units
- The solution set is (1,2/3,1,0,1,1,1).
- In the above problem it can be observed that, if the sum of all the weights is <=M then all x_i= 1, is an optimal solution.
- If we assume that the sum of all weights exceeds M, all x_i's cannot be one.
- Sometimes it becomes necessary to take a fraction of some items to completely fill the knapsack.
- This type of knapsack problems is a general knapsack problem/Fractional knapsack problem.

Class exercise #1

- Given items as {value,weight} pairs {{60,20},{50,25},{20,5}}. The capacity of knapsack=40. The maximum value output assuming items to be divisible and nondivisible respectively

0-1 knapsack problem

- We call this the 0-1 knapsack problem because for each item, the thief must either take it or leave it behind; he cannot take a fractional amount of an item or take an item more than once



0-1 knapsack problem

- To see that this greedy strategy does not work for the 0-1 knapsack problem, consider the problem instance illustrated in Figure a .
- This example has 3 items and a knapsack that can hold 50pounds.
  - Item1weighs10pounds and is worth 60dollars.
  - Item2weighs20pounds and is worth100dollars.
  - Item3 weighs30pounds and is worth120dollars.
- Thus, the value per pound of item1is 6dollars per pound, which is greater than the value per pound of either item2(5 dollars per pound) or item3(4dollars per pound).
- The greedy strategy, therefore, would take item1first.
- As you can see from the case analysis in Figure b, however, the optimal solution takes items2and3, leaving item 1behind.
- The two possible solutions that take item1 are both suboptimal.

0-1 knapsack problem

- 0-1 knapsack problem cannot results optimal solution by using Greedy.
- Hallmark for Dynamic Programming
- Optimal solution is achieved by Dynamic Programming.
- The Complexity of Fractional Knapsack is O(nlogn).

Optimal Storage on Tapes: Introduction

- Consider n programs that are to be stored on a tape of length L.
- Each program I is of length L_i where i lies between 1 and n.
- All programs can be stored on the tape if the sum of the lengths of the programs is at most L.
- It is assumed that, whenever a program is to be retrieved the tape is initially positioned at the start/end.
- There are n programs that are to be stored on a computer tape of length L. Associated with each program i is a length L_i.
- Assume the tape is initially positioned at the front. If the programs are stored in the order $L = i_1, i_2, \ldots, i_n$, the time t_j needed to retrieve program i_j

$$t_j = \sum_{k=1}^{j} L_{i_k}$$

Optimal Storage on Tapes

- If all programs are retrieved equally often, then the mean retrieval time

$$(\text{MRT}) = \frac{1}{n}\sum_{j=1}^{n} t_j$$

- This problem fits the ordering paradigm. Minimizing the MRT is equivalent to minimizing

$$d(L) = \sum_{j=1}^{n}\sum_{k=1}^{j} L_{i_k}$$

Assume that 3 sorted files are given. Let the length of files A, B and C be 7, 3 and 5 units respectively. All these three files are to be stored on to a tape S in some sequence that reduces the average retrieval time. The table shows the retrieval time for all possible orders.

Order of recording	Retrieval time	MRT
ABC	7+(7+3)+(7+3+5)=32	32/3
ACB	7+(7+5)+(7+5+3)=34	34/3
BAC	3+(3+7)+(3+7+5)=28	28/3
BCA	3+(3+5)+(3+5+7)=26	26/3
CAB	5+(5+7)+(5+7+3)=32	32/3
CBA	5+(5+3)+(5+3+7)=28	28/3

Example

- Let n = 3, $(L_1, L_2, L_3) = (5,10,3)$. 6 possible orderings. The optimal is 3,1,2

Ordering I	d(I)
1,2,3	5+5+10+5+10+3 = 38
1,3,2	5+5+3+5+3+10 = 31
2,1,3	10+10+5+10+5+3 = 43
2,3,1	10+10+3+10+3+5 = 41
3,1,2	3+3+5+3+5+10 = 29
3,2,1,	3+3+10+3+10+5 = 34

Optimal Storage on Tapes

- From the above discussion one can observe that the MRT can be minimized if the programs are stored in an increasing order i.e., l1 <= l2 <= l3, … ln.
- Hence the ordering defined minimizes the retrieval time.
- The solution set obtained need not be a subset of data but may be the data set itself in a different sequence.

Pseudo code : Optimal Storage on Tapes

```
tapes(n, m) // here n is the numbers of programs and m is the numbers of tapes
{
j=0;
Sort (n);
for(i=1 to n do)
        {
        write ("append", i "to the permutation of the tape", j)
        j=(j+1) mod m
        }
}
```

Container Loading problem

- A large ship is to be loaded with containers of cargos. Different containers, although of equal size, will have different weights.
- Let w_i be the weight of the i^{th} container, $1 \leq i \leq n$, and the capacity of the ship is c.
- Find out how could we load the ship with the maximum number of containers.
- Let $x_i \in \{0, 1\}$.
  - If $x_i = 1$, we will load the i^{th} container, otherwise, we will not load it.
  - We wish to assign values to $x'_i s$ such that $\sum_{i=1}^{n} x_i \leq$ c, and $x_i \in \{0, 1\}$.
  - The optimization function is $\sum_{i=1}^{n} x_i$.

Example

- Given that ,
  - n = 8,
  - i = [1,2,3,4,5,6,7,8],
  - w_i = [100,200, 50, 90, 150, 50, 20, 80]
  - C= 400
- What is the order of loading ?
  - $[x_1 \ldots x_8]$=[]
- $[x_1 \ldots . x_8]$ = [1,0,1,1,0,1,1,1]

The Container Loading Problem

- Let x_i be a boolean variable (1=container loaded , 0=container not loaded).
- The problem is to assign values x_i such that with
  - W_i - the weight of the container i and
  - C - the maximum cargo carrying capacity of a ship
- $\sum_{i=1}^{n} x_i.$ is maximized with respect to the constraints that

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

Algorithm Container Loading

```
1.  for i=1 to n
2.        do x[i] = 0;
3.  t <- Allocate the memory (n);
4.  Indirect-Sort(w, t, n);
5.  while (i <=n) and w[t[i]] ≤ C)
6.  do
7.        x[t[i]] = 1;
8.        C = C – w[t[i]];
9.        i=i+1;
10. delete t
```