Chapter – 3 – Greedy Algorithm

- General Characteristics of greedy algorithms
- Activity selection problem
- Elements of Greedy Strategy
- Minimum Spanning Trees
- Kruskal’s algorithm
- Prim’s algorithm
- Shortest paths – Dijkstra’s Algorithm
- The Knapsack problem

General Characteristics of greedy algorithms

- Most straightforward design technique
  - Most problems have n inputs
  - Solution contains a subset of inputs that satisfies a given constraint
  - Feasible solution: Any subset that satisfies the constraint
  - Need to find a feasible solution that maximizes or minimizes a given objective function – optimal solution

General Characteristics of greedy algorithms

- Used to determine a feasible solution that may or may not be optimal
  - At every point, make a decision that is locally optimal; and hope that it leads to a globally optimal solution
  - Leads to a powerful method for getting a solution that works well for a wide range of applications.
  - May not guarantee the best solution

Greedy v/s. Dynamic Programming

- *Greedy algorithms focus on making the best local choice at* each decision point. In the absence of a correctness proof such greedy algorithms are very likely to fail.
- Dynamic programming gives us a way to design custom algorithms which systematically search all possibilities (thus guaranteeing correctness) while storing results to avoid recomputing (thus providing efficiency)

Greedy Algorithm Vs Dynamic Programming

Comparison:

Dynamic Programming	Greedy Algorithms
• At each step, the choice is determined based on solutions of subproblems.	• At each step, we quickly make a choice that currently looks best. --A local optimal (greedy) choice.
• Sub-problems are solved first.	• Greedy choice can be made first before solving further sub-problems.
• Bottom-up approach	• Top-down approach
• Can be slower, more complex	• Usually faster, simpler

General Characteristics of greedy algorithms

- Characteristics of greedy algorithms
  - **1) We have some problem to solve in an optimal way. To construct the solution of our problem, we have a set of candidates:** the coins that are available, the edges of a graph that may be used to build a path, the set of jobs to be scheduled, or whatever.
  - **2) As the algorithm proceeds, we accumulate two other sets.** One contains candidates that have already been considered and chosen, while the other contains candidates that have been considered and rejected.

General Characteristics of greedy algorithms

- Characteristics of greedy algorithms
  - **3) There is a function that checks whether a particular set of candidates provides a solution to our problem, ignoring questions of optimality for the time being.** For instance, do the coins we have chosen add up to the amount to be paid? Do the selected edges provide a path to the node we wish to reach ? Have all the jobs been scheduled ?
  - **4) A second function checks whether a set of candidates is feasible, that is, whether or not it is possible to complete the set by adding further candidates so as to obtain at least one solution to our problem.** We usually expect the problem to have at least one solution that can be obtained using candidates from the set initially available.

General Characteristics of greedy algorithms

- Characteristics of greedy algorithms
    - **5) Yet another function, the selection function, indicates at any time which of the remaining candidates, that have neither been chosen nor rejected, is the most promising.**
    - **6) Finally an objective function gives the value of a solution we have found:** the number of coins we used to make change, the length of the path we constructed, the time needed to process all the jobs in the schedule, or whatever other value we are trying to optimize.

General Characteristics of greedy algorithms

- Function greedy(C: set) : set

```
{
    { C is the set of candidates}

    S □ ∅ { we construct the solution in the set S}

      while C ≠ ∅ and not solution (S) do
                x □ select(C)
                C □ C | {x}
                if feasible (S ∪ {x}) then S □ S ∪ {x}

    if solution(S) then return S
                else return “there are no solutions”
  }
```

General Characteristics of greedy algorithms

- Example – Suppose we live in country where the following coins are available: 1 dollar(100 cent) , 25 cent, 10 cent(1 dime), 5 cent and 1 cent.

- Our problem is to devise an algorithm for paying a given amount to a customer using the smallest possible number of coins.

- For instance, if we must pay $ 2.89(289 cents), the best solution is to give the customer 10 coins: 2 dollars, 3 quarters, 1 dime and 4 – 1 cent coins.

General Characteristics of greedy algorithms

- Function {make - change}(n) : set of coins

```
{
    { Makes change of n units using the least possible number of coins. The constant C specifies
    the coinage}

    const C □ { 100, 25, 10, 5, 1}
    S □ ∅ { S is a set that will hold the solution}
    s □ 0 { s is the sum of the items in S}

    While s ≠ n do
            x □ the largest item in C such that s + x <= n

            if “there is no such item” then
                    return “no solution found”

            S □ S ∪ { a coin of value x}
            s □ s + x
  }
```

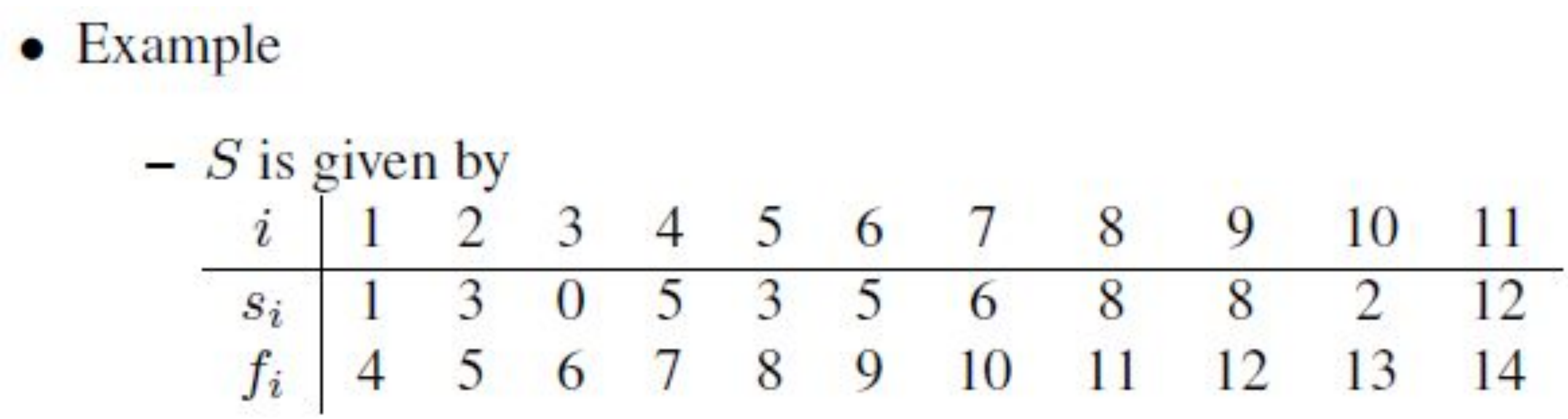
Activity selection problem

- Several competing activities require exclusive use of a common resource.

- Goal is to select a set of maximum - size set of mutually compatible activities.

  - Set S of n proposed activities, requiring exclusive use of a resource, such as a lecture hall S = {a1, a2, . . . , an}
  - Each activity Ai has a start time Si and a finish time Fi, such that $0 <= Si < Fi < \infty$
  - Activity Ai takes place in the interval [Si, Fi], if selected.
  - Activities Ai and Aj are compatible if intervals [Si, Fi] and [Sj , Fj] do not overlap
  - Activity selection problem is to select a maximum-size subset of mutually compatible activities

Activity selection problem



- Example
  - S is given by

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

Activity selection problem

- Example
  - S is given by

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

Activity selection problem

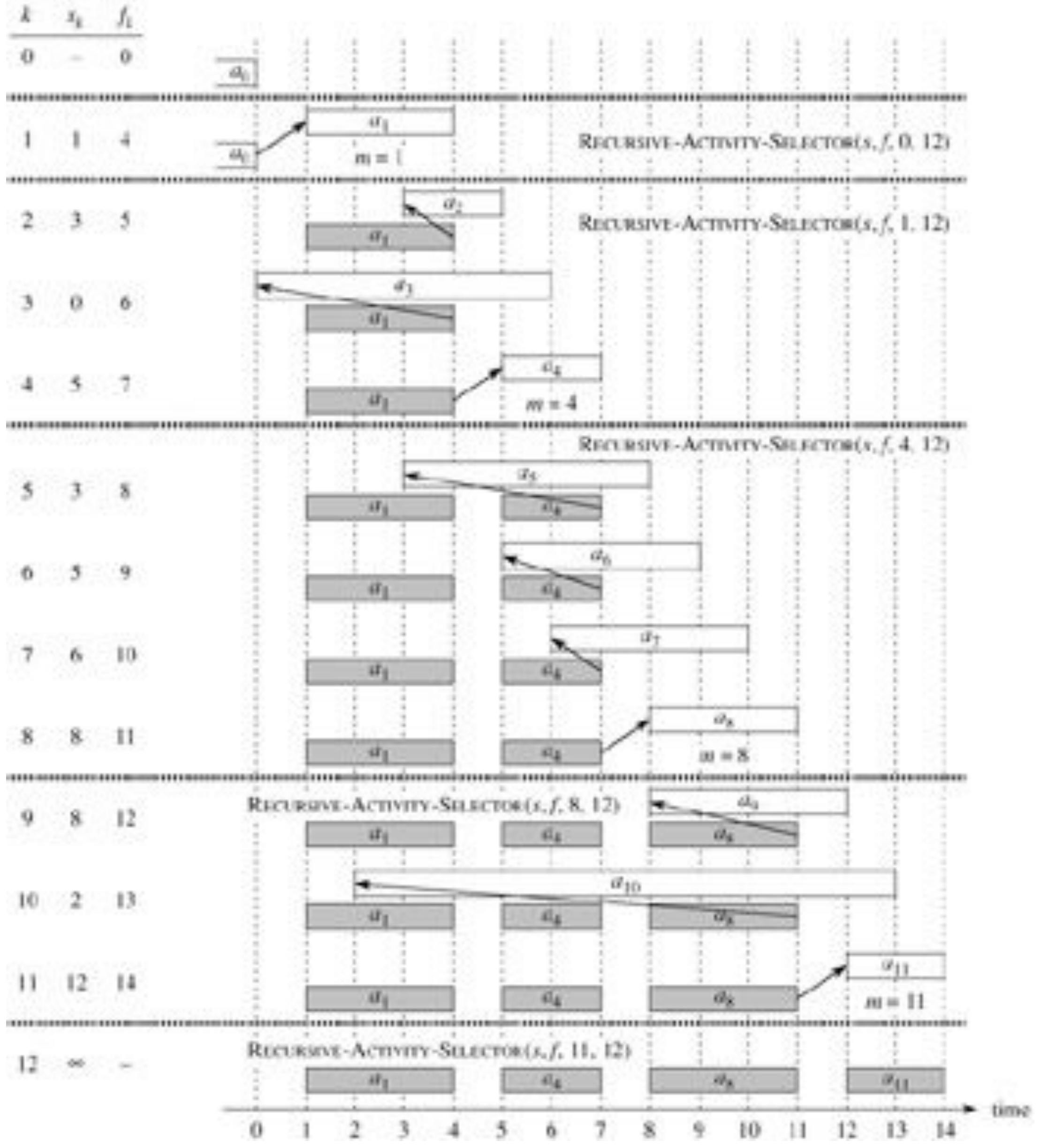
- RECURSIVE-ACTIVITY-SELECTOR
  - It takes the start and finish times of the activities, represented as arrays **s** and **f**, as well as the starting indices i and j of the sub problem Sij it is to solve.
  - It returns a maximum-size set of mutually compatible activities in *Si.j.*
  - We assume that the n input activities are ordered by monotonically increasing finish time.
  - If not, we can sort them into this order in O(n lg n) time, breaking ties arbitrarily.
  - The initial call is RECURSIVE-ACTIVITY-SELECTOR(s, f, 0, n + 1).

Activity selection problem

- RECURSIVE-ACTIVITY-SELECTOR(*s, f, i, j)*

```
{
     m ← i + 1
    while m < j and Sm < fi     ▹ Find the first activity in Sij.
          do m ← m + 1
    if m < j
          then return {Am}  ∪ RECURSIVE-ACTIVITY-
    SELECTOR(s, f, m, j)
    else return Ø
}
```

Activity selection problem


RECURSIVE-ACTIVITY-SELECTOR(s, f, 0, 12)
m = 1
RECURSIVE-ACTIVITY-SELECTOR(s, f, 1, 12)
m = 4
RECURSIVE-ACTIVITY-SELECTOR(s, f, 4, 12)
m = 8
RECURSIVE-ACTIVITY-SELECTOR(s, f, 8, 12)
m = 11
RECURSIVE-ACTIVITY-SELECTOR(s, f, 11, 12)
time

k	s_k	f_k
0	–	0
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14
12	∞	–

Activity selection problem

- GREEDY-ACTIVITY-SELECTOR(S*, F)*

```
{
        n ← length[S]      //total no. of activities
        A ← {A1}           //select 1st activity
        i ← 1              //current activity is $A_i$
        for m ← 2 to n
            do if Sm ≥ Fi
                then A ← A ∪ {Am}
                    i ← m
        return A
}
```

Elements of Greedy Strategy

- A greedy algorithm obtains an optimal solution to a problem by making a sequence of choices. For each decision point in the algorithm, the choice that seems best at the moment is chosen.

- This heuristic strategy does not always produce an optimal solution, but as we saw in the activity-selection problem, sometimes it does.

- This section discusses some of the general properties of greedy methods.

Elements of Greedy Strategy

- **1**. Determine the optimal substructure of the problem.
- **2.** Develop a recursive solution.
- **3.** Prove that at any stage of the recursion, one of the optimal choices is the greedy choice. Thus, it is always safe to make the greedy choice.
- **4.** Show that all but one of the sub problems induced by having made the greedy choice are empty.
- **5**. Develop a recursive algorithm that implements the greedy strategy.
- **6**. Convert the recursive algorithm to an iterative algorithm.

Elements of Greedy Strategy

- More generally, we design greedy algorithms according to the following sequence of steps:
  - **1.** Cast the optimization problem as one in which we make a choice and are left with one sub problem to solve.
  - **2.** Prove that there is always an optimal solution to the original problem that makes the greedy choice, so that the greedy choice is always safe.
  - **3.** Demonstrate that, having made the greedy choice, what remains is a sub problem with the property that if we combine an optimal solution to the sub problem with the greedy choice we have made, we arrive at an optimal solution to the original problem.

Elements of Greedy Strategy

- How can one tell if a greedy algorithm will solve a particular optimization problem?
- There is no way in general, but the **greedy-choice property and optimal sub-structure** are the two key ingredients.
- If we can demonstrate that the problem has these properties, then we are well on the way to developing a greedy algorithm for it.

Elements of Greedy Strategy

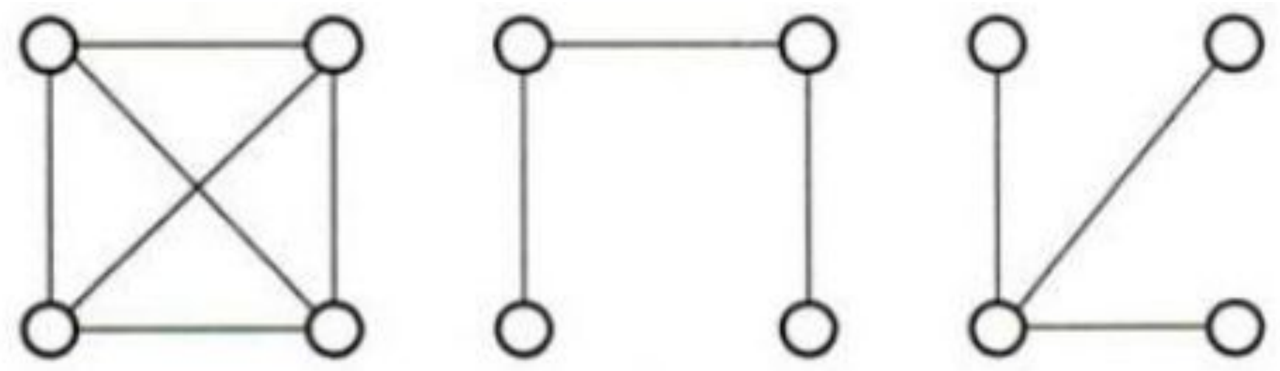
- **Greedy-choice property**
  - The first key ingredient is the greedy-choice property: a globally optimal solution can be arrived at by making a locally optimal (greedy) choice.
  - In other words, when we are considering which choice to make, we make the choice that looks best in the current problem, without considering results from sub problems.

Elements of Greedy Strategy

- **Optimal substructure**
  - A problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solutions to sub problems.
  - This property is a key ingredient of assessing the applicability of dynamic programming as well as greedy algorithms.
  - If an optimal solution to sub problem S_{ij} *includes an activity* a_k*, then it must also contain optimal solutions to the* sub problems S_{ik} *and* S_{kj}*.*
  - Given this optimal substructure, we argued that if we knew which activity to use as a_k, we could construct an optimal solution to S_{ij} by selecting a_k along with all activities in optimal solutions to the sub problems S_{ik} and S_{kj}.
  - Based on this observation of optimal substructure, we were able to devise the recurrence that described the value of an optimal solution.

Minimum Spanning Trees

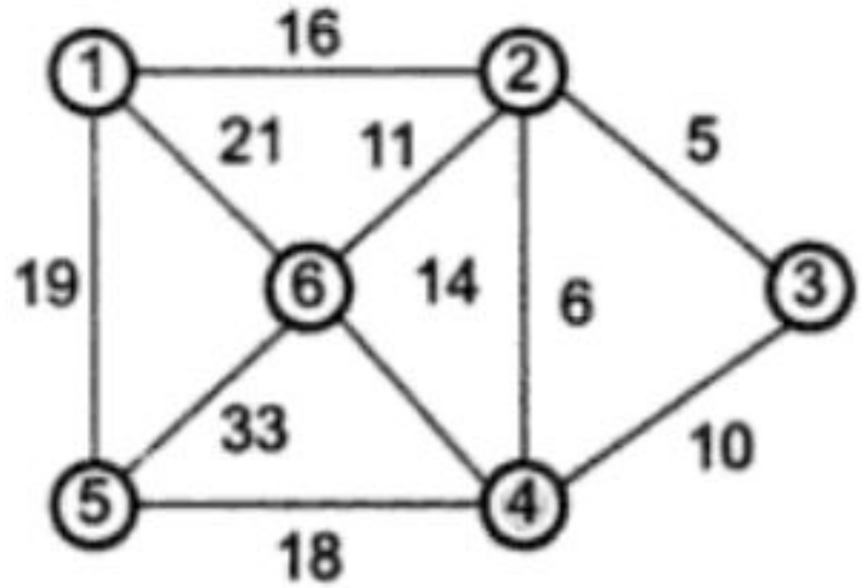
- Let G = <V, E> be an undirected, connected graph, having vertices V and edges E.
- A sub – graph T = <V’, E’> of G is a spanning tree of G if T is a tree.
- An undirected graph with |V| = n, may have as many as n(n-1)/2 edges, but a spanning tree T will have exactly (n-1) edges.


Undirected Graph some of its Spanning Trees

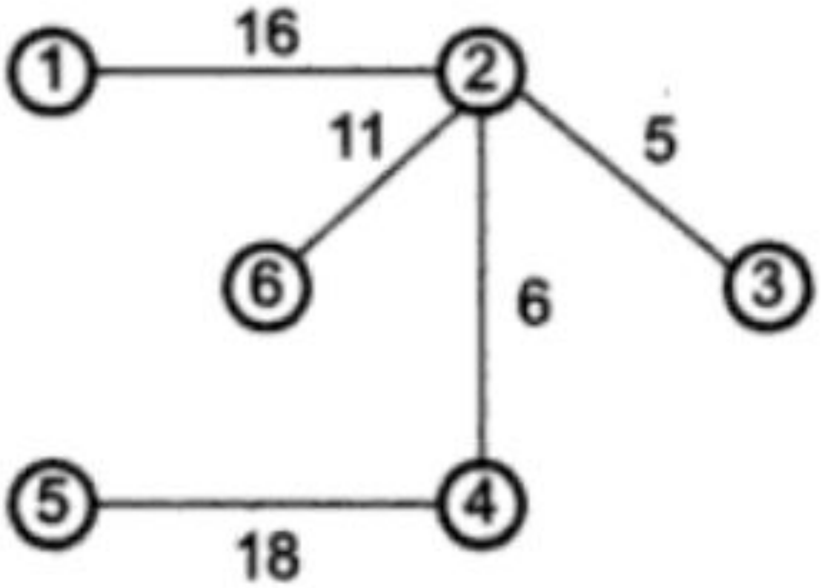
Minimum Spanning Trees

- In a practical situation, the edges will have a weight or cost associated with it.
- For example, the weight of an edge in the case of the roads between cities is the length of the road.
- Then suppose we are planning roads between cities, we would be interested in the roads with a minimum total length.
- That would give us a minimum spanning tree for the problem graph.

Minimum Spanning Trees



A Graph



one of its Minimum Spanning Trees

Minimum Spanning Trees

- A greedy method to obtain the minimum spanning tree would construct the tree edge by edge, where each next edge is chosen according to some optimization criterion.

- An obvious criterion would be to choose an edge which adds a minimum weight to the total weight of the edges so far selected.

- There are two ways in which this criterion can be achieve.
    - 1) Kruskal’s algorithm
    - 2) Prim’s algorithm

Kruskal’s algorithm

- The set T of edges is initially empty. As the algorithm progress, edges are added to T. So long as it has not found a solution, the partial graph formed by the nodes of G and the edges in T consists of several connected components.

- The elements of T included in a given connected component form a minimum spanning tree for the nodes in this component.

- At the end of the algorithm only one connected component remains, so T is then a minimum spanning tree for all the nodes of G.

Kruskal’s algorithm

- To build bigger and bigger connected components, we examine the edges of G in order of increasing length.
- If an edge joins two nodes in different connected components, we add it to T. Consequently, the two connected components now form only one component.
- Otherwise the edge is rejected: it joins two nodes in the same connected component, and therefore can not be added to T without forming a cycle.
- The algorithm stops when only one connected component remains.

Kruskal’s algorithm

- Function Kruskal(G = <N, A>: graph;): set of edges

```
{   {initialization}
    Sort A by increasing length
    n  the number of nodes in N
    T  ∅ {will contain the edges of minimum                    spanning tree.}
    repeat
        e  {u, v}  shortest edge not yet considered
        ucomp  find (u)
        vcomp  find (v)
        if ucomp ≠ vcomp then
            merge(ucomp, vcomp)
        T  T ∪ {e}
    until T contains n-1 edges

    return T
}
```

Prim’s algorithm

- In Kruskal’s algorithm the selection function chooses edges in increasing order of length without worrying too much about their connection to previously chosen edges except that we are careful never to form a cycle.
- In Prim’s algorithm, on the other hand, the minimum spanning tree grows in a natural way, starting from an arbitrary root.
- At each stage we add a new branch to the tree already constructed the algorithm stops when all the nodes have been reached.

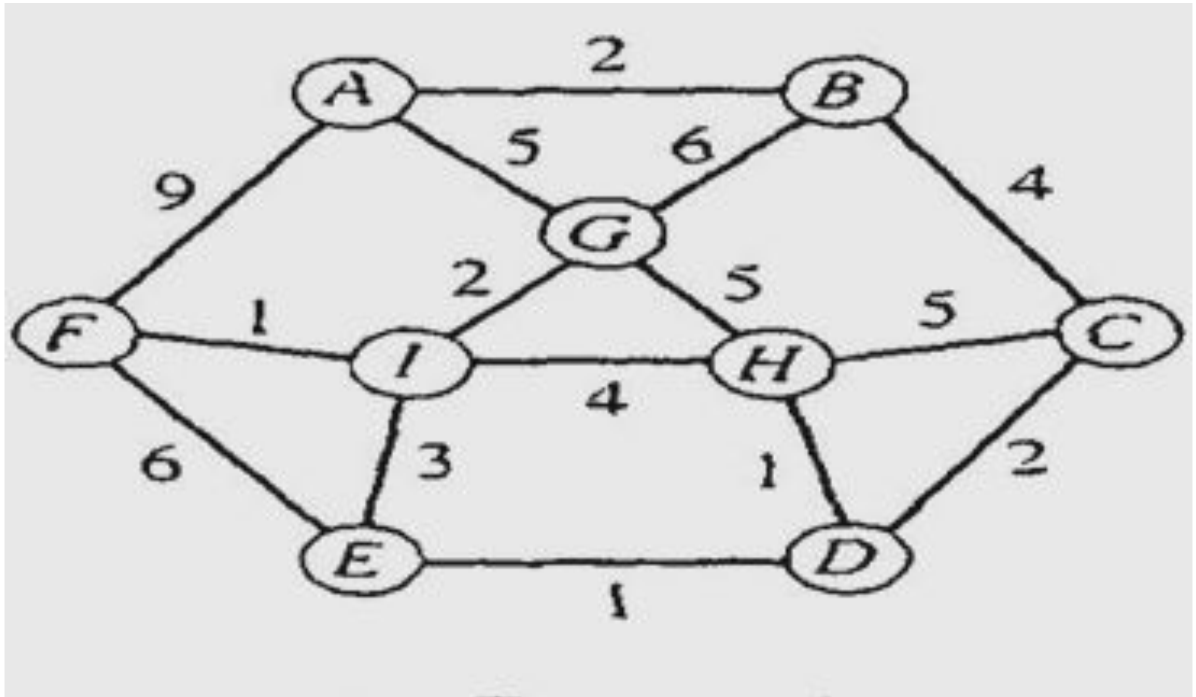
Prim’s algorithm

- Function Prim(G = <N, A> graph): set of edges

```
{
    {initialization}
    T □ ∅
    B □ { an arbitrary member of N}
    while B ≠ N do
        find e = {u, v} of minimum length such that
            u ∈ B and v ∈ N/B

        T □ T ∪ {e}
        B □ B ∪ {v}
    return T
}
```

Prim’s algorithm

- Example



(a) The graph

Shortest paths – Dijkstra’s Algorithm

- Consider now a directed graph G = <N, A> where N is the set of nodes of G and A is the set of directed edges. Each edge has a nonnegative length.

- One of the nodes is designated as the source node. The problem is to determine the length of the shortest path from the source to each of the other nodes of the graph.

- The cost of an edge instead of its length, and pose the problem of finding the cheapest path from the source to each other node.

Shortest paths – Dijkstra's Algorithm

- For simplicity, we again assume that the nodes of G are numbered from 1 to n, so N = {1, 2, ….. n}.
- We can suppose without loss of generality that node 1 is the source.
- Suppose also that matrix L gives the length of each directed edge: L[i, j] >= 0 if the edge (i, j) ∈ A, and L[i, j] = ∞ otherwise.
- Here is the algorithm.

Shortest paths – Dijkstra’s Algorithm

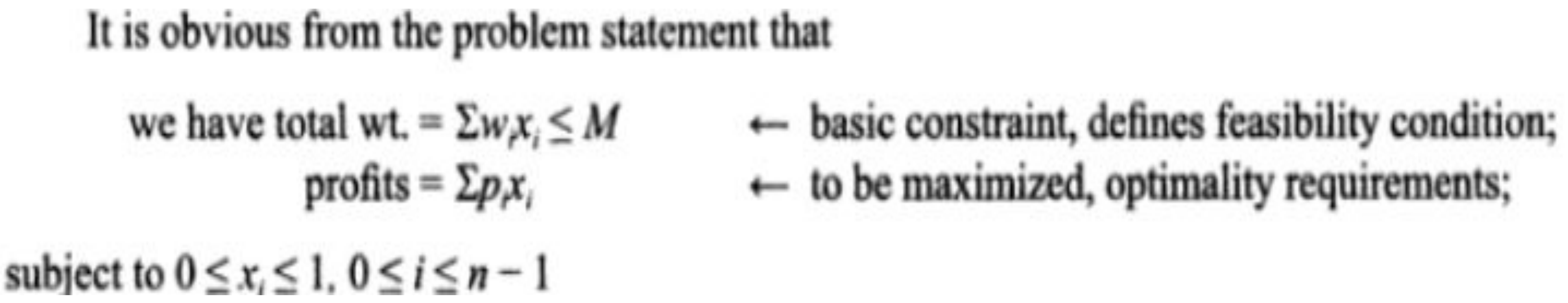
- Function Dijkstra(L[1…n, 1…n]) : array [2…n]

```
{
    array D[2…n]
    {initialization}

    C  {2, 3, … n} {S = N – C exists only implicitly}
    for i  2 to n do D[i]  L[1, i]

    {greedy loop}
    repeat n – 2 times

        v  some element of C minimizing D[v]
        C  C – {v} {and implicitly S  S ∪ {v}}
        for each w ∈ C do
            D[w]  min (D[w], D[v]+L[v, w])
    return D
}
```

The Knapsack problem

- This problem arises in various forms. We are given n objects and a knapsack.
- For i = 1, 2, … n, object I has a positive weight w_i and a positive value V_i. The knapsack can carry a weight not exceeding W.
- Our aim is to fill the knapsack in a way that maximizes the value of the included objects, while respecting the capacity constraint.
- We assume that the objects can be broken into smaller pieces, so we may decide to carry only a fraction x_i of object i, where $0 <= x_i <=1$.
- In this case, object I contributes xiwi to the total weight in the knapsack, and xivi to the value of the load.
- In symbols, the problem can be stated as follows:

The Knapsack problem



It is obvious from the problem statement that

we have total wt. = $\Sigma w_i x_i \leq M$ ← basic constraint, defines feasibility condition;

profits = $\Sigma p_i x_i$ ← to be maximized, optimality requirements;

subject to $0 \leq x_i \leq 1$, $0 \leq i \leq n - 1$

The Knapsack problem

- We should use a greedy algorithm to solve the problem. In terms of our general schema, the candidates are the different objects, and a solution is a vector (x1, …, xn) telling us what fraction of each object to include.

- A feasible solution is the total value of the objects constraints given above, and the objective function is the total value of the objects in the knapsack.

- It is also clear that an optimal solution must fill the knapsack exactly, for otherwise we could add a fraction of one of the remaining objects and increase the value of the load.

The Knapsack problem

- Function knapsack(w[1…n], v[1…n], W) array [1…n]

```
{
    {initialization}

    for I = 1 to n do x[i]  0
    weight  0
    {greedy loop}
    while weight < W do
        i  the best remaining object (will see v/w)
        if weight + w[i] <= W then x[i]  1
                                   weight  weight + w[i]
                              else
                                   x[i]  (W - weight)/w[i]
                                   weight  W

    return x
}
```

The Knapsack problem

- There are at least three possible selection functions for this problem:
  - At each stage we might choose the **most valuable remaining object**, arguing that this increases the value of the load as quickly as possible.
  - We might choose the **lightest remaining object**, on the around that this uses up capacity as slowly as possible.
  - If we select the objects in order of decreasing v_i/w_i and it lead to an optimal solution.

The Knapsack problem

We can assume that p and w are positive numbers.
To understand the problem better, consider a numerical example for this problem:

$n = 3$, $M = 20$, $P = \{25, 24, 15\}$, $W = \{18, 15, 10\}$.

Some of the feasible solutions are shown in Table 10.1

Solu. no.	x_1	x_2	x_3	$\Sigma w_i x_i$	$\Sigma p_i x_i$
1	0.5	0.333	0.25	16.5	24.5
2	1	2/15	0	20	28.2
3	0	0.667	1	20	31.0
4	0	1	0.5	20	31.5 (optimal)

The Knapsack problem

- This example shows that the solution obtained by a greedy algorithm that maximizes the value of each object it selects is not necessarily optimal, nor is the solution obtained by minimizing the weight of the each object that is chosen.

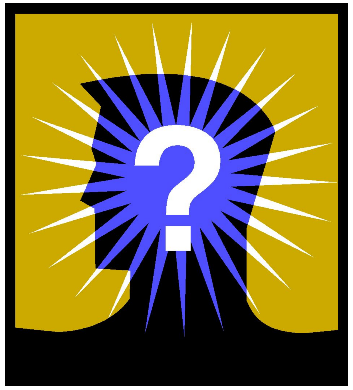
- Fortunately the following proof shows that the third possibility, selecting the object that maximizes the value per unit weight, does lead to an optimal solution.

The Knapsack problem

N = 5	W = 100

W	10	20	30	40	50
V	20	30	66	40	60
V/W	2.0	1.5	2.2	1.0	1.2

Question ?



How Projects Really Work (version 1.5)

How the customer explained it

How the project leader understood it

How the analyst designed it

How the programmer wrote it

What the beta testers received

How the business consultant described it

How the project was documented

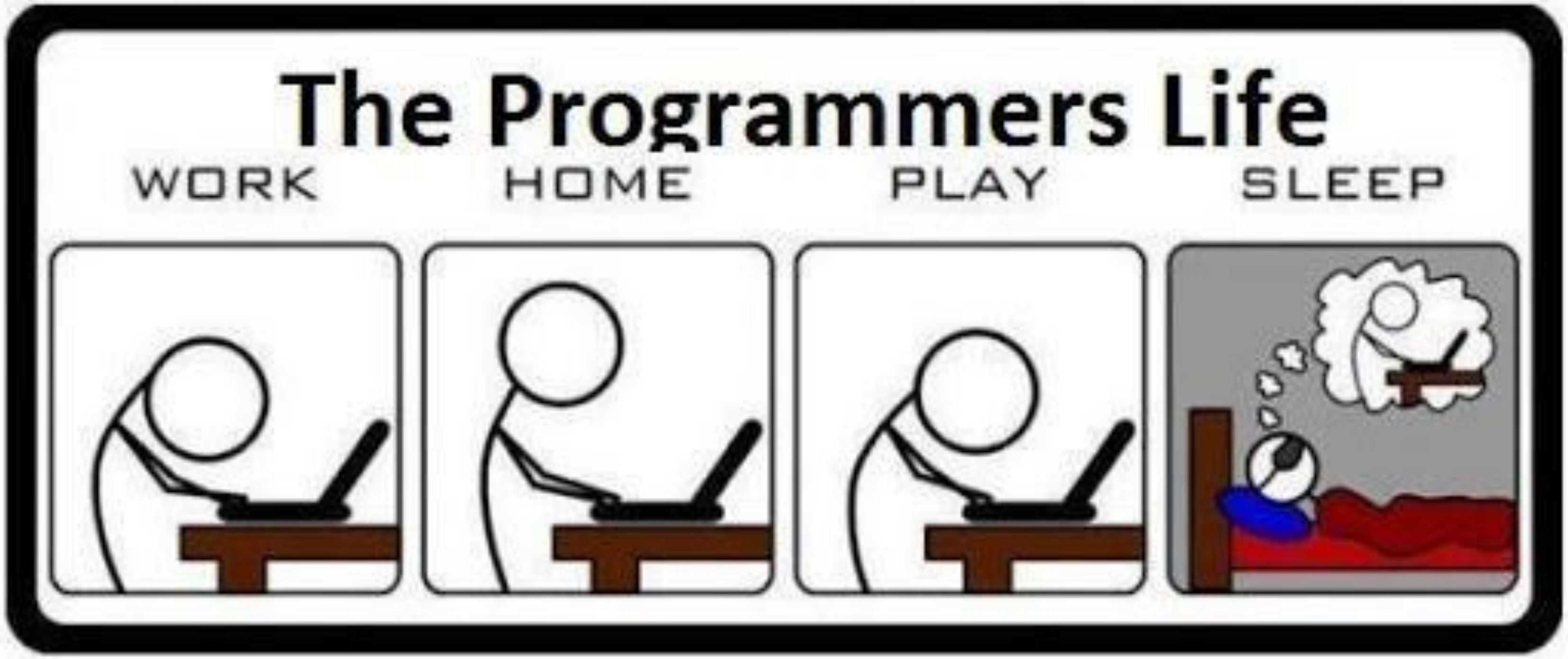
What operations installed

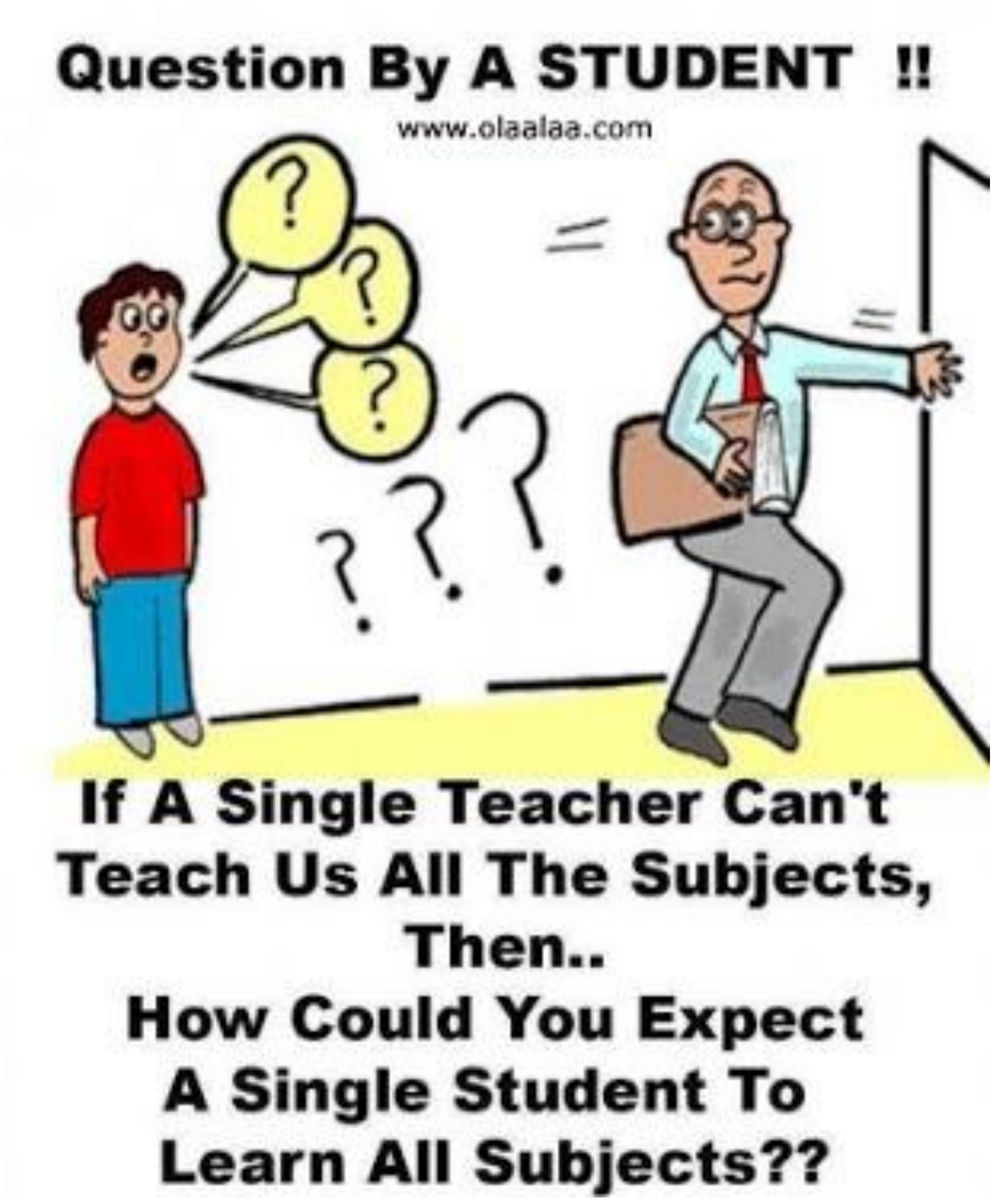
How the customer was billed

How it was supported

What marketing advertised

What the customer really needed


The Programmers Life
WORK
HOME
PLAY
SLEEP


Question By A STUDENT !!
www.olaalaa.com
?
?
?
???
If A Single Teacher Can't Teach Us All The Subjects, Then.. How Could You Expect A Single Student To Learn All Subjects??