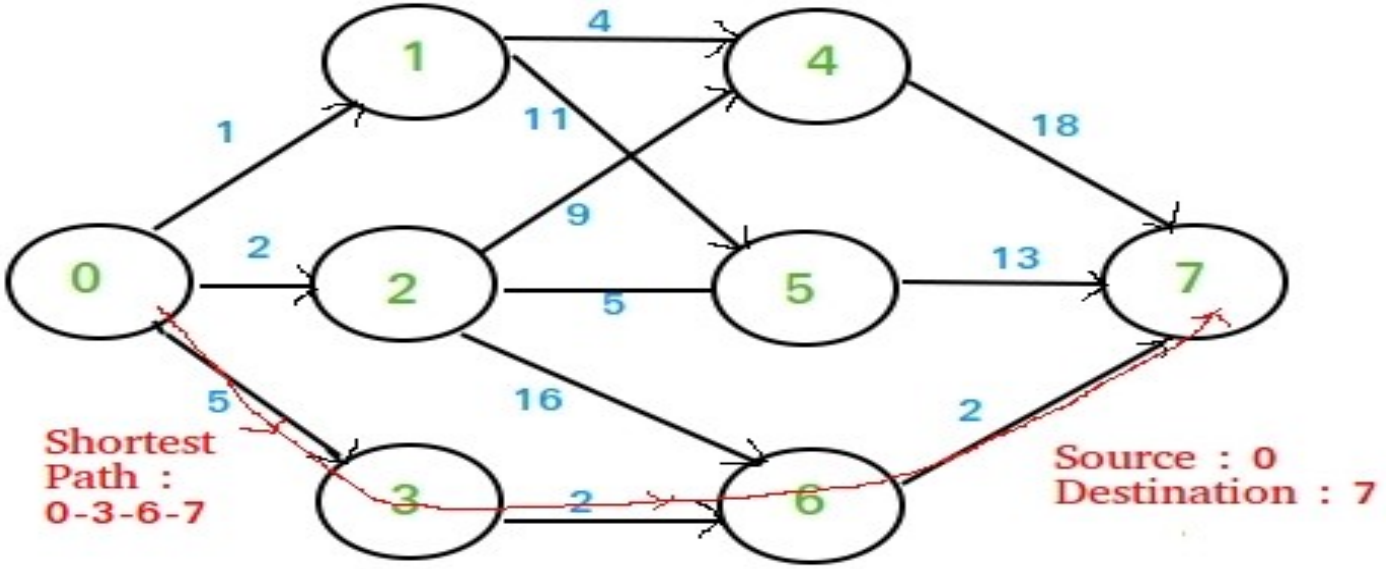
Greedy Algorithm Introduction

Dr. Bibhudatta Sahoo,
National Institute of Technology Rourkela

The Greedy Method : or Greedy heuristic

- Algorithms for optimization problems typically go through a sequence of steps, with a set of choices for each step.
- For many optimization problems, using **dynamic programming** to determine the best choices is overkill; simpler, more efficient algorithms are sufficient.
- A *greedy algorithm* always makes a **choice** that is locally optimal in the hope that it will lead to a globally optimal solution.

Multistage Graph (Shortest Path)



Simple Greedy Method – At each node, choose the shortest outgoing path. If we apply this approach to the example graph give above we get the solution as 1 + 4 + 18 = 23.

Steps to design greedy algorithms

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 that we have made, we arrive at an optimal solution to the original problem.

Introduction

- **Optimization problem**: there can be many possible solution. Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value

- **Greedy algorithm**: an algorithmic technique to solve optimization problems
  - Always makes the choice that looks best at the moment.
  - Makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution

The greedy method

- Suppose that a problem can be solved by a sequence of decisions. The greedy method has that each decision is locally optimal. These locally optimal solutions will finally add up to a globally optimal solution.
- Only a few **optimization problems** can be solved by the greedy method.

An simple example

- Problem: Pick k numbers out of n numbers such that the sum of these k numbers is the largest.
- Algorithm:

  FOR i = 1 to k

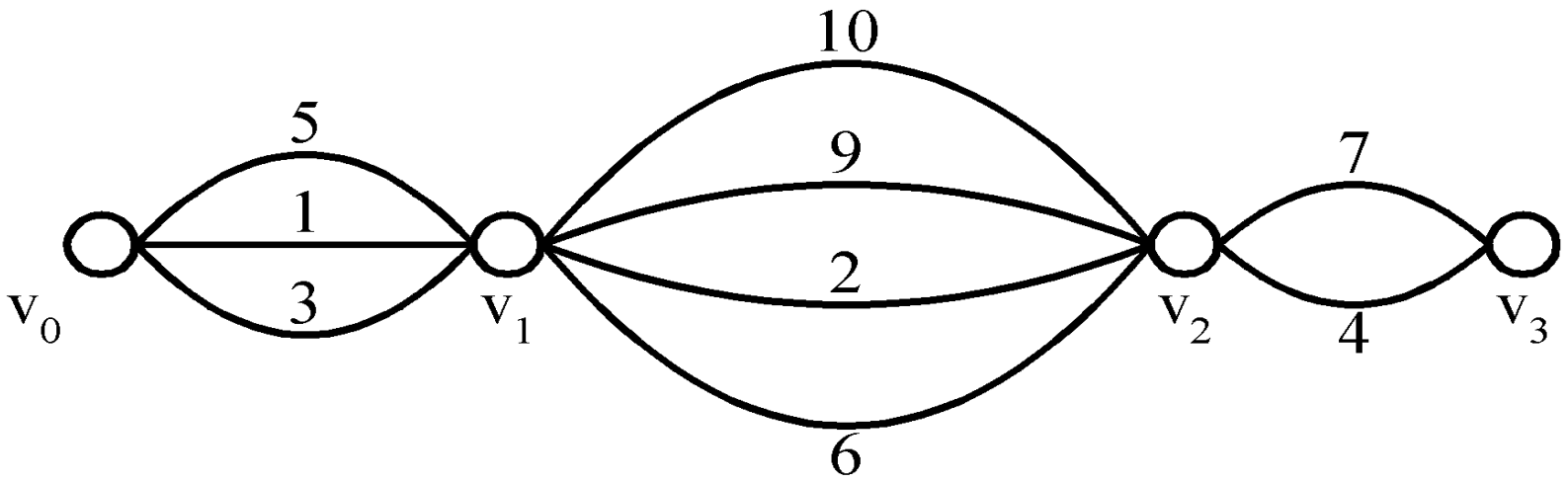
  pick out the largest number and

  delete this number from the input.

  ENDFOR

Shortest paths on a special graph

- Problem: Find a shortest path from v_0 to v_3.
- The greedy method can **solve** this problem.
- The shortest path: 1 + 2 + 4 = 7.



Optimization Problems

A problem that may have many feasible solutions.

Each solution has a value

In **maximization problem**, we wish to find a solution to maximize the value

In the **minimization problem**, we wish to find a solution to minimize the value

Greedy algorithms

An optimization problem is one in which you want to find, not just *a* solution, but the *best* solution

A "greedy algorithm" sometimes works well for optimization problems

A greedy algorithm works in phases: At each phase:

- You take the best you can get right now, without regard for future consequences
- You hope that by choosing a *local* optimum at each step, you will end up at a *global* optimum

Example: Counting money

- Suppose you want to count out a certain amount of money, using the fewest possible coins
- A greedy algorithm would do this would be: At each step, take the largest possible coin that does not overshoot
  - Example: To make Rs6.50, you can choose:
    - a Rs. 5 coin
    - a Rs. 1 coin , to make Rs. 6
    - a 50 paisa coin, to make Rs.6.50
- **For INDIAN money, the greedy algorithm always gives the optimum solution**

Greedy algorithm for optimization problem

Greedy algorithms **do not always** yield a genuinely optimal solution. In such cases the greedy method is frequently the basis of a *heuristic approach*.

Even for problems which can be solved exactly by a greedy algorithm, establishing the **correctness** of the method may be a non-trivial process.

In order to give a precise description of the greedy paradigm we must first consider a more detailed definition of the environment in which typical optimization problems occur.

Typical Steps to be used for optimization problem

- Cast the optimization problem as one in which we make a choice and are left with one sub-problem to solve.
- Prove that there's always an optimal solution that makes the **greedy choice**, so that the greedy choice is always safe.
- Show that greedy choice and optimal solution to subproblem $\Rightarrow$ optimal solution to the problem.
- Make the greedy choice and **solve top-down**.
- May have to **preprocess** input to put it into greedy order.
  - **Example:** Sorting the edges by weight to find MST.

Finding Solution to optimization Problem

- An optimisation problem involves finding a subset, *S*, from a collection of candidates, *C*; the subset, *S*, must satisfy some specified criteria, i.e. be a solution and be such that the *objective function* is optimised by *S*.
- *`Optimised'* may mean **Minimised or Maximised**
- Depending on the precise problem being solved. Greedy methods are distinguished by the fact that the **selection function** assigns a *numerical value* to each candidate, *x*, and chooses that candidate for which:
  - ***SELECT(x)*** is **largest**
  - or ***SELECT(x)*** is **smallest**

Elements of Greedy Algorithms:1

- An greedy algorithm makes a sequence of choices, each of the choices that seems best at the moment is chosen
  - NOT always produce an optimal solution
- Two ingredients that are exhibited by most problems that lend themselves to a greedy strategy
  - **Greedy-choice property**
  - **Optimal substructure**

Greedy-Choice Property

- A globally optimal solution can be arrived at by making a locally optimal (greedy) choice
  - Make whatever choice seems best at the moment and then solve the sub-problem arising after the choice is made
  - The choice made by a greedy algorithm may depend on choices so far, but it cannot depend on any future choices or on the solutions to sub-problems
- Of course, we must prove that a greedy choice at each step yields a globally optimal solution

Optimal Substructures

- A problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solutions to sub-problems
  - If an optimal solution A to S begins with activity 1, then A' = A – {1} is optimal to S'=$\{i \in S: s_i \geq f_1\}$

Elements of Greedy Algorithms:2

- Computer Scientists consider Greedy paradigm as a general design technique despite the fact that it is applicable to optimization problem.
- The Greedy techniques suggests constructing a solution to an OPTIMIZATION problem through a sequence of steps, each steps expanding a partially constructed solution obtained so far, until a complete solution to the problem is reached.
- **The choice made at each step must be:**
- # **Feasible**: It has to satisfy the problem's constraints
- # **Locally optimal**: It has to be the best local choice among all feasible choices available on that step.
- # **Irrevocable**: once made, it cannot be changed on subsequent steps of the algorithm

Form of Optimization problem in the context of Greedy algorithm

- A collection (set, list, etc) of **candidates**, e.g. nodes, edges in a graph, etc.
- A set of candidates which have already been `used'.
- A **predicate** (***solution***) to test whether a given set of candidates give a *solution* (not necessarily optimal).
- A predicate (***feasible***) to test if a set of candidates can be **extended** to a (not necessarily optimal) solution.
- A **selection function** (***select***) which chooses some candidate which h as not yet been used.
- An **objective function** which assigns a ***value*** to a solution.

Generic procedure for Greedy Method

- **Procedure GREEDY(C)**
- General frame work for greedy algorithm to find the optimal solution to the optimization problem. **C**: The set of all candidates to be the solution; **S**: The set of candidate that represent a solution, i.e. **S** $\subseteq$ **C**

1. **S** = $\varnothing$;
2. While not solution (S) and C $\neq \varnothing$ do
3. Select x $\in$ C based on the **selection criterion**
4. C $\leftarrow$ C - { x };
5. If **feasible** (S $\cup$ { x}) then
6. S $\leftarrow$ S $\cup$ { x };
7. End if;
8. End while;
9. If **Solution**(S) then
10. return(S);
11. Else
12. return (" No Solution");
13. End if;
14. End greedy;

Greedy Method Control abstraction for Subset paradigm

```
1   Algorithm Greedy(a, n)
2   // a[1 : n] contains the n inputs.
3   {
4       solution := ∅; // Initialize the solution.
5       for i := 1 to n do
6       {
7           x := Select(a);
8           if Feasible(solution, x) then
9               solution := Union(solution, x);
10      }
11      return solution;
12  }
```

A greedy algorithm has five components:

- A set of candidates, from which to create solutions.
- A selection function, to select the best candidate to add to the solution.
- A feasible function is used to decide if a candidate can be used to build a solution.
- An objective function, fixing the value of a solution or an incomplete solution.
- An evaluation function, indicating when you find a complete solution.

Greedy Method with SUBSET Paradigm

- **Algorithm GREEDY(a, n)**
- This algorithm is based on the selection of next feasible component for a solution by using a selection function SELECT() from the remaining list of candidates
- **a[1:n]** contains n inputs representing candidate set **C**

```
1.  S = ∅; //Initializing the solution //
2.  for i=1 to n  do
3.  {
4.  x := SELECT(a[i]);
5.  if  FEASIBLE (S, x) then
6.  S := S ∪ { x };
7.  }
8.  Return S;
9.  End greedy;
```

Greedy Method with ODERING Paradigm

- **Algorithm GREEDY(a, n)**
- This algorithm is based on the selection of next feasible component for a solution by using the order from the a[1:n]
- **a[1:n] contains n inputs in some order based upon the requirement by objective function**

1. S = ∅; //Initializing the solution //
2. for i=1 to n do
3. {
4. if FEASIBLE (S, a_i) then
5. S := S ∪ { a_i};
6. }
7. Return S;
8. End greedy;

Greedy Paradigm

- At every step choose the best candidates remaining
- While solving a problem, **Heap structure** to be preferred so that, for all x ∈ C, can be arranged so that the **selection process will be O(1).**
- A greedy algorithm builds solution to a problem in steps.
- At each step best available remaining candidates are selected.
- A greedy algorithm may or may not be optimal. If it is not optimal it is sometimes close to the optimal and so yields a solution that in many applications is good enough.

Greedy Paradigm

- All Greedy Algorithms have exactly the same general form.
- A Greedy Algorithm for a particular problem is specified by describing the predicates `*solution*' and `*feasible*'; and the selection function `*select*'.
- Consequently, Greedy Algorithms are often very easy to design for optimization problems.

Change-making problem

- Coin values can be modeled by a set of n distinct positive integer values (whole numbers), arranged in increasing order as w_1 through w_n.
- The problem is: given an amount W, also a positive integer, to find a set of non-negative (positive or zero) integers $\{x_1, x_2, \ldots, x_n\}$, with each x_j representing how often the coin with value w_j is used, which minimize the total number of coins $f(W)$
- $\boldsymbol{Minimize\ f(W) = \sum_{j=1}^{n} x_j}$ **subject to** $\boldsymbol{\sum_{j=1}^{n} w_j x_j = W}$

- For $W = 10$ and $\{w_1, w_2 \ldots w_n\} = \{2, 5, 3, 6\}$, there are five solutions: {2,2,2,2,2}, {2,2,3,3}, {2,2,6}, {2,3,5} and {5,5}. So the output should be **2**.

Machine Scheduling Problem

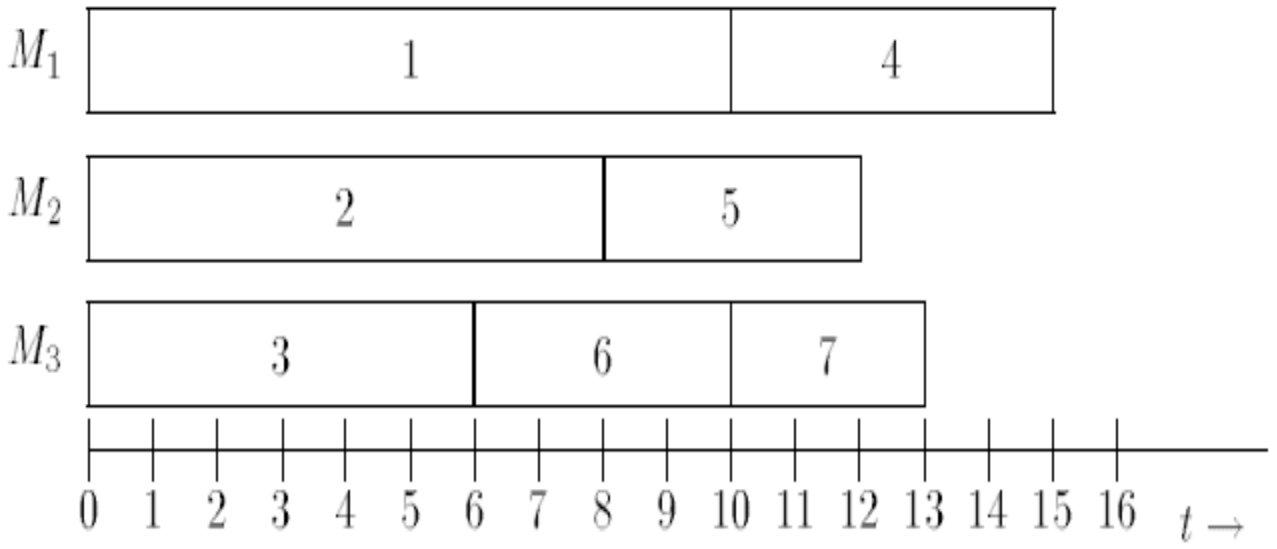
This is a typical problem in machine scheduling. It is modeled as follows. A set of n jobs has to be processed; for the processing of the jobs, there are m identical machines available from time zero onwards that can handle only one job at a time. The processing of job j $(j = 1, \ldots, n)$ takes an *uninterrupted* period of p_j units of time, and it has to be executed by a single machine. We are looking for a feasible schedule, that is, an allocation of each job j to a time interval of length p_j on a machine such that no two jobs are processed by the same machine at the same time. Given a feasible schedule, we denote the *completion time* of job j $(j = 1, \ldots, n)$ by C_j. The objective is to find a schedule in which the latest job finishes as soon as possible, i.e., we want to minimize the maximum completion time $C_{max} = \max_{j=1,\ldots,n} C_j$.

Machine Scheduling Problem

There are 3 machines available for processing 7 jobs. The processing times are given by the following table.

j	1	2	3	4	5	6	7
p_j	10	8	6	5	4	4	3

A feasible schedule is visualized by a so-called Gantt-chart as follows.



A failure of the greedy algorithm

- In some (fictional) monetary system, "krons" come in 1 kron, 7 kron, and 10 kron coins
- Using a greedy algorithm to count out 15 krons, you would get
  - A 10 kron piece
  - Five 1 kron pieces, for a total of 15 krons
  - This requires six coins
- A better solution would be to use two 7 kron pieces and one 1 kron piece
  - This only requires three coins
- The greedy algorithm results in a solution, but not in an optimal solution

Greedy algorithm Notes

- Greedy algorithms sometimes fail to produce the optimal solution, and may even produce the unique worst possible solution. One example is the travelling salesman problem
- Greedy algorithms can be characterized as being 'short sighted', and as 'non-recoverable'. They are ideal only for problems which have 'optimal substructure'.
- Greedy algorithms are best suited for simple problems (e.g. giving change).
- The greedy algorithm can be used as a selection algorithm to prioritize options within a search, or branch and bound algorithm.

Variations to the greedy algorithm:

1. **Pure greedy algorithms**
2. **Orthogonal greedy algorithms**
3. **Relaxed greedy algorithms**

- Greedy algorithms mostly (but not always) fail to find the globally optimal solution, because they usually do not operate exhaustively on all the data.
- Greedy algorithms are useful because they are quick to think up and often give good approximations to the optimum

Advantages and disadvantages [Greedy Algorithm]

- It is quite easy to **come up with a greedy algorithm** (or even multiple greedy algorithms) for a problem.
- **Analyzing the run time for greedy algorithms will generally be much easier** than for other techniques (like Divide and conquer). For the Divide and conquer technique, it is not clear whether the technique is fast or slow. This is because at each level of recursion the size of gets smaller and the number of sub-problems increases.
- The difficult part is that for greedy algorithms **you have to work much harder to understand correctness issues**. Even with the correct algorithm, it is hard to prove why it is correct. Proving that a greedy algorithm is correct is more of an art than a science. It involves a lot of creativity.

Greedy Approach vs Dynamic Programming (DP)

- Greedy and Dynamic Programming are methods for solving optimization problems.
- Greedy algorithms are usually more efficient than DP solutions.
- However, often you need to use dynamic programming since the optimal solution cannot be guaranteed by a greedy algorithm.
- DP provides efficient solutions for some problems for which a brute force approach would be very slow.
- To use Dynamic Programming we need only show that the **principle of optimality** applies to the problem.

Greedy Algorithm: Knapsack Problem

Bibhudatta Sahoo,
National Institute of Technology Rourkela

Knapsack problem; introduction

- The knapsack problem is well known in operations research literature. This problem arises whenever there is a resource allocation problem with financial constraints.
- For example, given that you have a fixed budget, how do you select what things you should buy? Assume that everything has a cost and value. We seek assignments that provide the most value for a given budget.
- The term *knapsack problem* invokes the image of a backpacker who is constrained by a fixed-size knapsack and so must fill it only with the most useful items.

Knapsack problem; introduction

- The classical knapsack problem assumes that each item must be put entirely in the knapsack or not included at all. It is this 0/1 property that makes the knapsack problem difficult.
- In the 0/1 Knapsack problem, you are given a bag which can hold W kg. There is an array of items each with a different weight and price value assigned to them.
- The objective of the 0/1 Knapsack is to maximize the value you put into the bag. You are allowed to only take all or nothing of a given item.

Knapsack problem

- *Knapsack* problem : Suppose we have n integers a_1, a_2, ..., a_n and a constant W. We want to find a subset of integers so that their sum is less than or equal to but is as close to W as possible. There are 2^n subsets. (n = 100, $2^n = 1.26765 \times 10^{30}$, it takes 4.01969×10^{12} years if our computer can examine 10^{10} subsets per second.)

- A problem is considered *tractable* (computationally easy, $O(n^k)$) if it can be solved by an efficient algorithm and *intractable* (computationally difficult, the lower bound grows fast than n^k) if there is no efficient algorithm for solving it.

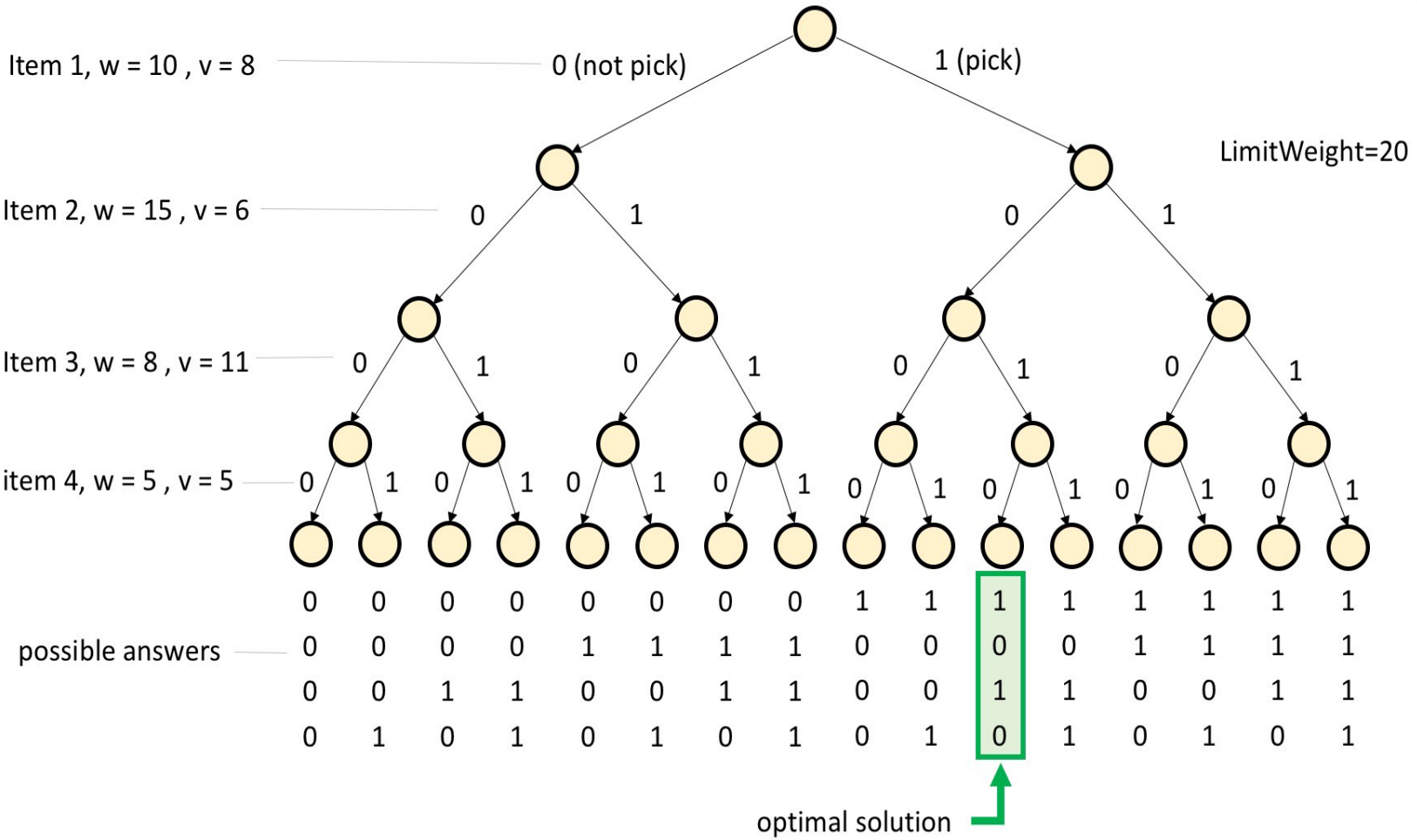
- The class of *NP-complete* problems: There is a class of problems, including TSP and Knapsack, for which no efficient algorithm is currently known.

0/1 Knapsack problem

- Given n objects 1 through n, each object i has an integer weight w_i and a real number value p_i, for $1 \leq i \leq n$.
- There is a knapsack with a total integer capacity ***W***.
- The 0–1 knapsack problem attempts to fill the sack with these objects within the weight capacity W while **maximizing** the total value of the objects included in the sack, where an object is totally included in the sack or no portion of it is in at all.
- That is, solve the following optimization problem with x_i = **0 or 1**, for $1 \leq i \leq n$

$$maximize \sum_{i=1}^{n} x_i p_i \quad subjected\ to \sum_{i=1}^{n} x_i w_i \leq W$$

4 item 0/1 Knapsack problem



Example: Greedy Method with SUBSET Paradigm

- There are n = 5 objects with integer weights $w[1..5]$ = {1,2,5,6,7}, and values $p[1..5]$ = {1,6,18,22,28}. Assuming a knapsack capacity of **11**.

Item/ weight	Status	Profit
1/1	1	1
2/2	1	1+6=7
3/5	1	7+18=25
4/6	0	25
5/7	0	25

GreedyKnapsack(Subset paradigm)

```
1.  Algorithm GreedyKnapsack(m, n)
2.  //p[1 : n] and w[1 : n] contain the profits and weights
3.  // respectively of the n objects
4.  // m is the knapsack size and x[1 : n] is the solution vector.
5.  {
6.  for i := 1 to n do x[i] := 0.0;
7.  U:=m; P:=0;
8.  for i := 1 to n do
9.  {
10. if (w[i] > U) then break;
11. x[i] := 1.0; U.:= U - w[i]; P := P + p[i]
12. }
13. Return P;
14. }
```

Example: Greedy Method with ORDERING Paradigm

- There are n = 5 objects with integer weights $w[1..5]$ = {1,2,5,6,7}, and values $p[1..5]$ = {1,6,18,22,28}. Assuming a knapsack capacity of **11**.

Item	w	p	p/w	Status	Profit
5	7	28	4	1	28
4	6	22	3.66	0	
3	5	18	3.6	0	
2	2	6	3	1	28+6=34
1	1	1	1	1	34+1=35

Optimal solution :40

GreedyKnapsack(ordering paradigm)

```
1.  Algorithm GreedyKnapsack(m, n)
2.  //p[1 : n] and w[1 : n] contain the profits and weights respectively
3.  //of the n objects ordered such that p[i]/w[i] ≥ p[i +1]/w[i + 1].
4.  // m is the knapsack size and x[1 : n] is the solution vector.
5.  {
6.  for i := 1 to n do x[i] := 0.0;
7.  U:=m; P :=0;
8.  for i := 1 to n do
9.  {
10. if (w[i] > U) then break;
11. x[i] := 1.0; U:= U - w[i]; P := P + p[i]
12. }
13. Return P;
14. }
```

O-1 knapsack is harder!

- 0-1 knapsack cannot be solved by the greedy strategy
  - Unable to fill the knapsack to capacity, and the empty space lowers the effective value per rupees of the packing
  - We must compare the solution to the sub-problem in which the item is included with the solution to the sub-problem in which the item is excluded before we can make the choice
  - **Dynamic Programming**

Knapsack Problem: Text Book page

- Suppose there are n objects and a knapsack or bag with maximum capacity m.
- An object i has a weight w_i and p_i is the profit associated with the object i.
- If a fraction x_i, $0 \le x_i \le 1$, of object i is place into the knapsack, then a profit of $p_i x_i$ is earned.
- The objective is to obtain a filling of the knapsack that maximizes the total profit earned.
- Since the knapsack capacity is m, we require the total weight of all chosen objects to be at most m.

The problem can be stated as

$$\text{maximize} \sum_{1 \le i \le n} p_i x_i \qquad (4.1)$$

$$\text{subject to} \sum_{1 \le i \le n} w_i x_i \le m \qquad (4.2)$$

$$\text{and } 0 \le x_i \le 1, \quad 1 \le i \le n \qquad (4.3)$$

The fractional knapsack problem

- n objects, each with a weight $w_i > 0$; a profit $p_i > 0$; capacity of knapsack: M

Maximize $$\sum_{1 \le i \le n} p_i x_i$$

Subject to $$\sum_{1 \le i \le n} w_i x_i \le M$$

$0 \le x_i \le 1,\ 1 \le i \le n$

The fractional knapsack algorithm

- The greedy algorithm:

  Step 1: Sort p_i/w_i into non-increasing order.

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

- e. g.

  n = 3, M = 20, $(p_1, p_2, p_3) = (25, 24, 15)$

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

  Sol: $p_1/w_1 = 25/18 = 1.32$

  $p_2/w_2 = 24/15 = 1.6$

  $p_3/w_3 = 15/10 = 1.5$

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

Greedy Algorithm for Fractional Knapsack problem

- Fractional knapsack can be solvable by the greedy strategy
  - Compute the value per pound p_i/w_i for each item
  - Obeying a greedy strategy, take as much as possible of the item with the greatest value per pound.
  - If the supply of that item is exhausted and there is still more room, take as much as possible of the item with the next value per pound, and so forth until there is no more room
  - O($n \log n$) (we need to sort the items by value per Rupees)
  - Greedy Algorithm?
  - Correctness?

On 0/1 knapsack problem

- 0/1 Knapsack is a typical problem that is used to demonstrate the application of **greedy algorithms** as well as **dynamic programming**. There are cases when applying the greedy algorithm does not give an optimal solution. There are many flavors in which *Knapsack problem* can be asked.

- 1. A thief enters a museum and wants to steal artifacts from there. Every artifact has a **weight** and **value** associated with it. Thief carries a knapsack (bag) which can take only a specific weight. Problem is to find the combination of artifacts thief steals so that he gets maximum value and weight of all taken artifacts is less capacity of knapsack he has. A thief cannot take any artifact partially. Either he takes it or leaves it. Hence the problem is 0/1 knapsack.

On 0/1 knapsack problem

- 2. We have N files each having a size say S_i. We have a total storage capacity of W bytes. For each file to be stored re-computation cost is V_i. Problem is to store as many files on storage that combined size of all files is less than W and their re-computation value is maximum. We can either store or leave a file, we cannot store a partial file. Hence this is a case of 0/1 knapsack problem.

The 0/1 knapsack problem statement

$$x_i = copies\ of\ each\ kind\ of\ item$$

$$v_i = value$$

$$w_i = weight$$

$$W = maximum\ weight\ capacity$$

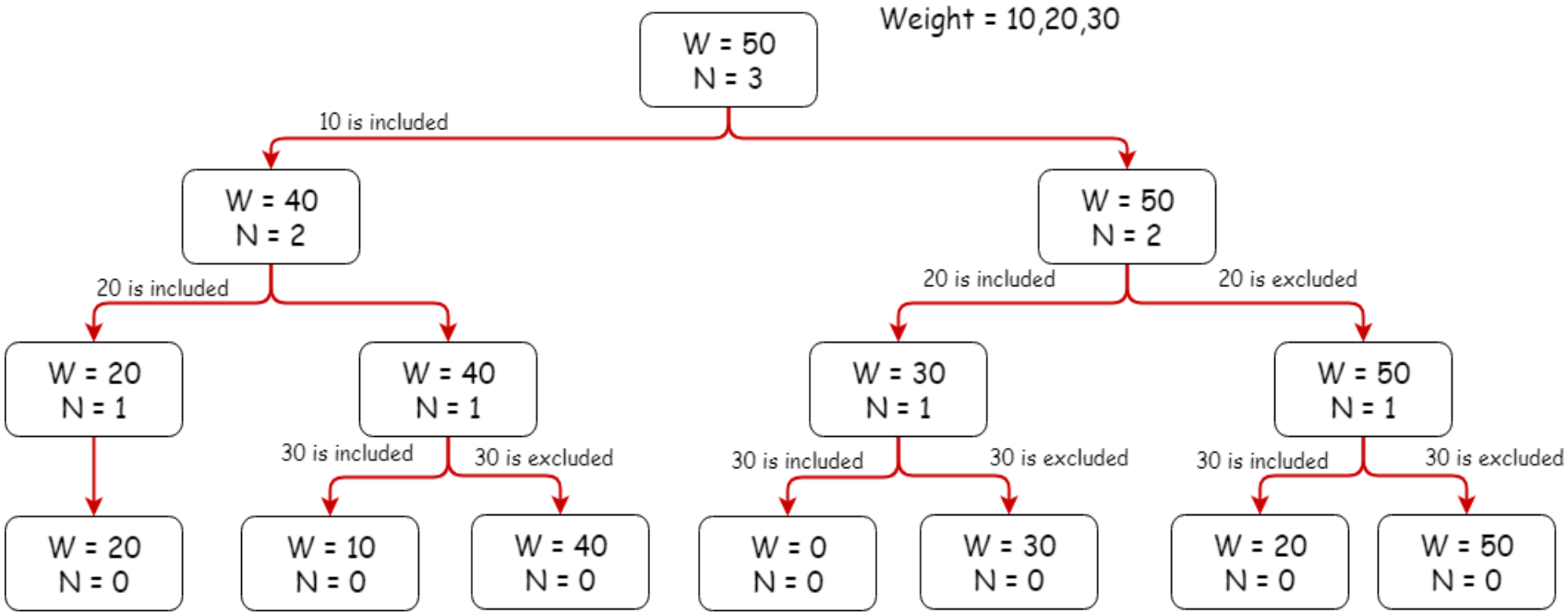
$$i = items\ numbered\ 1..n$$

$$maximize \sum_{i=1}^{n} v_i\, x_i$$

$$subject\ to \sum_{i=1}^{n} w_i\, x_i \leq W, x_i \in \{0,1\}$$

The 0/1 knapsack problem: example

- With val = {60, 100, 120}; wt = {10, 20, 30}; with knapsack capacity 50, there are seven problems to be solved at the leaf level. For n = 3, there are 7 problems to be solved before we start optimizing for the max value. For n, in general, it will take 2^n subproblems to be solved. Hence, complexity of recursive implementation is $O(2^n)$.



Advantages and disadvantages [Greedy Algorithm]

- It is quite easy to **come up with a greedy algorithm** (or even multiple greedy algorithms) for a problem.
- **Analyzing the run time for greedy algorithms will generally be much easier** than for other techniques (like Divide and conquer). For the Divide and conquer technique, it is not clear whether the technique is fast or slow. This is because at each level of recursion the size of gets smaller and the number of sub-problems increases.
- The difficult part is that for greedy algorithms **you have to work much harder to understand correctness issues**. Even with the correct algorithm, it is hard to prove why it is correct. Proving that a greedy algorithm is correct is more of an art than a science. It involves a lot of creativity.

Thanks for Your Attention!

Exercises

Exercises

2. [0/1 Knapsack] Consider the knapsack problem discussed in this section. We add the requirement that $x_i = 1$ or $x_i = 0$, $1 \leq i \leq n$; that is, an object is either included or not included into the knapsack. We wish to solve the problem

$$\max \sum_{1}^{n} p_i x_i$$

$$\text{subject to} \sum_{1}^{n} w_i x_i \leq m$$

$$\text{and} \quad x_i = 0 \text{ or } 1, \quad 1 \leq i \leq n$$

One greedy strategy is to consider the objects in order of nonincreasing density p_i/w_i and add the object into the knapsack if it fits. Show that this strategy doesn't necessarily yield an optimal solution.

Exercises

1. What are the general characteristic of greedy algorithms and the problems soled by these algorithm?
2. Design a greedy algorithm for **coin-changing problem**. Under what condition, the greedy algorithm always yields an optimal solution.

   coin-changing problem: Suppose there are n types of coin denominations, d_1, d_2, …, and d_n. (We may assume one of them is penny.) There are an infinite supply of coins of each type. To make change for an arbitrary amount j using the minimum number of coins.

3. What is **Principle of optimality?** Differentiate between greedy and dynamic method.

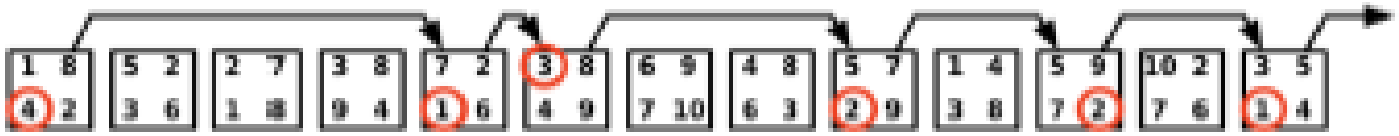
Exercises

4. **Chained matrix multiplication Problem**: Given a series of n arrays (of appropriate sizes) to multiply: $A_1 \times A_2 \times \cdots \times A_n$. Determine where to place parentheses to minimize the number of multiplications. Does any greedy approach to the chained matrix multiplication work?

- Idea: For matrix multiplication our idea of a "move" was to place the *outermost* parentheses. When we did this, we broke our original problem up into two smaller instances of the matrix chain multiplication problem. Could we make different kinds of "moves" so that we would always have only one sub problem to solve? This would certainly simplify things, since we would no longer have this big ugly tree of recursive calls.
- For matrix multiplication, a different kind of "move" might be multiplying two matrices in the sequence together, essentially placing *innermost* parentheses, resulting in a shorter chain of matrices, because two elements of the chain have been replaced with one - their product. Each time you make such a "move" you find yourself faced with the same kind of problem, just smaller, so you simply make another move until no moves remain.

Exercises

5. Consider the following solitaire game played on a row of n squares. Each square contains four positive integers. The player begins by placing a token on the leftmost square. On each move, the player chooses one of the numbers on the token's current square and then moves the token that number of squares to the right. The game ends when the token moves past the rightmost square. The object of the game is to make as many moves as possible before the game ends.



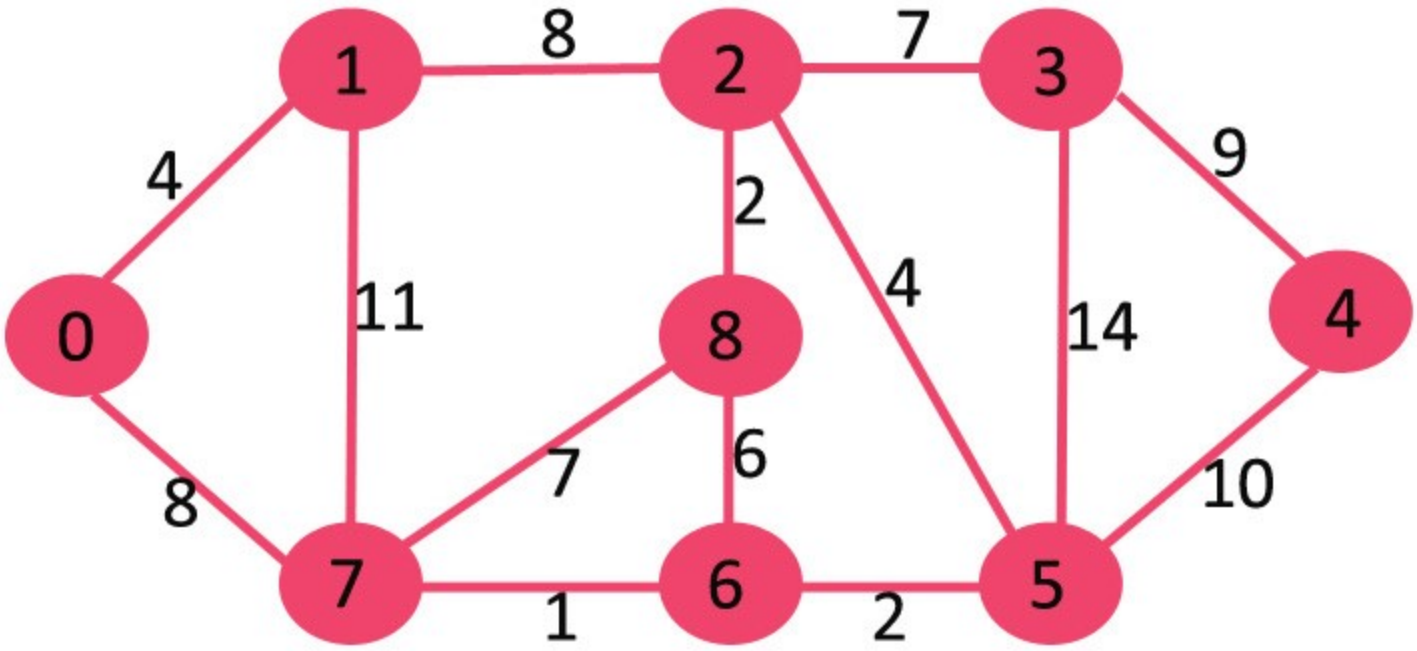
An instance of the puzzle that allows six moves. (This is *not* the longest legal sequence of moves.)

(a) Prove that the obvious greedy strategy (always choose the smallest number) does not give the largest possible number of moves for every instance of the puzzle. Your proof should be a small counterexample and a description of the optimal sequence of moves that does better than the greedy strategy.

(b) Describe and analyze an efficient algorithm to find the largest possible number of legal moves for a given instance of the puzzle. Your algorithm should ideally run in $O(n)$ time. *[Hint: Two possible strategies are to design a dynamic programming algorithm or to reduce to longest paths in directed acyclic graphs. Any correct $O(n)$ time algorithm is worth full credit.]*

Exercises

6. Compare Greedy and dynamic programming approach for algorithm Design. Explain How both can be used to solve Knapsack problem?
7. Compare Prim's & Kruskal's method for finding Minimum spanning Tree find MST for following using prims method.



Programming questions

Problem 1: Given an array of N integer, we have to maximize the sum of arr[i] * i, where i is the index of the element (i = 0, 1, 2, …, N). We can rearrange the position of the integer in the array to maximize the sum.

- Time complexity of Brute Force Approach: O(N × N!)
- Time complexity of Greedy Algorithm: O(N logN)

Problem 2: Given two arrays array_One[] and array_Two[] of same size N. We need to first rearrange the arrays such that the sum of the product of pairs(1 element from each) is minimum. That is SUM (A[i] * B[i]) for all i is minimum.

- Time complexity of Brute Force Approach: $O((N!)^2)$
- Time complexity of Greedy Algorithm: O(N logN)

Programming questions

Problem 3: Given an array of non-negative integers. Our task is to find minimum number of elements (subset) such that their sum should be greater than the sum of rest of the elements of the array.

- Time complexity of Brute Force Approach: $O(2^N)$
- Time complexity of Greedy Algorithm: O(N logN)

Problem 4: For a given array of elements, we have to find the non-empty subset having the minimum product.

- Time complexity of Brute Force Approach: $O(2^N)$
- Time complexity of Greedy Algorithm: O(N)

Problem 5: For numbers from 1 to given n, we have to find the division of the elements into two groups having minimum absolute sum difference. We will explore two techniques:

- Time complexity of Brute Force Approach: $O(2^N)$
- Time complexity of Greedy Algorithm: O(N)

Programming questions

Problem 6: For a given array, find the largest lexicographic array which can be obtained from it after performing K consecutive swaps. Given two arrays A1 and A2, A1 is defined to be lexicographically larger than A2 if for the first i (from 0 to length of smaller array), element at index i is greater than element at index i of A2.

- Time complexity of Brute Force Approach: O(N!)
- Time complexity of Greedy Algorithm: O(N)

Problem 7: Given a number N, our task is to find the largest perfect cube that can be formed by deleting minimum digits (possibly 0) from the number. Any digit can be removed from the given number to reach the goal.

- Time complexity of Brute Force Approach: $O(2^N)$
- Time complexity of Greedy Algorithm: O(N^(1/3)log(N)log(N))

Programming questions

Problem 8: Given a set of n activities with their start and finish times, we need to select maximum number of non-conflicting activities that can be performed by a single person, given that the person can handle only one activity at a time.

- Time complexity of Brute Force Approach: $O(2^N)$
- Time complexity of Greedy Algorithm: O(N logN)

Problem 9: A maximal clique is a clique that cannot be extended by including one more adjacent vertex, that is, a clique which does not exist exclusively within the vertex set of a larger clique. The problem is to find a single Maximal Clique.

- Time complexity of Brute Force Approach: $O(2^V)$
- Time complexity of Greedy Algorithm: $O(V^2)$

Programming questions

Problem 10: Given a number N, the problem is to count the maximum number of composite numbers that sum up to N. First few composite numbers are 4, 6, 8, 9, 10, 12, 14, 15...

- Time complexity of Brute Force Approach: $O(2^N)$
- Time complexity of Greedy Algorithm: O(1)

Problem 11: The problem we will solve is that given a number N, we need to find the minimum number of fibonacci numbers required that sums up to a given number N. Note a fibonacci number can be repeated.

- Time complexity of Brute Force Approach: O(N)
- Time complexity of Greedy Algorithm: O(log N)

Question & Answer

[1] Describe the main ideas behind greedy algorithms.

- Greedy algorithms are based on the idea of optimizing locally. When a greedy algorithm needs to make a choice, it chooses what will be best in the short term, without thinking about future consequences, and commits to its decision.

[2] What problem does Dijkstra's algorithm solve?

- Single source shortest paths.

[3] What problem does Prim's algorithm solve?

- Prim's algorithm computes minimal spanning trees.

Question & Answer

- A ***vertex cover*** of an undirected graph G is a set S of vertices such every edge in G has at least one of its endpoints in S. The vertex cover problem is: Given an undirected graph G, find a vertex cover of G that has the smallest possible size. That is, find a set of as few vertices as possible such that every edge hits at least one of the vertices in this set. The following is a proposed algorithm for the vertex cover problem. It uses the notion of the degree of a vertex, which is the number of edges that hit this vertex.

1. S = empty set.
2. While G has at least one edge
3. Find a vertex v of the highest possible degree in G
4. Add v to S.
5. Remove vertex v from G, along with all edges that hit v
6. Answer S.

Question & Answer

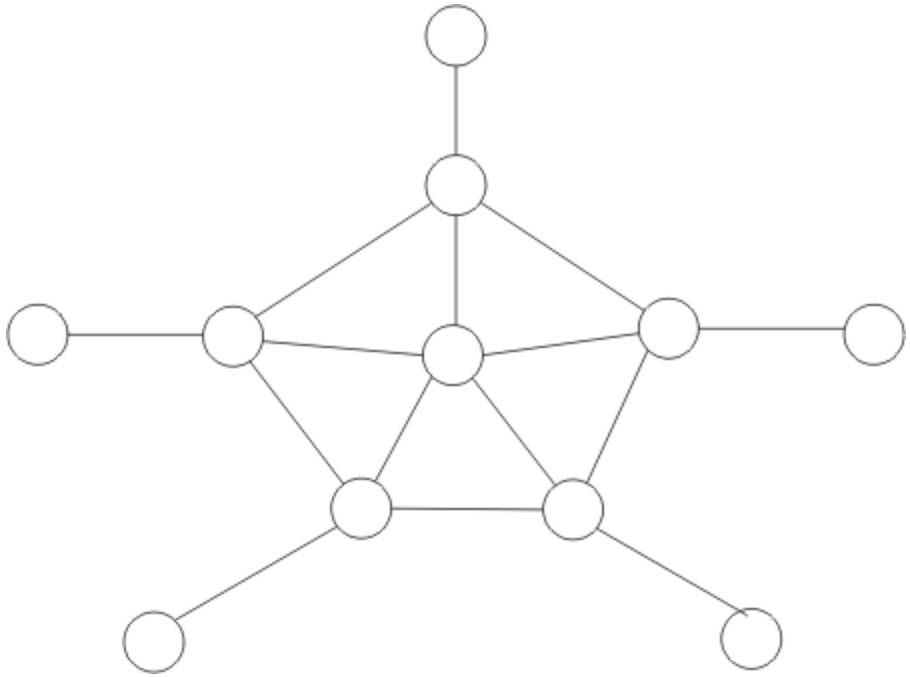
- A vertex cover needs to *cover* all of the edges, where an edge is covered by choosing a vertex from that edge. This algorithm is based on the principle that, by choosing a vertex with the most edges hitting it, we get cover the most edges.

What kind of algorithm is this? Is it guaranteed to find the smallest vertex cover? If so, why? If not, give a counterexample.

- This is a greedy algorithm, since it tries to get the most edges as possible with the next vertex, without thinking ahead.
- This algorithm does not work. The following graph is a counterexample. The greedy algorithm will start by selecting the center vertex, which is the only vertex of degree 5. But the smallest vertex cover contains the five vertices that form a pentagon, and does not use the center vertex.

Question & Answer

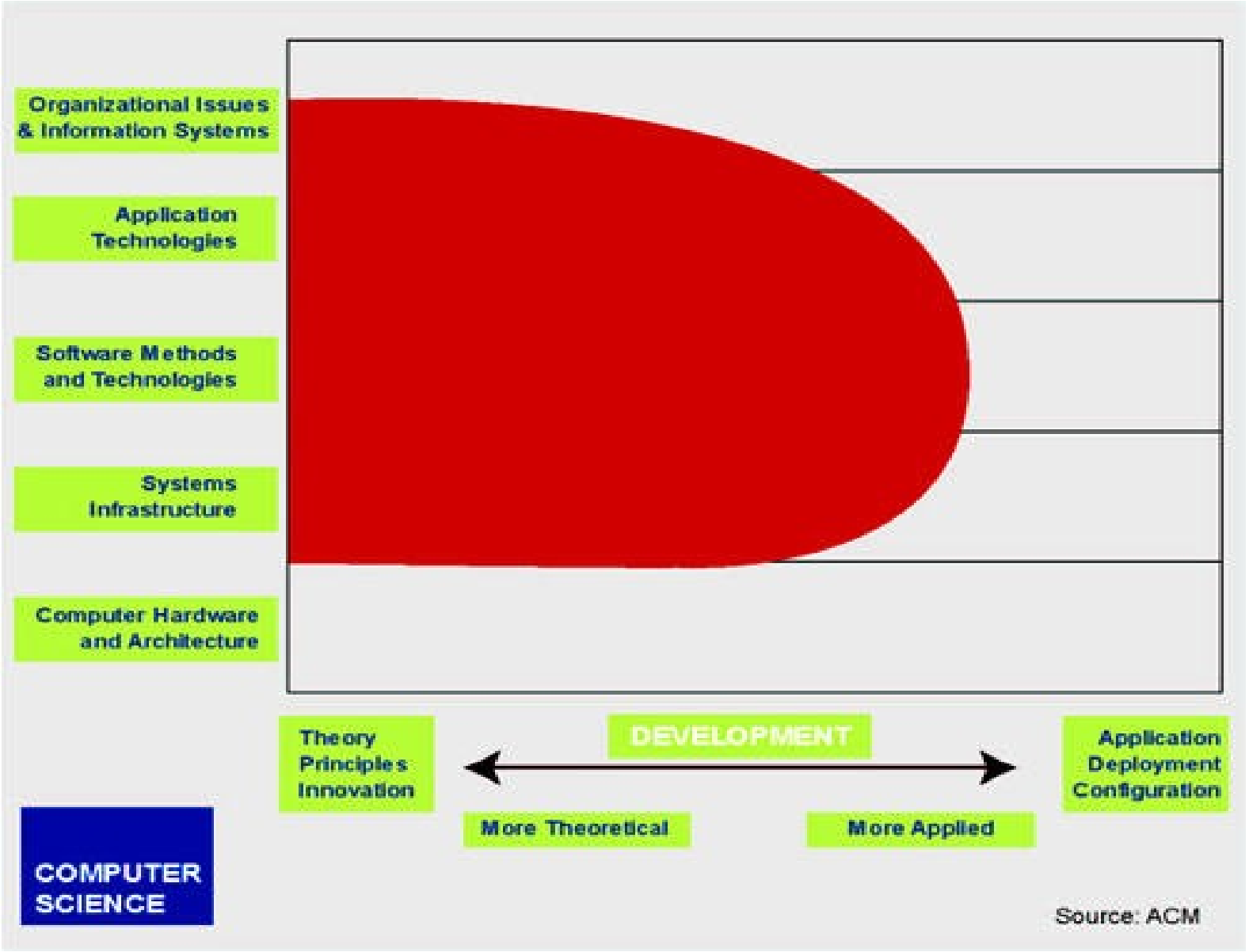
- This algorithm does not work. The following graph is a counterexample. The greedy algorithm will start by selecting the center vertex, which is the only vertex of degree 5. But the smallest vertex cover contains the five vertices that form a pentagon, and does not use the center vertex.



Reference

- http://www.algorithmsandme.com/tag/coin-change-problem/
- https://iq.opengenus.org/greedy-algorithms-you-must-attempt/

Thank you


Organizational Issues & Information Systems
Application Technologies
Software Methods and Technologies
Systems Infrastructure
Computer Hardware and Architecture
Theory Principles Innovation
DEVELOPMENT
Application Deployment Configuration
More Theoretical
More Applied
COMPUTER SCIENCE
Source: ACM