CSE225: Data Structure and Algorithms

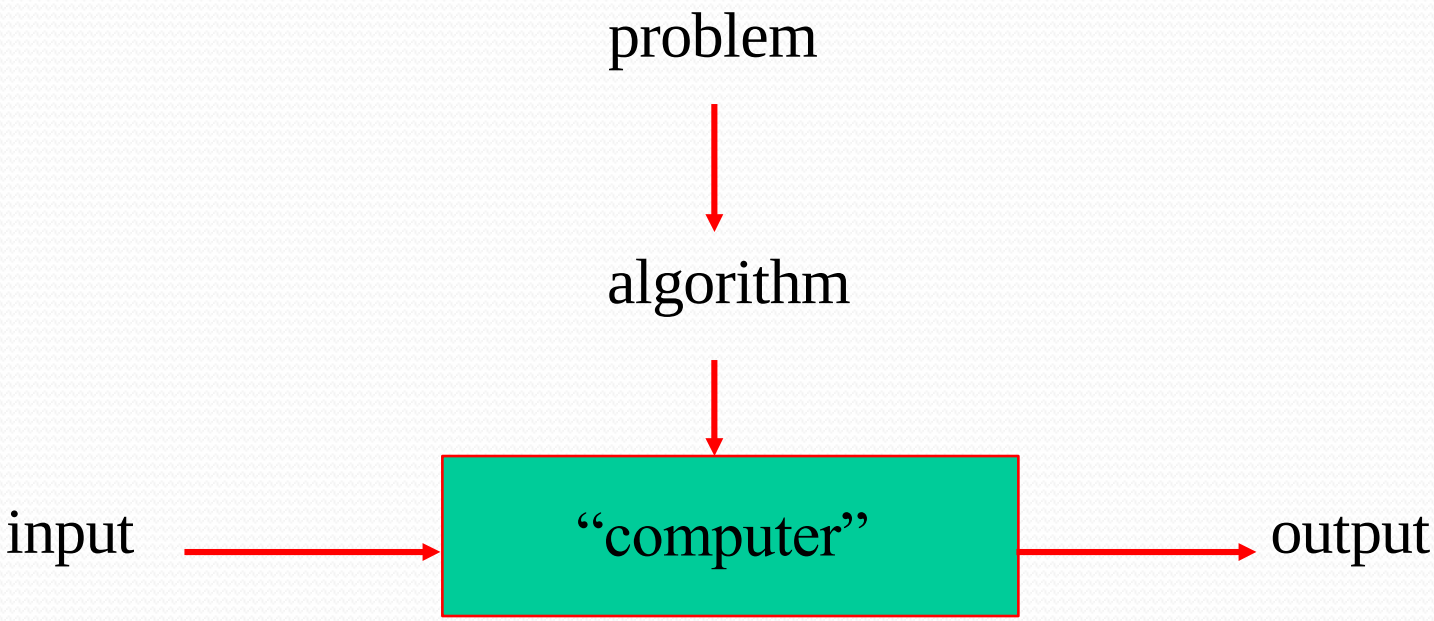
Types of Algorithms

Algorithm

An Algorithm is any well-defined computational procedure that takes some values or set of values as input and produces some values or set of values as output.

An Algorithm is a well defined list of steps to solve a particular problem.

NOTION OF ALGORITHM



Algorithmic solution

A short list of categories

- Algorithm types we will consider include:
  - Simple recursive algorithms
  - Backtracking algorithms
  - Divide and conquer algorithms
  - Dynamic programming algorithms
  - Greedy algorithms
  - Branch and bound algorithms
  - Brute force algorithms
  - Randomized algorithms

SOME WELL-KNOWN COMPUTATIONAL PROBLEMS

- Sorting
- Searching
- Shortest paths in a graph
- Minimum spanning tree
- Primality testing
- Traveling salesman problem
- Knapsack problem
- Chess
- Towers of Hanoi
- Program termination

Algorithms

Properties of a good Algorithm:

- **Input** from a specified set,
- **Output** from a specified set (solution),
- **Effectiveness** of each calculation step and
- **Finiteness** of the number of calculation steps,
- **Correctness** of output for every possible input

Types of Algorithms

Algorithm classification

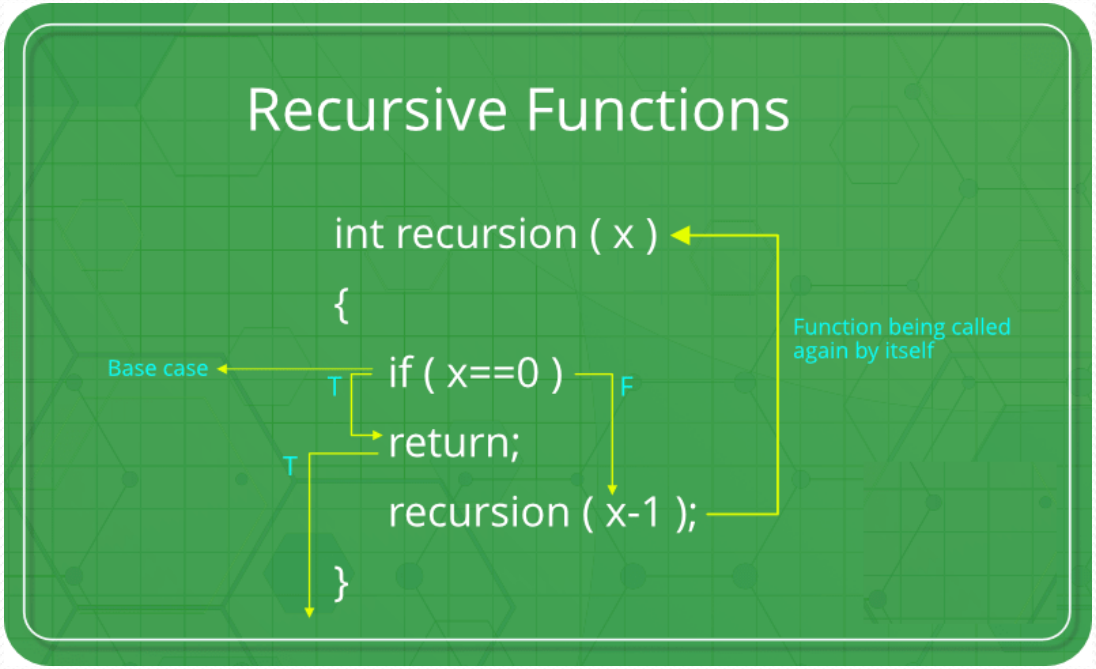
- Algorithms that use a similar problem-solving approach can be grouped together
- This classification scheme is neither exhaustive nor disjoint
- The purpose is not to be able to classify an algorithm as one type or another, but to highlight the various ways in which a problem can be attacked

A short list of categories

- Algorithm types we will consider include:
  - Simple recursive algorithms
  - Backtracking algorithms
  - Divide and conquer algorithms
  - Dynamic programming algorithms
  - Greedy algorithms
  - Branch and bound algorithms
  - Brute force algorithms
  - Randomized algorithms

What is Recursion?

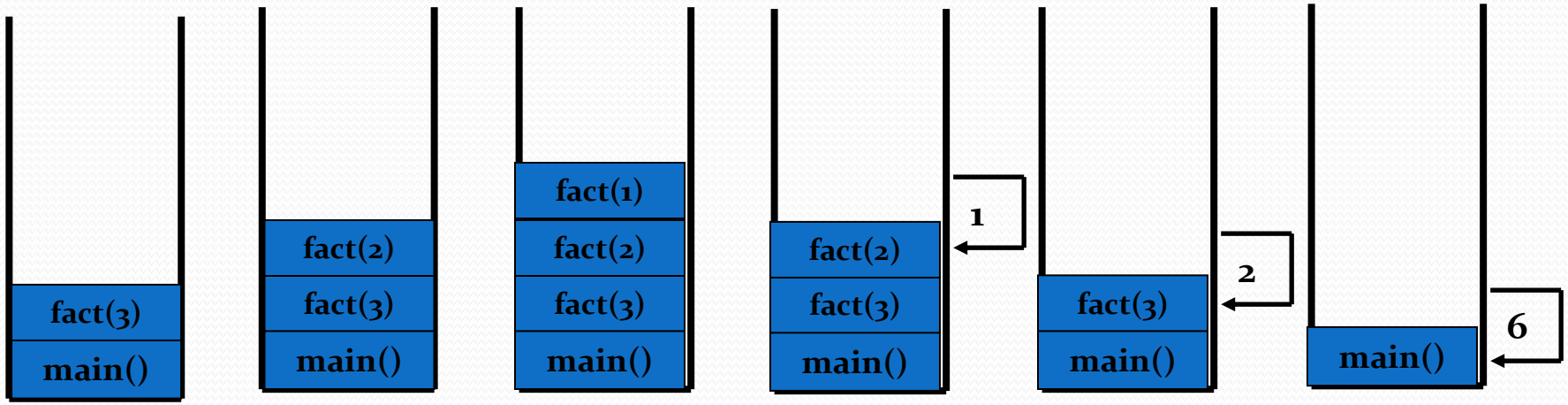
- The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function. Using recursive algorithm, certain problems can be solved quite easily. Examples of such problems are Towers of Hanoi (TOH), Inorder/Preorder/ Postorder Tree Traversals, DFS of Graph, etc.



Simple recursive algorithms I

- A simple recursive algorithm:
  - Solves the base cases directly
  - Recurs with a simpler subproblem
  - Does some extra work to convert the solution to the simpler subproblem into a solution to the given problem
- "simple" because several of the other algorithm types are inherently recursive

Finding the factorial of 3



Time 2:
Push: fact(3)

Time 3:
Push: fact(2)

Time 4:
Push: fact(1)

Time 5:
Pop: fact(1)
returns 1.

Time 6:
Pop: fact(2)
returns 2.

Time 7:
Pop: fact(3)
returns 6.

Inside findFactorial(3):
if (number <= 1) return 1;
else return (3 * factorial (2));

Inside findFactorial(2):
if (number <= 1) return 1;
else return (2 * factorial (1));

Inside findFactorial(1):
if (number <= 1) return 1;
else return (1 * factorial (0));

Complexity of Recursive Algorithm

```
int factorial(int n) {
  if (n == 1)
    return 1;
  else
    return n * factorial(n-1);
}
```

T(1) = 1
T(2) = T(1) + 1 = 2
T(3) = T(2) + 1 = 3
T(4) = T(3) + 1 = 4
…
T(n) = n
So in general this algorithm will require n units of work to complete (i.e. T(n) = n).
So O(n)

Backtracking Algorithm

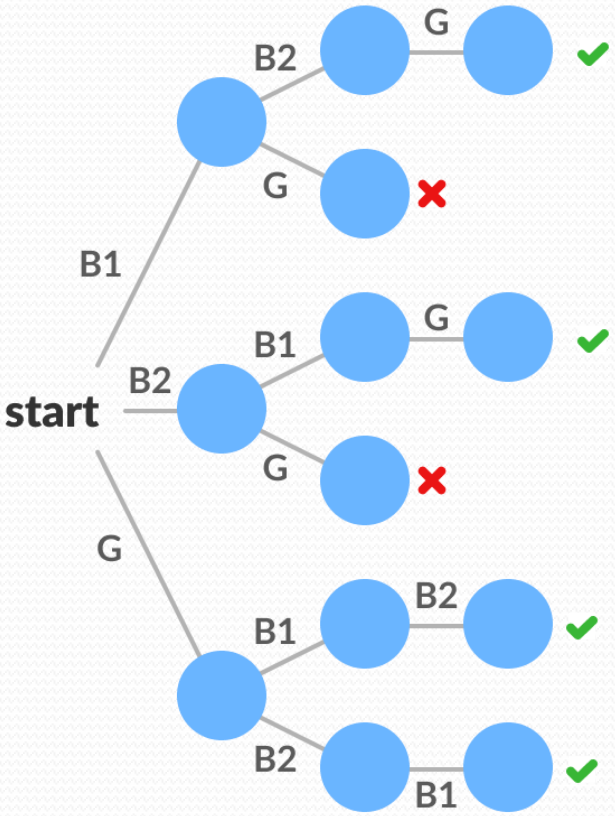
- Backtracking is a general algorithm for finding all (or some) solutions to some computational problems, notably constraint satisfaction problems, that incrementally builds candidates to the solutions, and abandons a candidate ("backtracks") as soon as it determines that the candidate cannot possibly be completed to a valid solution.[1]

- The classic textbook example of the use of backtracking is the eight queens puzzle, that asks for all arrangements of eight chess queens on a standard chessboard so that no queen attacks any other.

Backtracking algorithms

- Backtracking algorithms are based on a depth-first recursive search
- A backtracking algorithm:
  - Tests to see if a solution has been found, and if so, returns it; otherwise
  - For each choice that can be made at this point,
    - Make that choice
    - Recur
    - If the recursion returns a solution, return it
  - If no choices remain, return failure

Example



If there are three students two boys and a girl and there are three chairs. We have to arrange them in those three chairs in how many ways we can arrange there are three number of students are three so we can arrange them in 3 factorial ways. That is six ways.

Complexity of Backtracking Algorithm

Backtracking algo:
n-queen problem:**O(n!)**
graph coloring problem:**O(nm^n)**//where n=no. of vertex,m=no. of color used
hamilton cycle:**O(N!)**
WordBreak and StringSegment:**O(2^N)**
subset sum problem:**O(nW)**

Divide and Conquer

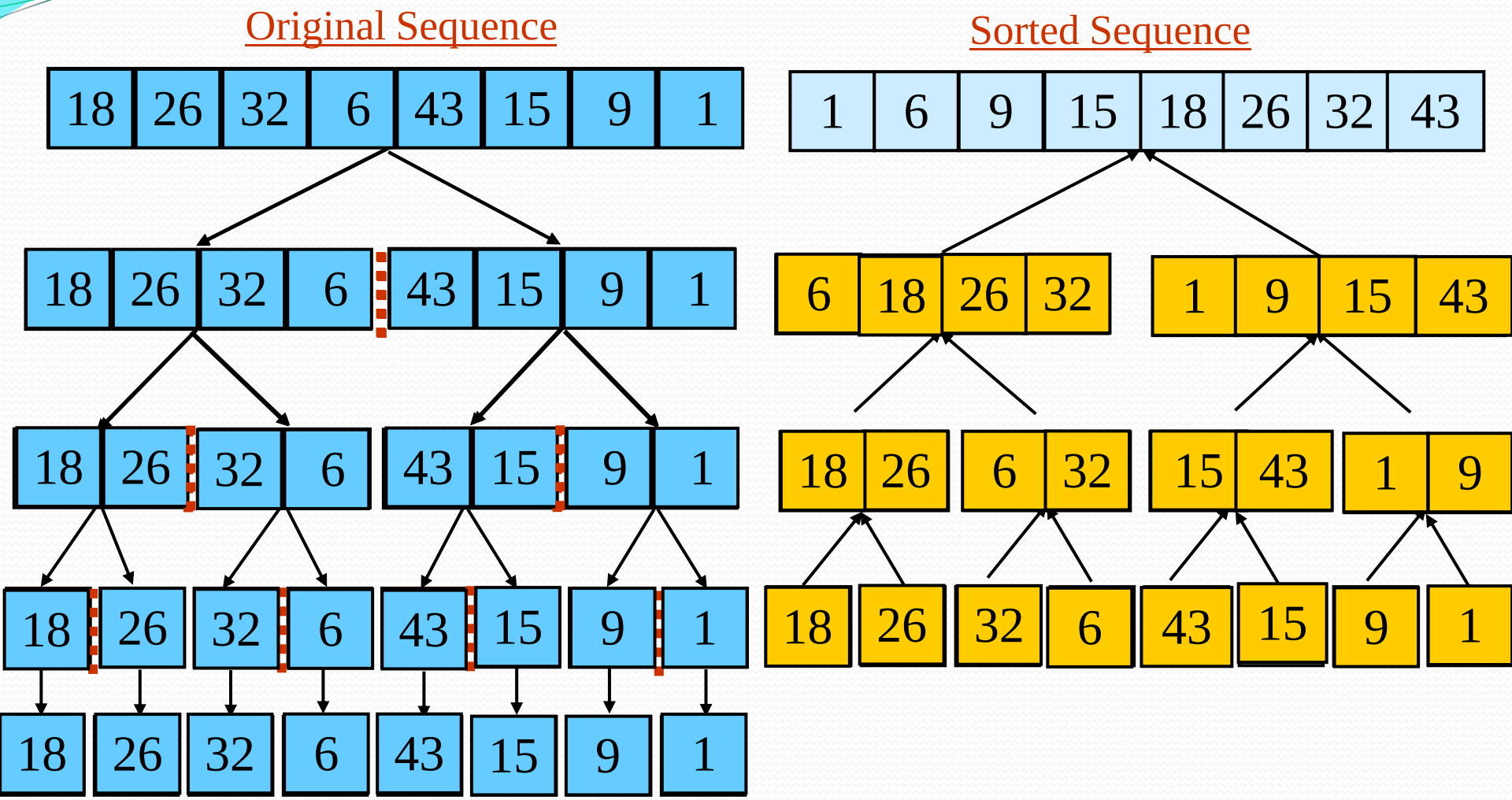
- **Recursive in structure**
  - ***Divide*** the problem into independent sub-problems that are similar to the original but smaller in size
  - ***Conquer*** the sub-problems by solving them recursively. If they are small enough, just solve them in a straightforward manner.
  - This can be done by reducing the problem until it reaches the base case, which is the solution.
  - ***Combine*** the solutions of the sub-problems to create a solution to the original problem

Example: Merge Sort

***Sorting Problem*:** Sort a sequence of *n* elements into non-decreasing order.

- ***Divide*:** Divide the *n*-element sequence to be sorted into two subsequences of *n/2* elements each
- ***Conquer:*** Sort the two subsequences recursively using merge sort.
- ***Combine*:** Merge the two sorted subsequences to produce the sorted answer.

Merge Sort – Example



Divide and Conquer

- A divide and conquer algorithm consists of two parts:
  - Divide the problem into smaller subproblems of the same type, and solve these subproblems recursively
  - Combine the solutions to the subproblems into a solution to the original problem
- Traditionally, an algorithm is only called divide and conquer if it contains two or more recursive calls

Divide and conquer algorithm

- divide and conquer is an algorithm design paradigm. A divide-and-conquer algorithm recursively breaks down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

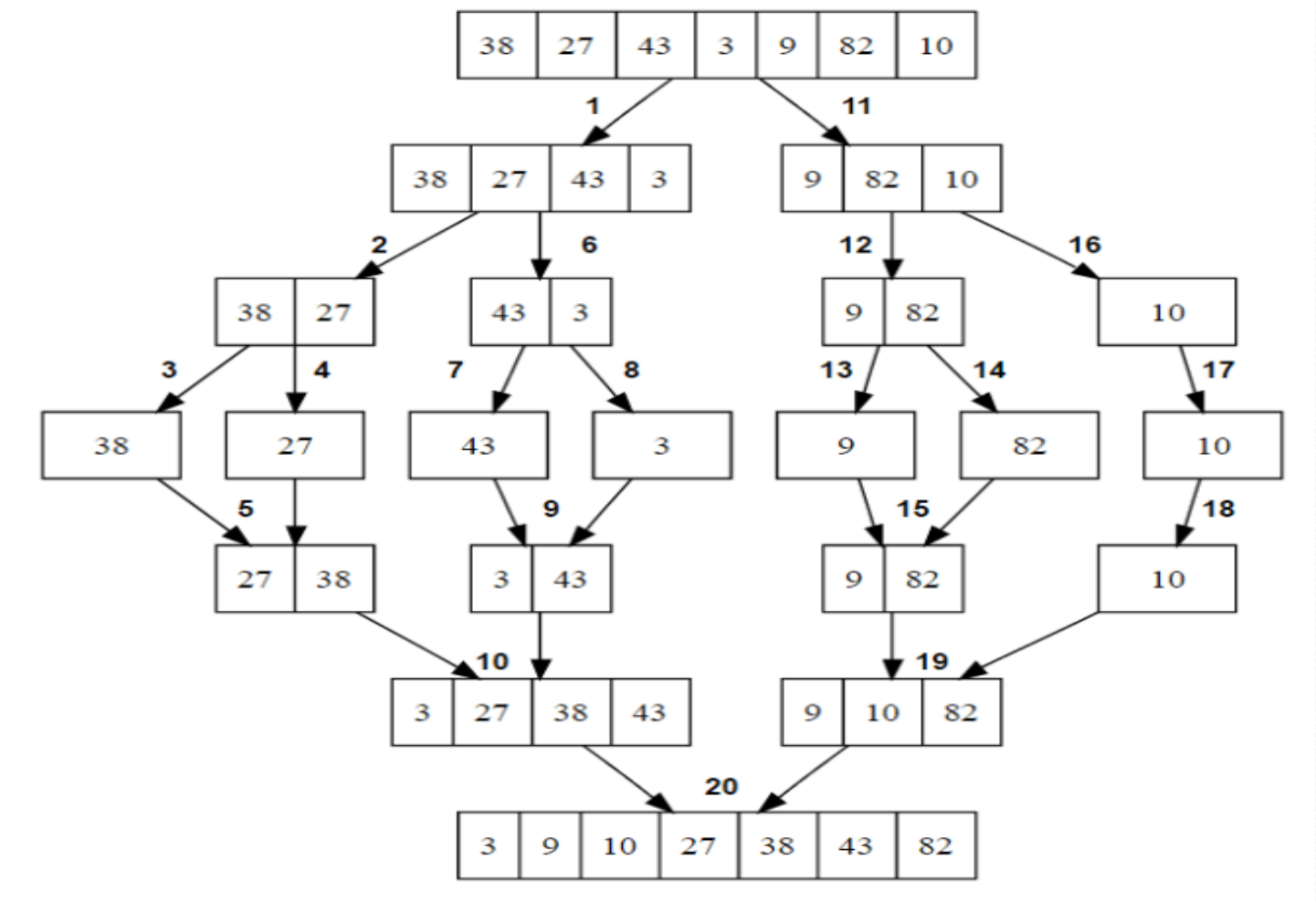
- The divide-and-conquer technique is the basis of efficient algorithms for many problems, such as sorting (e.g., quicksort, merge sort), multiplying large numbers (e.g., the Karatsuba algorithm), finding the closest pair of points, syntactic analysis (e.g., top-down parsers), and computing the discrete Fourier transform (FFT).

Examples

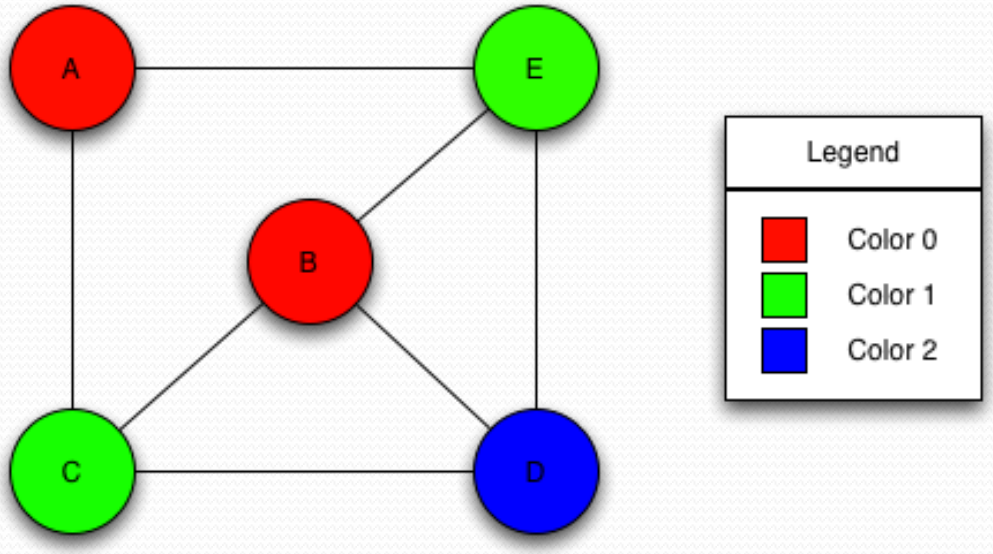
- Quicksort:
  - Partition the array into two parts, and quicksort each of the parts
  - No additional work is required to combine the two sorted parts
- Mergesort:
  - Cut the array in half, and mergesort each half
  - Combine the two sorted arrays into a single sorted array by merging them

Merge Sort Example

Complexity of Merge sort: O(n*log n)



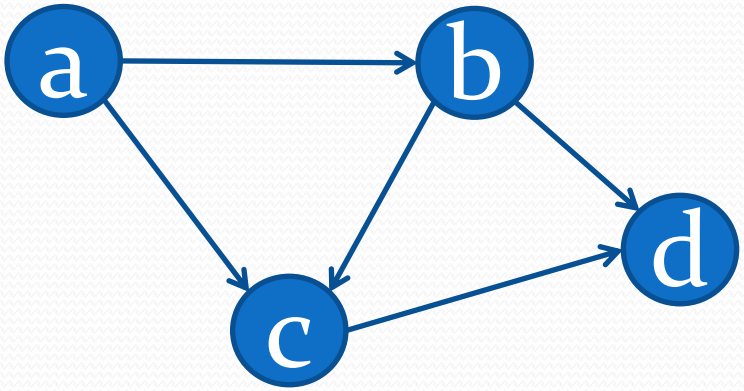
Greedy Algorithm

- **Decision Problems:**
  - 3-coloring *decision problem*: given an undirected graph, G= (V,E), determine if there is a way to assign a "color" chosen from {0, 1, 2} to each vertex in such a way that no two adjacent vertices have the same color
  - 

  - Path *decision problem*: Is there any path from u to v in G?

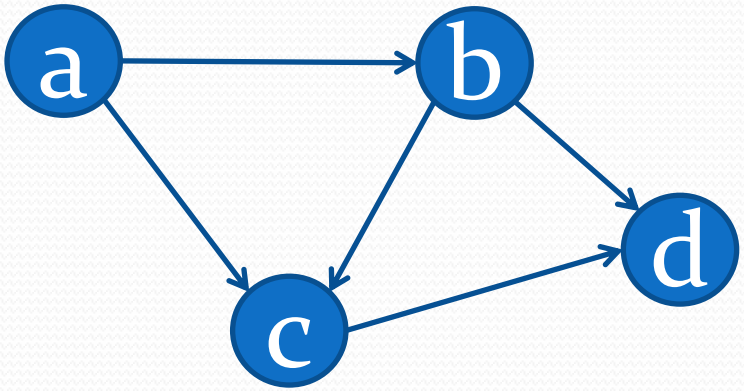
- **Search problem:**

- 3-coloring *search problem*: given an undirected graph G = (V, E) find, if it exists, a coloring c: V → {1, 2, 3} of the vertices, such that for every (u, v) ∈ E: c(u) ≠ c(v)
- Path *search problem*: Find all the paths from u to v in G



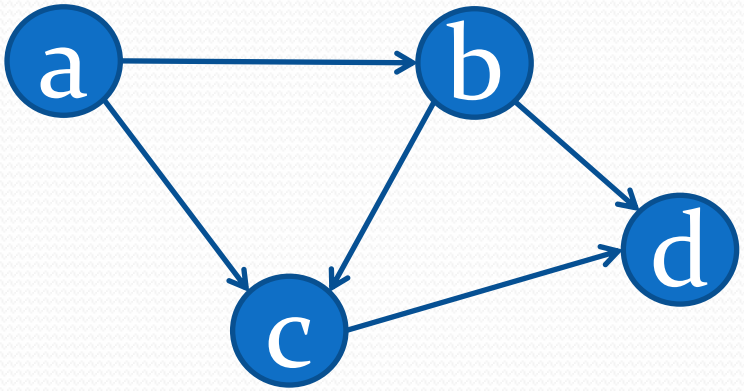
Find all the paths from a to d:
a →b →d
a →c →d
a →b →c →d

- **Counting Problem:** Compute the number of solutions to a given search problem
  - 3-coloring *counting problem*: given an undirected graph G = (V, E) compute the number of ways we can color the graph using only 3 colors: {1,2,3}. While coloring we have to ensure that no two adjacent vertices have the same color.
  - Path *counting Problem*: Compute the number of possible paths from u to v in a given graph G.



Compute the number of paths from a to d: **3**

- **Optimization Problems:** Find the best possible solution among the set of all possible solutions to a search problem
  - Graph coloring *optimization problem*: given an undirected graph, G= (V,E), what is the minimum number of colors required to assign "colors" to each vertex in such a way that no two adjacent vertices have the same color
  - Path *optimization problem*: Find the shortest path(s) from u to v in a given graph G.



Find all the **shortest** paths from a to d:
a →b →d
a →c →d

Greedy Algorithm

- Solves an optimization problem
- Example:
  - Activity Selection Problem
  - Dijkstra's Shortest Path Problem
  - Minimum Spanning Tree Problem
- For many optimization problems, greedy algorithm can be used. (not always)
- Greedy algorithm for optimization problems typically go through a sequence of steps, with a set of choices at each step. Current choice does not depend on evaluating potential future choices or pre-solving repeatedly occurring subproblems (a.k.a., *overlapping subproblems*). With each step, the original problem is reduced to a smaller problem.
- Greedy algorithm always makes the choice that looks best at the moment.
- It makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.

Optimal Solution

What is an Optimal Solution?

- Given a problem, more than one solution exist
- One of the solution is the best based on some given constraints, that solution is called the optimal solution

What is Global Optimal Solution?

- Optimal Solution to the main problem

What is local Optimal Solution?

- Optimal Solution to the subproblems

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

Greedy algorithm

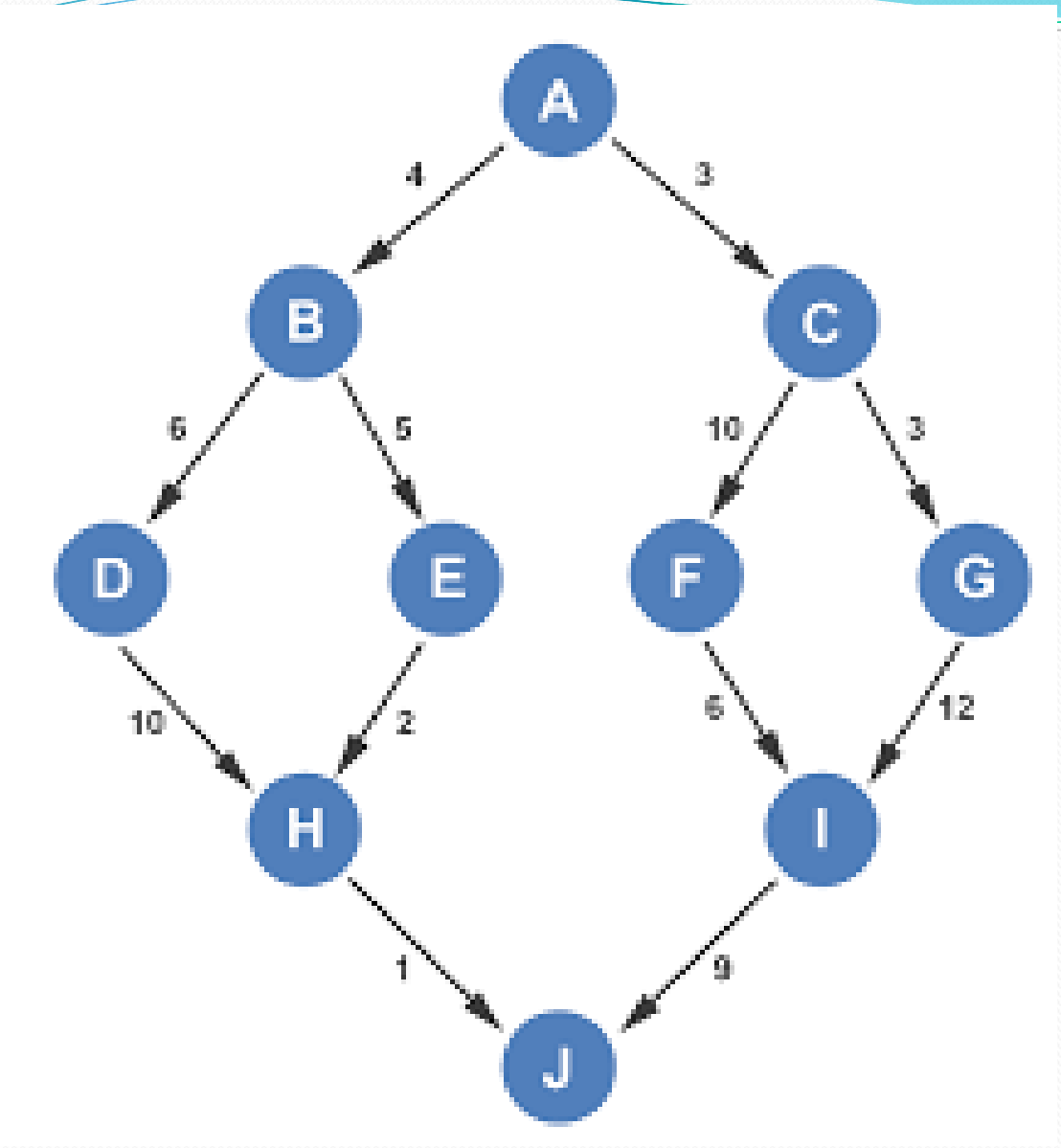
- A greedy algorithm is any algorithm that follows the problem-solving heuristic of making the locally optimal choice at each stage.[1] In many problems, a greedy strategy does not usually produce an optimal solution, but nonetheless, a greedy heuristic may yield locally optimal solutions that approximate a globally optimal solution in a reasonable amount of time.

Example: Huffman Coding, TSP

- For example, a greedy strategy for the travelling salesman problem (which is of a high computational complexity) is the following heuristic: "At each step of the journey, visit the nearest unvisited city." This heuristic does not intend to find a best solution, but it terminates in a reasonable number of steps; finding an optimal solution to such a complex problem typically requires unreasonably many steps. In mathematical optimization, greedy algorithms optimally solve combinatorial problems having the properties of matroids, and give constant-factor approximations to optimization problems with submodular structure.

Traveling salesperson problemor TSP

- The travelling salesman problem asks the following question: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?" It is an NP-hard problem in combinatorial optimization, important in theoretical computer science and operations research.


A
4
3
B
C
6
5
10
3
D
E
F
G
10
2
6
12
H
I
1
9
J

Complexity of Greedy Algorithm

- complexity is O(nlogn).

Dynamic Programming

Dynamic Programming **is a general algorithm design technique for solving problems defined by recurrences with overlapping subproblems**

- **Invented by American mathematician Richard Bellman in the 1950s to solve optimization problems and later assimilated by CS**

- **"Programming" here means "planning"**

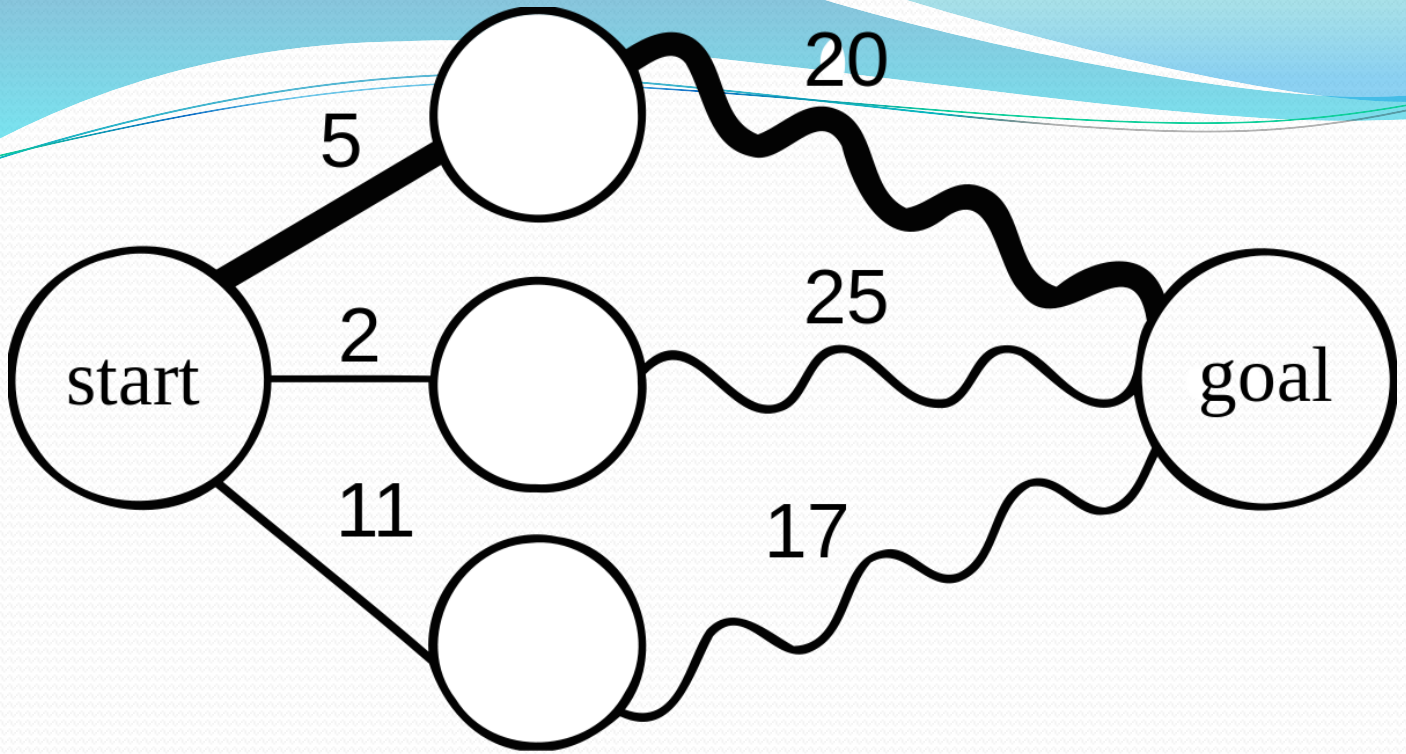
- **Main idea:**
  1. set up a recurrence relating a solution to a larger instance to solutions of some smaller instances
  2. solve smaller instances once
  3. record solutions in a table
  4. extract solution to the initial instance from that table

Dynamic Programming (DP)

- Build up the solution by computing solutions to the subproblems.
- Don’t solve the same subproblem twice, but rather save the solution so it can be re-used later on.
- Often used for a large class to optimization problems.
- Unlike Greedy algorithms, implicitly solve all subproblems.
- Motivating the case for DP with Memoization – a top-down technique, and then moving on to Dynamic Programming – a bottom-up technique.

Dynamic programming algorithms

- A dynamic programming algorithm remembers past results and uses them to find new results
- Dynamic programming is generally used for optimization problems
  - Multiple solutions exist, need to find the "best" one
  - Requires "optimal substructure" and "overlapping subproblems"
    - Optimal substructure: Optimal solution contains optimal solutions to subproblems
    - Overlapping subproblems: Solutions to subproblems can be stored and reused in a bottom-up fashion
- This differs from Divide and Conquer, where subproblems generally need not overlap



if a problem can be solved optimally by breaking it into sub-problems and then recursively finding the optimal solutions to the sub-problems, then it is said to have optimal substructure

Complexity of Dynamic Programming

- In Dynamic programming problems, Time Complexity is **the number of unique states/subproblems * time taken per state**.
- In this problem, for a given n, there are **n** unique states/subproblems. For convenience, each state is said to be solved in a constant time. Hence the time complexity is **O(n * 1).**
- Hence the time complexity is **O(n) or linear.**

Branch and bound algorithm

- **Branch and bound** is an algorithm design paradigm which is generally used for solving combinatorial optimization problems. These problems are typically exponential in terms of time complexity and may require exploring all possible permutations in worst case. The Branch and Bound Algorithm technique solves these problems relatively quickly.

Example branch and bound algorithm

- Travelling salesman problem: A salesman has to visit each of n cities (at least) once each, and wants to minimize total distance travelled
  - Consider the root problem to be the problem of finding the shortest route through a set of cities visiting each city once
  - Split the node into two child problems:
    - Shortest route visiting city A first
    - Shortest route *not* visiting city A first
  - Continue subdividing similarly as the tree grows

Branch and bound algorithms

- Branch and bound algorithms are generally used for optimization problems
  - As the algorithm progresses, a tree of subproblems is formed
  - The original problem is considered the "root problem"
  - A method is used to construct an upper and lower bound for a given problem
  - At each node, apply the bounding methods
    - If the bounds match, it is deemed a feasible solution to that particular subproblem
    - If bounds do *not* match, partition the problem represented by that node, and make the two subproblems into children nodes
  - Continue, using the best known feasible solution to trim sections of the tree, until all nodes have been solved or trimmed

Printing Items in 0/1 Knapsack

- Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack. In other words, given two integer arrays val[0..n-1] and wt[0..n-1] which represent values and weights associated with n items respectively. Also given an integer W which represents knapsack capacity, find out the items such that sum of the weights of those items of given subset is smaller than or equal to W. You cannot break an item, either pick the complete item, or don't pick it (0-1 property).

- Input : val[] = {60, 100, 120};
  wt[] = {10, 20, 30};
  W = 50;

Output : 220 //maximum value that can be obtained
  30 20 //weights 20 and 30 are included.

- Input : val[] = {40, 100, 50, 60};
  wt[] = {20, 10, 40, 30};
  W = 60;

Output : 200
  30 20 10

Complexity of 0-1 knapsack problem

- O (nC) where n is number of elements in knapsack and C is a capacity of knapsack.

Brute force algorithm

- A brute force algorithm simply tries *all* possibilities until a satisfactory solution is found
  - Such an algorithm can be:
    - Optimizing: Find the *best* solution. This may require finding all solutions, or if a value for the best solution is known, it may stop when any best solution is found
      - Example: Finding the best path for a travelling salesman
    - Satisficing: Stop as soon as a solution is found that is *good enough*
      - Example: Finding a travelling salesman path that is within 10% of optimal

Traveling salesman problem (TSP

- A classic example in computer science is the traveling salesman problem (TSP). Suppose a salesman needs to visit 10 cities across the country. How does one determine the order in which those cities should be visited such that the total distance traveled is minimized?

- The brute force solution is simply to calculate the total distance for every possible route and then select the shortest one. This is not particularly efficient because it is possible to eliminate many possible routes through clever algorithms
- The time complexity of brute force is **O(mn)**, which is sometimes written as **O(n*m)** .

Randomized algorithms

- A randomized algorithm uses a random number at least once during the computation to make a decision
  - Example: In Quicksort, using a random number to choose a pivot
  - Example: Trying to factor a large prime by choosing random numbers as possible divisors

Crossing the Rivers

- Woman, G1 G2
- Man, B1 B2
- Police, Thieves

- 1. Boat can carry only 2 persons
- 2. Woman, Man and Police can ride the boat
- 3. a. If Woman absent Men killed woman's Girls
  b. If Man absent then Woman killed man's boys.
  c. If Police absent then Thieves killed everybody.