DESIGN AND ANALYSIS OF ALGORITHMS

(CSB 252)

PREPARED BY :
MOTUPALLI VISHISHTA (181210031)
SUMITRA SIVAKUMAR (181210053)

ALGORITHM DESIGN TECHNIQUES

1. Divide and Conquer
2. Backtracking
3. Greedy Method
4. Dynamic Programming
5. Branch and Bound

DIVIDE AND CONQUER

INTRODUCTION

- Based on multi-branched recursion.
- Recursively breaks down a problem into two or more sub-problems, until it becomes simple enough to solve the problem directly.
- Solutions of sub-problems are combined to give solution to the original problem.

INTRODUCTION

- Used to find optimal solution of a problem.
- Few standard algorithms that follow D&C algorithm are :
  - Binary Search
  - Quick Sort
  - Merge Sort
  - Strassen’s Algorithm
- Few advantages of D&C algorithm are :
  - Solving difficult problems
  - Algorithm efficiency
  - Memory Access

WORKING OF ALGORITHM

1.**Divide :** Divide the given problem into subproblems using recursion.

2.**Conquer :** Solve the smaller subproblems recursively. If the subproblem is small enough, solve it directly.

3.**Combine :** Combine the solutions of the sub problems which is part of the recursive process to achieve the solution.

REAL LIFE PROBLEM

PROBLEM STATEMENT :

To **search sorted sequence** using Divide and Conquer algorithm with the **aid of Fibonacci numbers**.

SOLUTION :

Using Fibonacci numbers, we calculate middle element of data array to search the data item. The time complexity of this approach is **O(log(n)).**

APPROACH TO PROBLEM

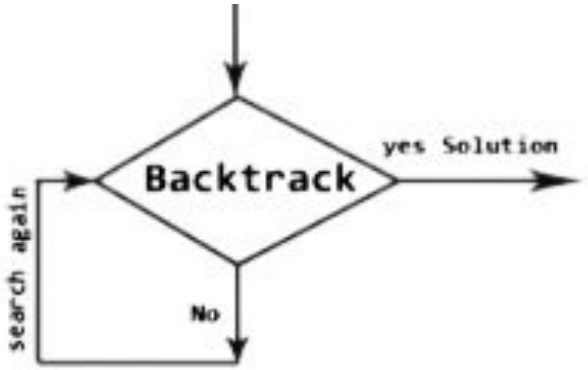
1. Assign the data to the array in a sorted manner.
2. Take input of the element to be searched.
3. Call FibonacciSearch() function.
4. Calculate the mid value using ‘start+fib[index-2]’ expression.
5. If the chosen item is equal to the value at mid index, print result and return to main function.
6. If it is lesser than the value at mid index, proceed with the left sub-array.
7. If it is more than the value at mid index, proceed with the right sub-array.
8. If the calculated mid value is equal to either start or end then the item is not found in the array.

BACKTRACKING

INTRODUCTION

- Solves problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point of time.

- Searches every possible combination in order to solve a computational problem.



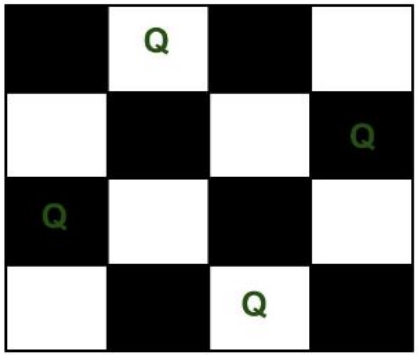
INTRODUCTION

- Important tool for solving problems with constraint satisfaction, such as crosswords, verbal arithmetic, Sudoku, and many other puzzles.
- There are three types of problems in backtracking –
  - **Decision Problem** – We search for a feasible solution.
  - **Optimization Problem** – We search for the best solution.
  - **Enumeration Problem** – We find all feasible solutions.
- It is based on Depth First recursive Search (DFS)
- Few advantages of Backtracking are :
  - Simple and easy to code.
  - Effective for constraint satisfactory problems.
  - Best option for solving tactical problems.

WORKING OF ALGORITHM

1. **Search :** Searches all possibilities to find solutions to the given computational problem.
2. **Check :** Check validity of solutions according to the given constraints.
3. **Elimination :** Eliminate invalid solutions.
4. **Repeat :** Repeat from step 1 until problem is solved and all constraints are satisfied.

REAL LIFE PROBLEM



PROBLEM STATEMENT :

To implement **N Queen problem.**

It is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other.

SOLUTION :

Time complexity of N Queen problem is **O(N!)**

Each time the branching factor decreases by 1 or more, but not much, hence the upper bound is of **O(N!).**

APPROACH TO PROBLEM

1. Start in the leftmost column.
2. If all queens are placed then,
   return true
3. Try all rows in the current column. Do following for every tried row :
   a. If the queen can be placed safely in this row then mark this [row, column] as part of the solution and recursively check if placing queen here leads to a solution.
   b. If placing the queen in [row, column] leads to a solution then,
      return true
   c. If placing queen doesn't lead to a solution then unmark this [row, column] **(Backtrack)** and go to step (a) to try other rows.
4. If all rows have been tried and nothing worked,
   return false to trigger backtracking.

GREEDY METHOD

INTRODUCTION

- Simple, intuitive algorithm that is used in optimization problems.
- Builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit at each stage without worrying about the global optimum solution.
- The result may be a good solution but not necessarily the best solution.
- Short sighted and Non-recoverable

INTRODUCTION

- Applications of greedy method:
  - Minimum Spanning Tree
  - Dijkstra’s algorithm for shortest paths from a single source
  - Huffman codes (data-compression codes)

- A problem must comprise these two components for a greedy algorithm to work:
  - It has optimal substructures.
  - It has a greedy property (hard to prove its correctness!).

WORKING OF ALGORITHM

1. **Candidate set:** A solution is created from this set.
2. **Feasibility :** Find a feasible condition which will help in solving your problem. → **Feasibility Function**
3. **Selection :** Search for locally optimum solutions that satisfy the feasible condition. → **Selection Function**
4. **Analyze :** Assign a value to the solution or the partial solution to the problem. → **Objective Function**
5. **Check :** Check if the complete solution to a problem is reached or not. → **Solution Function**

REAL LIFE PROBLEM

PROBLEM STATEMENT :

To implement **Fractional Knapsack problem**.

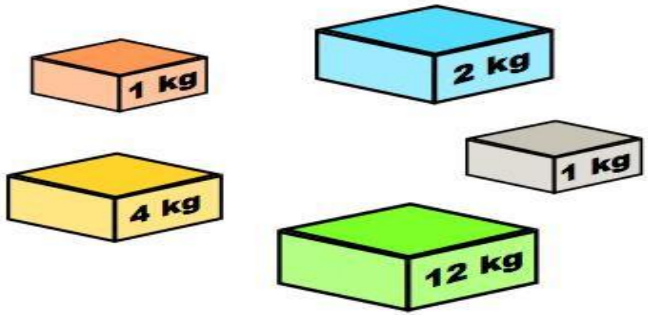
Given weights and values of n items, we need to put these items in a knapsack of capacity W to get the maximum total value in the knapsack.

SOLUTION :

Fractional Knapsack has time complexity **O(NlogN)** where N is the number of items in S. Assuming S is a **heap**-based priority queue and then the removal has complexity Θ(logN) so the up to N removals take O(NlogN). The rest of the algorithm is O(N)

APPROACH TO PROBLEM

1. Calculate the ratio (value/weight) for each item.
2. Sort the items based on this ratio.
3. Take the item with highest ratio and add them until we can’t.
4. Add the next item as whole.
5. At the end add the next item as much(fraction) as we can.



APPROACH TO PROBLEM

ALGORITHM : Greedy-Fractional-Knapsack (w[1..n], p[1..n], W)

```
for i = 1 to n
  do x[i] = 0
weight = 0
for i = 1 to n
  if weight + w[i] ≤ W then
    x[i] = 1
    weight = weight + w[i]
  else
    x[i] = (W - weight) / w[i]
    weight = W
    break
return x
```

DYNAMIC PROGRAMMING

INTRODUCTION

- Developed by Richard Bellman in the 1950s.
- It is a computer programming and mathematical optimization method.In both contexts it refers to simplifying a complicated problem by breaking it down into simpler sub-problems in a recursive manner.
- This technique is used for solving problems with overlapping subproblems.
- Uses bottom up approach and problem solving.

INTRODUCTION

- In terms of computer programming methods, there are two key attributes that a problem must have in order for dynamic programming to be applicable: optimal substructure and overlapping sub-problems.
- In terms of mathematical optimization, dynamic programming usually refers to simplifying a decision by breaking it down into a sequence of decision steps over time.
- Applications of Dynamic programming are :
  - Matrix Chain Multiplication
  - Longest Common Subsequence
  - Travelling Salesman Problem

WORKING OF ALGORITHM

1. **Characterize solution :** Characterize the structure of an optimal solution.
2. **Define :** Recursively define the value of an optimal solution.
3. **Computation :** Compute the value of an optimal solution, typically in a bottom-up fashion.
4. **Construction :** Construct an optimal solution from the computed information.

REAL LIFE PROBLEM

PROBLEM STATEMENT :

To solve **Traveling Salesman Problem.**

Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible tour that visits every city exactly once and returns to the starting point.

SOLUTION :

There are at most **O(n*(2^n))** subproblems, and each one takes linear time to solve. The total running time is therefore **O((n^2)*(2^n))**. The time complexity is much less than O(n!), but still exponential.

APPROACH TO PROBLEM

NAIVE SOLUTION :

1. Consider city 1 as the starting and ending point.
2. Generate all (n-1)! Permutations of cities.
3. Calculate cost of every permutation and keep track of minimum cost permutation.
4. Return the permutation with minimum cost.

BRANCH AND BOUND

INTRODUCTION

- It is an algorithm design paradigm used for solving discrete and combinatorial optimization problems.

- Combinatorial optimization problem is an optimization problem, where an optimum solution has to be identified from a finite set of solutions.

- Applicable when the greedy method and dynamic programming method tend to fail.

INTRODUCTION

- Typically exponential in terms of time complexity and may require exploring all possible permutations in worst case.
- Depends on efficient estimation of the lower and upper bounds of regions/branches of the search space.
- Uses state space tree for solving the problem (like Backtracking)

WORKING OF ALGORITHM

1. **Estimation :** Find the ratios of each element and sort.
2. **State Space Tree :** Implement a state space tree for the given problem with all the elements and its branches.
3. **Analyze :** Explore each of the branches of the tree, that represents the subset of the solution set.
4. **Check :** Before enumerating the candidate solutions of a branch, the branch is checked against upper and lower estimated bounds on the optimal solution, and is discarded if it cannot produce a better solution than the best one found so far by the algorithm.
5. **Repeat :** Repeat from Step 3 until the problem is solved, satisfying all its conditions and obtaining the best possible result.

REAL LIFE PROBLEM

PROBLEM STATEMENT :

To implement **0/1 Knapsack problem**.

Given two integer arrays **val[0..n-1]** and **wt[0..n-1]** that represent values and weights associated with n items respectively. Find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to Knapsack capacity W.

SOLUTION :

These problems typically exponential in terms of time complexity and may require exploring all possible permutations in worst case as there are **2^n** possible combinations of items are possible.

APPROACH TO PROBLEM

1. Sort all items in decreasing order of ratio of value per unit weight so that an upper bound can be computed using Greedy Approach.
2. Initialize maximum profit, maxProfit = 0
3. Create an empty queue, Q.
4. Create a dummy node of decision tree and enqueue it to Q. Profit and weight of dummy node are 0.

APPROACH TO PROBLEM

- Do following while Q is not empty.
  - Extract an item from Q. Let the extracted item be u.
  - Compute profit of next level node. If the profit is more than maxProfit, then update maxProfit.
  - Compute bound of next level node. If bound is more than maxProfit, then add next level node to Q.
  - Consider the case when next level node is not considered as part of solution and add a node to queue with level as next, but weight and profit without considering next level nodes.

THANK YOU !