Design and Analysis of Algorithms

L25: Intro to
Dynamic Programming

Dr. Ram P Rustagi
Sem IV (2020-Even)
Dept of CSE, KSIT
rprustagi@ksit.edu.in

Resources

- Text book 2: Horowitz
  - Sec 5.1,5.2,5.4,5.8,5.9
- Text book 1: Levitin
  - Sec 8.2-8.4
- R1: Introduction to Algorithms
    - Cormen et al.
- https://en.wikipedia.org/wiki/Dynamic_programming
- https://www.codechef.com/wiki/tutorial-dynamic-programming
- https://www.hackerearth.com/practice/algorithms/dynamic-programming/introduction-to-dynamic-programming-1/tutorial/

Dynamic Programming

- Quote :
  - Those who can not remember the past are condemned to repeat it.
- Approach of dynamic programming
  - Solve a complex problem by breaking it into set of sub-problems
    - Solving each sub-problem only once,
      - Storing the solution of subproblem
      - Use the result of solved subproblem when needed
    - Sub problems can be *overlapping*
- Applications using dynamic programming
  - Optimization solutions for problems

Example:Overlapping Subproblems

- Fibonacci number series

  $F_n = F_{n-1} + F_{n-2}$

- Computing F_{n-1} requires computing F_{n-2}
  - F_{n-2} is computed twice.
  - Hence it is called overlapping sub problem
  - Wasting compute power.
- If we can store F_{n-2} and use its value later
  - It is computed only once.
  - Corresponds to Dynamic Programming approach
- Dynamic programming works best for overlapping subproblems
  - Computation is done only once and stored/reused.

Dyn.Prog vs Divide-n-Conquer

- Divide and Conquer
  - Problem is divided into sub problems
  - Sub problems are solved, and then
  - Their results are combined to solve the main problem
  - Works better when sub problems do not overlap
  - Doesn't work when sub problems share subproblems
- Dynamic Programming
  - Results of sub problems are stored and used
  - Sub problems can share sub problems
  - Essentially, a sequence of decision making
    - Decision at step of sequence is stored (in table)

Example: Compute x^n

- Divide and conquer

```
pow(x,n)
   if n==0
      return 1
   if n==1
      return x
   if n is even
      return pow(x,n/2)*pow(x,n/2)
   else
      return pow(x,(n-1)/2)*pow(x,(n+1)/2)
```

- Time Complexity

   `T(n) = 2T(n/2)+1 = O(n)`

- Which are overlapping subproblems in this case?

Example: Compute x^n

- Dynamic Programming - Solution A

```
pow(x,n)
   res=1
   for i = 1 to n
      res = res * x
   return res
```

- Time Complexity
  - O(n)
- Solutions of which subproblems are stored and used?

Example: Compute x^n

- Dynamic Programming - Solution B

```
pow(x,n)
   if n==1
      return x
   if n is even
      y = pow(x,n/2)
      return y²
   else
      y=pow(x,(n-1)/2)
      return x*y²
```

- Time Complexity: $O(\log_2 n)$
- Solutions of which subproblems are stored and used?

Dyn.Prog vs Greedy Algos

- Greedy Algorithms
  - Choose first step (decision) as the most optimal one
  - Choose next step (decision) as the next best optimal
  - Continue the process to find next step till solution
  - Once a decision for a step is taken, it is not revisited.
  - Suitable only for class of problems where
    - Greedy approach works
  - Provides only 1 solution
- Dynamic Programming
  - Can provide multiple optimal solution

Greedy Algos: Examples

- Fractional knapsack problem
- Job scheduling problem
- Machine scheduling problem
- Minimum cost spanning tree problem
  - Prim's algorithm
  - Kruskal's algorithm
- Single source shortest path problem
  - Dijkstra's algorithm
- Note: we get only 1 solution
  - There may exist multiple optimal solutions.
  - e.g. Multiple Minimum Cost Spanning Trees?

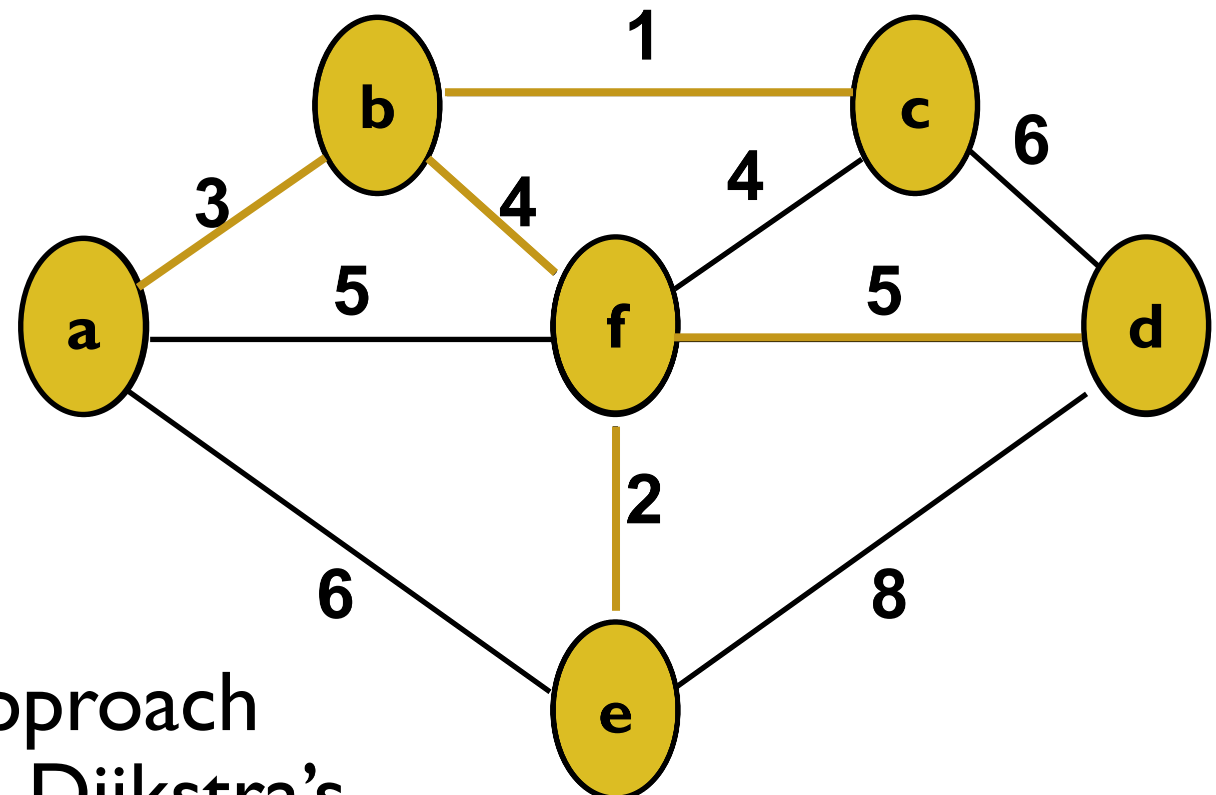
Dynamic Programming

- Problems solved with use of basic memorization
  - Fibonacci number series
  - Knapsack problems
  - Tower of Hanoi
  - Single source shortest path problems
  - All pair shortest path problems
  - Project scheduling
- Usage
  - Top down approach
  - Bottom up approach
  - Example: computation of `n!`
    - Bottom up approach: using iteration
    - Top down approach: using recursion

Greedy Approach vs Dyn.Prog

- Problem: Given directed graph, find the shortest path from node A to node B.
  - Let the path be $A, n_1, n_2, \ldots, n_k, B$.
  - Sequnece of decisions should involve only these nodes
  - No other nodes should be made part of this sequence and then later discarded.
- Q: Does greedy approach work?
  - It works for Single Source shortest paths to all nodes
  - Consider the example graph (next page)

Single Source Single Dstination



Q: Find shortest path between b and e?

- Consider any greedy approach
  - e.g. Prim's, Kruskal, Dijkstra's
  - Each will pick next node as c,
  - Node c is not on shortest path between b and e.
- Reason: Neighbors of b are a, f, c
  - By this local info, can't decide which one to pick up

Overlapping Sub-problems

- Overlapping subproblems
  - When a problem can be broken into subproblems
  - Subproblems are reused multiple times, e.g.
    - A recursive algorithm solves the same subproblem again and again e.g. `pow(x,n)`
    - Instead of generating new subproblems
- Example:
  - Computation of Fibonacci number F_n
    - Reuses F_{n-2} twice (1st directly, 2nd via F_{n-1})
      - This in turn invokes F_{n-3} 3 times, F_{n-4} 5 times

$$F_n \Rightarrow F_{n-1}+F_{n-2} \Rightarrow (\mathbf{F_{n-2}}+F_{n-3})+\mathbf{F_{n-2}} \Rightarrow (F_{n-2}+F_{n-3})+(F_{n-3}+F_{n-4})$$

$$\Rightarrow ((\mathbf{F_{n-3}}+F_{n-4})+\mathbf{F_{n-3}})+(\mathbf{F_{n-3}}+F_{n-4})$$

$$\Rightarrow (((\mathbf{F_{n-4}}+F_{n-5})+\mathbf{F_{n-4}})+(\mathbf{F_{n-4}}+F_{n-5}))+((\mathbf{F_{n-4}}+F_{n-5})+\mathbf{F_{n-4}})$$

Fibonacci Numbers

fib(6)

src https://www.hackerearth.com/practice/algorithms/dynamic-programming/introduction-to-dynamic-programming-1/tutorial/

Examples: Sequence of Decisions

- Knapsack problem (Greedy approach)
  - To decide values of x_i, $1 \le i \le n$
  - First decision is made on x_1
  - Then decision is made on x_2, x_3 and so on.
  - Optimal sequence of decisions maximizes
    - Objective function $\Sigma p_i x_i$ subject to constraint
      - $\Sigma p_i x_i \le m$, and $0 \le x_i \le 1$.
- Huffman Trees (or optimal merging of files)
  - First decision to merge two nodes with smallest weights (freq /file size) to make node with higher weight.
  - Repeat the process for next 2 smallest weights

Examples: Sequence of Decisions

- Shortest path from vertex v_i to vertex v_j in directed graph G
  - Which should be the second vertext?
  - Which should be the 3^{rd}, 4^{th} and subsequent vertices?
  - What is the sequence of optimal decisions, that
    - yields the path of shortest (cost) length.
  - Can we make step wise decision?
- Q: Can we always make step wise optimal decision?
  - There exists problems where decisions based on local information can not be made optimally.
- Does the work of step wise optimal decision making works for the problem of finding shortest path from node v_i to all other vertices of directed graph G?

Principle of Optimality

- Definition:

  The principle of optimality states that an optimal sequence of decisions has the property that whatever the initial state and the decision are, the remaining decisions must consitute an optimal decision sequence with regard to the state resulting from first decision.

- Essential difference between greedy method and dynamic programming.
  - In Greedy method, only one decision sequence is ever generated.
  - In Dynamic programming, many decision sequences may be generated.
    - Sequences containing sub-optimal subsequences can't be optimal, and thus can't be generated.

Optimality Principle: Shortest Path

- Shortest path problem:
  - Consider a shortest path from vertex v_i to vertex v_j in directed graph G
  - Assume the shortest path i.e. optimal path is
    $v_i, v_{i1}, v_{i2}, \ldots, v_j$.
  - After choosing first vertex v_{i1}, the problem becomes
  - Shortest path from v_{i1} to vertex v_j.
  - This shortest path must be
    $v_{i1}, v_{i2}, \ldots, v_j$.
  - If not, then let the shortest path be $v_{i1}, v_{r2}, v_{r3}, \ldots, v_j$
  - This implies the original shortest path must be
    $v_i, v_{i1}, v_{r2}, v_{r3}, \ldots, v_j$
  - This, contradicts our initial assumption.
  - Thus, the principle of optimality holds for this problem.

Optimality Principle: $0-1$ Knapsack

- $0-1$ Knapsack problem: KNAP$(1, n, m)$
  - Let KNAP(i, j, y) be the subproblem, i.e.
  - Maximize $\Sigma_{i \le k \le j}\ p_k x_k$,
  - Subject to $\Sigma_{i \le k \le j}\ w_k x_k \le y$, $x_k = 0$ or 1, $i \le k \le j$
- Let optimal sequence for $0/1$ values for $x_1, x_2, \ldots, x_n$ be $y_1, y_2, \ldots, y_n$
- If $y_1 = 0$, then optimal sequence for KNAP$(2, n, m)$ must be $y_2, y_3, \ldots, y_n$
  - If above is not optimal sequence for KNAP$(2, n, m)$, then $y_1, y_2, \ldots, y_n$ can't be optimal seq for KNAP $(1, n, m)$
- If $y_1 = 1$, then $y_2, \ldots, y_n$ must be optimal for KNAP$(2, n, m - w_1)$
  - if not, let some seq $z_2, \ldots, z_n$ is an optimal seq, then $\Sigma_{2 \le k \le j}\ w_k x_k \le y - w_1$, and $\Sigma_{2 \le k \le j}\ p_k z_k \ge \Sigma_{2 \le k \le j}\ p_k y_k$
  - Then, optimal solution is $y_1, z_2, \ldots, z_n$, a contradiction
- Thus, the principle of optimality works for this problem.

Principle of Optimality

- The principle of optimality is stated w.r.t. only initial state and decision.
  - This equally holds well for intermediate states and decisions.
- Example: Shortest path
  - Let v_k be the an intermediate vertex on a shortest path $v_i, v_{i1}, v_{i2}, v_k, v_{k+1} \ldots, v_j$ from v_i to v_j.
  - Then, path v_i, v_{i1}, v_{i2}, v_k must be the shortest path from v_i to v_k, and
  - path $v_k, v_{k+1} \ldots, v_j$ must be the shortest path from v_k to v_j

Principle of Optimality: $0-1$ Knapsack

- First decision: Let $g_j(y)$ be the value of optimal solution for $K_{NAP}(j+1,n,y)$
  - Thus, $g_0(m)$ is the optimal solution to $K_{NAP}(1,n,m)$
  - The possible solutions for x_1 are 0 and 1.
- From the principle of optimality,

$$g_0(m)=\max\{g_1(m),g_1(m-w_1)+p_1)\} \ldots\ldots\ldots\ldots (1)$$

- Let $y_1,y_2,\ldots,y_n$ be an optimal sol^n to $K_{NAP}(1,n,m)$
- Then, $\forall j \;\; 1\le j\le n,\; y_1,\ldots,y_j$ must be optimal solution to $K_{NAP}(1,j,\ \Sigma_{1\le i\le j}\ w_i x_i)$, and
  - $y_{j+1},\ldots,y_n$ must be optimal solution to $K_{NAP}(j+1,\ n,\ m-\Sigma_{1\le i\le j}\ w_i x_i)$, Thus

$$g_i(y)=\max\{g_{i+1}(y),g_{i+1}(y-w_{i+1})+p_{i+1})\} \ldots\ldots\ldots\ldots (2)$$

Approach in Dynamic Programming

- Top down approach (uses recursion)
  - Break down the problem into sub-problems
  - Start solving these subproblems
  - If the problem is already solved, return saved answer
  - If not solved, solve it, save the answer, return answer
    - Also known as **memorization**
- Bottom up approach (dynamic programming)
  - Analyze the problem to identify smallest sub problem
  - Start solving from the trivial subproblem
  - Move up towards the given problem
  - Guarantees that subproblems are solved before solving given problem.
  - May solve unneeded subproblems too, e.g. ${}^{n}C_{k}$

Common Dyn.Prog. Problems

- Floyd-Warshall algo: All pair shortest paths
- Transitive closure of a graph
- `0-1` knapsack problem
- Longest common subsequence
- Matrix Chain multiplication
  – Fibonacci number generation

Summary

- Dynamic programming concepts
  - Overlapping sub problems
- Comparison with Greedy approach
- Comparison with Divide and Conquer
- Principle of optimality.
- Examples of problems suitable for DP.