Lecture 16: Dynamic Programming I

How to Solve an Algorithms Problem (Review)

- Reduce to a problem you already know (use data structure or algorithm)

Search Data Structures	Sort Algorithms	Shortest Path Algorithms
Array	Insertion Sort	Breadth First Search
Sorted Array	Selection Sort	DAG Relaxation (DFS + Topo)
Linked List	Merge Sort	Dijkstra
Dynamic Array	Heap Sort	Bellman-Ford
Binary Heap	AVL Sort	Johnson
Binary Search Tree / AVL	Counting Sort	
Direct-Access Array	Radix Sort	
Hash Table		

- Design your own **recursive** algorithm
  - Recursive so constant-sized program can solve large input, analysis by induction
  - Recursive function calls are nodes in a graph, directed edge from $A \to B$ if A calls B
  - Dependency graph of recursive calls must be acyclic, classify based on shape

Class	Graph
Brute Force	Star
Decrease & Conquer	Chain
Divide & Conquer	Tree
Dynamic Programming	DAG

  - Hard part is thinking inductively to construct recurrence on subproblems
  - How to solve a problem recursively (**SR. BST**)
    1. Define **Subproblems**
    2. **Relate** Subproblems
    3. Identify **Base** Cases
    4. Compute **Solution** from Subproblems
    5. Analyze Running **Time**

Fibonacci

- Subproblems: the ith Fibonacci number $F(i)$
- Relate: $F(i) = F(i-1) + F(i-2)$
- Base cases: $F(0) = F(1) = 1$
- Solution: $F(n)$

```
def fib(n):
    if n < 2: return 1                    # base case
    return fib(n - 1) + fib(n - 2)        # recurrence
```

- Divide and conquer implies a tree of **recursive calls** (draw tree)
- $T(n) = T(n-1) + T(n-2) + O(1) > 2T(n-2),\ T(n) = \Omega(2^{n/2})$ exponential... :(
- Subproblem $F(k)$ computed more than once! ($F(n-k)$ times)

- Draw subproblem dependencies as a DAG
- To solve, either:
  - **Top down:** record subproblem solutions in a memo and reuse
  - **Bottom up:** solve subproblems in topological sort order
- For Fibonacci, n subproblems (vertices) and $2(n-1)$ dependencies (edges)
- Then time to compute is then $O(n)$

```
# recursive solution (top down)
F = {}                                    # memo
def fib(n):
    if n < 2: return 1                    # base case
    if n not in F:                        # check memo
        F[n] = fib(n - 1) + fib(n - 2)    # recurrence
    return F[n]
```

```
# iterative solution (bottom up)
F = {}                                    # memo
def fib(n):
    F[0], F[1] = 1, 1                     # base case
    for i in range(2, n + 1):             # topological sort order
        F[i] = F[i - 1] + F[i - 2]        # recurrence
    return F[n]
```

Dynamic Programming

- Weird name coined by Richard Bellman
  - Wanted government funding, needed cool name to disguise doing mathematics!
  - Updating (dynamic) a plan or schedule (program)
- Existence of recursive solution implies that subproblems are decomposable[1]
- Recursive algorithm implies a graph of computation
- Dynamic programming if subproblems dependencies **overlap** (form a DAG)
- "Recurse but reuse" (Top down: record and lookup subproblem solutions)
- "Careful brute force" (Bottom up: do each subproblem in order)

Dynamic Programming Steps (SR. BST)

1. Define **Subproblems** subproblem $x \in X$
   - Describe the meaning of a subproblem **in words**, in terms of parameters
   - Often subsets of input: prefixes, suffixes, contiguous subsequences
   - Often record partial state: add subproblems by incrementing some auxiliary variables
2. **Relate** Subproblems $x(i) = f(x(j), \ldots)$ for one or more $j < i$
   - State topological order to argue relations are acyclic and form a DAG
3. Identify **Base** Cases
   - State solutions for all reachable independent subproblems
4. Compute **Solution** from Subproblems
   - Compute subproblems via top-down memoized recursion or bottom-up
   - State how to compute solution from subproblems (possibly via parent pointers)
5. Analyze Running **Time**
   - $\sum_{x \in X} \text{work}(x)$, or if $\text{work}(x) = W$ for all $x \in X$, then $|X| \times W$

---

[1] This property often called **optimal substructure**. It is a property of recursion, not just dynamic programming

Single Source Shortest Paths Revisited

- Find shortest path weight from s to v for all $v \in V$
- Observation: Subsets of shortest paths are shortest paths
- Try to find a recursive solution in terms of subproblems!
- Attempt 1:
  1. **Subproblems**
     - Try $\boxed{x(v) = \delta(s, v)\text{, shortest path weight from } s \text{ to } v}$
  2. **Relate**
     - $x(v) = \min\{x(u) + w(u, v) \mid (u, v) \in E\}$
     - Dependency graph is the same as the original graph!
     - If graph had cycles, subproblem dependencies can have cycles... :(
     - For now, **assume graph is acyclic**
     - Then we can compute subproblems in a topological sort order!
  3. **Base**
     - $x(s) = \delta(s, s) = 0$
     - $x(v) = \infty$ for any other $v \neq s$ without incoming edges
  4. **Solution**
     - Compute subproblems via top-down memoized recursion or bottom-up
     - Solution is subproblem $x(v)$, for each $v \in V$
     - Can keep track of parent pointers to subproblem that minimized recurrence
  5. **Time**
     - # subproblems: $|V|$
     - Work for subproblem $x(v)$: $O(deg_{in}(v))$

$$\sum_{v \in V} O(deg_{in}(v)) = O(|V| + |E|)$$

- This is just DAG Relaxation! What if graph contains cycles? Next time!