Lecture 15: Recursive Algorithms

How to Solve an Algorithms Problem (Review)

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

<table>
<thead>
<tr>
<th>Search Data Structures</th>
<th>Sort Algorithms</th>
<th>Graph Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>Insertion Sort</td>
<td>Breadth First Search</td>
</tr>
<tr>
<td>Linked List</td>
<td>Selection Sort</td>
<td>DAG Relaxation (DFS + Topo)</td>
</tr>
<tr>
<td>Dynamic Array</td>
<td>Merge Sort</td>
<td>Dijkstra</td>
</tr>
<tr>
<td>Sorted Array</td>
<td>Counting Sort</td>
<td>Bellman-Ford</td>
</tr>
<tr>
<td>Direct-Access Array</td>
<td>Radix Sort</td>
<td>Johnson</td>
</tr>
<tr>
<td>Hash Table</td>
<td>AVL Sort</td>
<td></td>
</tr>
<tr>
<td>AVL Tree</td>
<td>Heap Sort</td>
<td></td>
</tr>
<tr>
<td>Binary Heap</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Design your own recursive algorithm
  - Constant-sized program to solve arbitrary input
  - Need looping or recursion, analyze by induction
  - Recursive function call: vertex in a graph, directed edge from $A \rightarrow B$ if $B$ calls $A$
  - Dependency graph of recursive calls must be acyclic (if can terminate)
  - Classify based on shape of graph

<table>
<thead>
<tr>
<th>Class</th>
<th>Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brute Force</td>
<td>Star</td>
</tr>
<tr>
<td>Decrease &amp; Conquer</td>
<td>Chain</td>
</tr>
<tr>
<td>Divide &amp; Conquer</td>
<td>Tree</td>
</tr>
<tr>
<td>Dynamic Programming</td>
<td>DAG</td>
</tr>
<tr>
<td>Greedy/Incremental</td>
<td>Subgraph</td>
</tr>
</tbody>
</table>

- Hard part is thinking inductively to construct recurrence on subproblems
- How to solve a problem recursively (SRT BOT)
  1. **Subproblem** definition
  2. **Relate** subproblem solutions recursively
  3. **Topological order** on subproblems ($\Rightarrow$ subproblem DAG)
  4. **Base** cases of relation
  5. **Original** problem solution via subproblem(s)
  6. **Time** analysis
Merge Sort in SRT BOT Framework

- Merge sorting an array $A$ of $n$ elements can be expressed in SRT BOT as follows:
  - Subproblems: $S(i, j) = \text{sorted array on elements of } A[i : j]$ for $0 \leq i \leq j \leq n$
  - Relation: $S(i, j) = \text{merge}(S(i, m), S(m, j))$ where $m = \lfloor(i + j)/2\rfloor$
  - Topo. order: Increasing $j - i$
  - Base cases: $S(i, i + 1) = [A[i]]$
  - Original: $S(0, n)$
  - Time: $T(n) = 2T(n/2) + O(n) = O(n \log n)$

- In this case, subproblem DAG is a tree (divide & conquer)

Fibonacci Numbers

- Suppose we want to compute the $n$th Fibonacci number $F_n$
- Subproblems: $F(i) = \text{the } i\text{th Fibonacci number } F_i$ for $i \in \{0, 1, \ldots, n\}$
- Relation: $F(i) = F(i - 1) + F(i - 2)$ (definition of Fibonacci numbers)
- Topo. order: Increasing $i$
- Base cases: $F(0) = 0, F(1) = 1$
- Original prob.: $F(n)$

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

- Divide and conquer implies a tree of recursive calls (draw tree)
- Time: $T(n) = T(n - 1) + T(n - 2) + O(1) > 2T(n - 2)$, $T(n) = \Omega(2^{n/2})$ exponential... :
- Subproblem $F(k)$ computed more than once! ($F(n - k)$ times)
- Can we avoid this waste?
Re-using Subproblem Solutions

- Draw subproblem dependencies as a DAG
- To solve, either:
  - Top down: record subproblem solutions in a memo and re-use (recursion + memoization)
  - Bottom up: solve subproblems in topological sort order (usually via loops)
- For Fibonacci, \( n + 1 \) subproblems (vertices) and \(< 2n\) dependencies (edges)
- Time to compute is then \( O(n) \) additions

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

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

- A subtlety is that Fibonacci numbers grow to \( \Theta(n) \) bits long, potentially \( \gg \) word size \( w \)
- Each addition costs \( O([n/w]) \) time
- So total cost is \( O(n[n/w]) = O(n + n^2/w) \) time
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 decomposable subproblems
- Recursive algorithm implies a graph of computation
- Dynamic programming if subproblem dependencies overlap (DAG, in-degree > 1)
- “Recurse but re-use” (Top down: record and lookup subproblem solutions)
- “Careful brute force” (Bottom up: do each subproblem in order)
- Often useful for counting/optimization problems: almost trivially correct recurrences

How to Solve a Problem Recursively (SRT BOT)

1. **Subproblem** definition subproblem \( x \in X \)
   - Describe the meaning of a subproblem in words, in terms of parameters
   - Often subsets of input: prefixes, suffixes, contiguous substrings of a sequence
   - Often record partial state: add subproblems by incrementing some auxiliary variables

2. **Relate** subproblem solutions recursively \( x(i) = f(x(j), \ldots) \) for one or more \( j < i \)

3. **Topological order** to argue relation is acyclic and subproblems form a DAG

4. **Base** cases
   - State solutions for all (reachable) independent subproblems where relation breaks down

5. **Original problem**
   - Show how to compute solution to original problem from solutions to subproblem(s)
   - Possibly use parent pointers to recover actual solution, not just objective function

6. **Time** analysis
   - \( \sum_{x \in X} \text{work}(x) \), or if \( \text{work}(x) = O(W) \) for all \( x \in X \), then \( |X| \cdot O(W) \)
   - \( \text{work}(x) \) measures nonrecursive work in relation; treat recursions as taking \( O(1) \) time

---

1This property often called **optimal substructure**. It is a property of recursion, not just dynamic programming
DAG Shortest Paths

- Recall the DAG SSSP problem: given a DAG $G$ and vertex $s$, compute $\delta(s, v)$ for all $v \in V$
- Subproblems: $\delta(s, v)$ for all $v \in V$
- Relation: $\delta(s, v) = \min \{ \delta(s, u) + w(u, v) \mid u \in \text{Adj}^- (v) \} \cup \{ \infty \}$
- Topo. order: Topological order of $G$
- Base cases: $\delta(s, s) = 0$
- Original: All subproblems
- Time: $\sum_{v \in V} O(1 + |\text{Adj}^- (v)|) = O(|V| + |E|)$
- DAG Relaxation computes the same min values as this dynamic program, just
  - step-by-step (if new value < min, update min via edge relaxation), and
  - from the perspective of $u$ and $\text{Adj}^+(u)$ instead of $v$ and $\text{Adj}^-(v)$

Bowling

- Given $n$ pins labeled 0, 1, $\ldots$, $n - 1$
- Pin $i$ has value $v_i$
- Ball of size similar to pin can hit either
  - 1 pin $i$, in which case we get $v_i$ points
  - 2 adjacent pins $i$ and $i + 1$, in which case we get $v_i \cdot v_{i+1}$ points
- Once a pin is hit, it can’t be hit again (removed)
- Problem: Throw zero or more balls to maximize total points
- Example: $[ -1, 1, 1, 1, 9, 9, 3, -3, -5, 2, 2 ]$
Bowling Algorithms

- Let’s start with a more familiar divide-and-conquer algorithm:
  - **Subproblems:** \( B(i, j) = \text{maximum score starting with just pins } i, i + 1, \ldots, j - 1, \) for \( 0 \leq i \leq j \leq n \)
  - **Relation:**
    * \( m = \lfloor (i + j)/2 \rfloor \)
    * Either hit \( m \) and \( m + 1 \) together, or don’t
    * \( B(i, j) = \max \{ v_m \cdot v_{m+1} + B(i, m) + B(m + 2, j), B(i, m + 1) + B(m + 1, j) \} \)
  - **Topo. order:** Increasing \( j - i \)
  - **Base cases:** \( B(i, i) = 0, B(i, i + 1) = \max \{ v_i, 0 \} \)
  - **Original:** \( B(0, n) \)
  - **Time:** \( T(n) = 4T(n/2) + O(1) = O(n^2) \)

- This algorithm works but isn’t very fast, and doesn’t generalize well (e.g., to allow for a bigger ball that hits three balls at once)

- **Dynamic programming algorithm:** use suffixes
  - **Subproblems:** \( B(i) = \text{maximum score starting with just pins } i, i + 1, \ldots, n - 1, \) for \( 0 \leq i \leq n \)
  - **Relation:**
    * Locally brute-force what could happen with first pin (original pin \( i \)):
      skip pin, hit one pin, hit two pins
    * Reduce to smaller suffix and recurse, either \( B(i + 1) \) or \( B(i + 2) \)
    * \( B(i) = \max \{ B(i + 1), v_i + B(i + 1), v_i \cdot v_{i+1} + B(i + 2) \} \)
  - **Topo. order:** Decreasing \( i \) (for \( i = n, n - 1, \ldots, 0 \))
  - **Base cases:** \( B(n) = B(n + 1) = 0 \)
  - **Original:** \( B(0) \)
  - **Time:** (assuming memoization)
    * \( \Theta(n) \) subproblems \( \cdot \Theta(1) \) work in each
    * \( \Theta(n) \) total time

- Fast and easy to generalize!
- Equivalent to maximum-weight path in Subproblem DAG:
Bowling Code

- Converting a SRT BOT specification into code is automatic/straightforward
- Here’s the result for the Bowling Dynamic Program above:

```python
# recursive solution (top down)
def bowl(v):
    memo = {}
    def B(i):
        if i >= len(v): return 0  # base cases
        if i not in memo:         # check memo
            memo[i] = max(B(i+1),   # relation: skip pin i
                           v[i] + B(i+1),  # OR bowl pin i separately
                           v[i] * v[i+1] + B(i+2))  # OR bowl pins i and i+1 together
        return memo[i]
    return B(0)  # original

# iterative solution (bottom up)
def bowl(v):
    B = {}
    B[len(v)] = 0  # base cases
    B[len(v)+1] = 0
    for i in reversed(range(len(v))):  # topological order
        B[i] = max(B[i+1],                  # relation: skip pin i
                    v[i] + B(i+1),      # OR bowl pin i separately
                    v[i] * v[i+1] + B(i+2))  # OR bowl pins i and i+1 together
    return B[0]  # original
```

How to Relate Subproblem Solutions

- The general approach we’re following to define a relation on subproblem solutions:
  - Identify a question about a subproblem solution that, if you knew the answer to, would reduce to “smaller” subproblem(s)
    * In case of bowling, the question is “how do we bowl the first couple of pins?”
  - Then locally brute-force the question by trying all possible answers, and taking the best
    * In case of bowling, we take the max because the problem asks to maximize
  - Alternatively, we can think of correctly guessing the answer to the question, and directly recursing; but then we actually check all possible guesses, and return the “best”

- The key for efficiency is for the question to have a small (polynomial) number of possible answers, so brute forcing is not too expensive
- Often (but not always) the nonrecursive work to compute the relation is equal to the number of answers we’re trying