Lecture 15: Recursive Algorithms

How to Solve an Algorithms Problem (Review)

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

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

- 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 $A \to B$ if A calls B
  - Dependency graph of recursive calls must be acyclic (if can terminate)
  - Classify based on shape of graph

Class	Graph
Brute Force	Star
Decrease & Conquer	Chain
Divide & Conquer	Tree
Dynamic Programming	DAG
Greedy/Incremental	Subgraph

  - 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

- **S**ubproblems: the ith Fibonacci number $F(i)$
- **R**elate: $F(i) = F(i-1) + F(i-2)$
- **B**ase cases: $F(0) = 0, F(i) = 1$
- **S**olution: $F(n)$

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

- Divide and conquer implies a tree of **recursive calls** (draw tree)
- **T**ime: $T(n) = T(n-1) + T(n-2) + O(1) > 2T(n-2)$, $T(n) = \Omega(2^{n/2})$ 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, $n+1$ subproblems (vertices) and $< 2n$ dependencies (edges)
- Then time to compute is then $O(n)$

```
# recursive solution (top down)
F = {}                                  # memo
def fib(n):
    if n < 2: return n                  # 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] = 0, 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 decomposable subproblems[1]
- Recursive algorithm implies a graph of computation
- Dynamic programming if subproblems dependencies **overlap** (DAG, in-degree > 1)
- "Recurse but reuse" (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

---

Solve a Problem Recursively (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**
   - $\sum_{x \in X} \mathrm{work}(x)$, or if $\mathrm{work}(x) = W$ 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

Max Subarray Sum

- **Problem:** Given an array A of n integers, what is the largest sum of any **nonempty** subarray? (in this class, **subarray** always means a contiguous sequence of elements)
- **Application:** You have a program for a long music festival marked with enjoyment of each act. You want to maximize total enjoyment, but can only enter or leave festival once.
- **Example:** `A = [-9, 1, -5, 4, 3, -6, 7, 8, -2]`, largest `subsum` is 16.
- Brute Force:
  - No relation between subproblems, # subarrays: $\binom{n}{2} + \binom{n}{1} = O(n^2)$
  - Can compute subarray sum of k elements in $O(k)$ time
  - n subarrays have 1 element, $n-1$ have 2, ... , 1 has n elements
  - Work is $c\sum_{k=1}^{n}(n-k+1)k = cn(n+1)(n+2)/6 = O(n^3)$
  - Graph: single node, or quadratic branching star, each with linear work

```python
def max_subsum(A):
    m = A[0]
    for j in range(1, len(A) + 1):
        for i in range(0, j):
            m = max(m, subsum(A, i, j))
    return m
```

```python
def subsum(A, i, j):
    s = 0
    for k in range(i, j):
        s += A[k]
    return s
.
```

- Divide & Conquer
  - **S**ubproblems: $m(A, i, j)$ is maximum subarray sum in subarray $A[i:j]$
  - **R**elate: Max subarray is either:
    1. fully in left half,
    2. fully in right half,
    3. or contains elements from both halves.
  - Third case: `max_subsum_from` middle plus `max_subsum_upto` middle

```python
def max_subsum_from(A, i, j):
    s = m = A[i]
    for k in range(1, j - i):
        s += A[i + k]
        m = max(s, m)
    return m
```

```python
def max_subsum_upto(A, i, j):
    s = m = A[j - 1]
    for k in range(1, j - i):
        s += A[j - 1 - k]
        m = max(s, m)
    return m
```

  - **B**ase case: only single item in subarray
  - **S**olution: call `max_subsum(A, 0, len(A))`
  - **T**ime: Binary tree with linear combine per vertex, $T(n) = 2\,T(n/2) + O(n) \implies T(n) = O(n \log n)$

```python
def max_subsum(A, i = 0, j = None):
    if j is None:  j = len(A)
    if j - i == 1: return A[i]     # base case
    c = (i + j) // 2
    return max(
        max_subsum(A, i, c),
        max_subsum(A, c, j),
        max_subsum_upto(A, i, c) + max_subsum_from(A, c, j)
    )
```

- Dynamic Programming
  - **S**ubproblems: $x(k) =$ `max_subsum_upto(A, 0, k)`, max subarray ending at $A[k]$
  - `max_subsum` ends somewhere, so check $x(k)$ for all k. (Brute Force)

```python
def max_subsum(A):
    m = A[0]
    for k in range(len(A)):
        s = max_subsum_upto(A, 0, k + 1)
        m = max(m, s)
    return m
```

  - But takes $c\sum_{k=1}^{n} k = cn(n+1)/2 = O(n^2)$ time.
  - Computing a lot of subarray sums; can we reuse any work?
  - **R**elate: Let's rewrite `max_subsum_upto` recursively

```python
def max_subsum_upto(A, i, j):
    if j - i == 1:  return A[i]    # base case
    return A[j - 1] + max(0, max_subsum_upto(A, i, j - 1))
```

  - **B**ase case: same as before
  - **S**olution: take maximum of subproblems
  - Graph of function calls is a tree with $O(n^2)$ nodes, same function called many times!
  - Redraw call graph as a DAG of overlapping problems.
  - **T**ime: Only $O(n)$ nodes, with only $O(1)$ work at each!
  - Dynamic programming: remember work done before, or compute from bottom up

```python
def max_subsum(A):
    m = mss_ut = A[0]
    for i in range(1, len(A)):
        mss_ut = A[i] + max(0, mss_ut)
        m = max(m, mss_ut)
    return m
```