Kruskal's, Dynamic Programming Intro, LCS

1332 Recitation: Week of July 14th

Kruskal's

- A greedy algorithm that finds an MST in a graph
- Only works for undirected graphs (because of the definition of MSTs)
- We will use a Set of edges to store and represent the MST
- Uses a Disjoint-Set data structure
    - Keeps track of a set of elements partitioned into disjoint sets
    - Each element is initially stored in its own set
    - Has two methods
        - Union fuses two sets together so that they become one set
        - Find returns an "id" for the set (if find(x) == find(y), x and y are in the same set)
    - We use a Disjoint-Set because it can perform these two methods very fast -> practically O(1)

Kruskal's in High-level

1. Add edges to the MST in sorted order by weights (which will be provided to us by a PQ, which initially contains **all** edges)
2. Skip an edge if adding it to the MST would create a cycle
3. Terminate when an MST is found (i.e. when the number of edges in the MST is 2(|V| - 1)

Kruskal's - Algorithm

- Initialize a Disjoint-Set of vertices using its constructor (note that each vertex starts in its own set)
- Build a Priority Queue of all edges (BuildHeap!)
- Until MST is found (set has 2(|V| - 1) edges) or PQ is empty:
  - Dequeue an edge (u,v)
  - If u and v are not in the same set and thus would not create a cycle (i.e. find(u) != find(v)):
    - Union the sets of u and v (i.e. union(u,v))
    - Add edge to MST
- Check validity of MST before returning (check if there are 2(|V|-1) edges in MST)
- Note that we check whether u and v are in the same set because if they were, adding their edge to the MST would create a cycle
- Example!
- Efficiency: O(|E|log|V|)
  - Comes from all the dequeues and all other parts of the algorithm are dominated by this

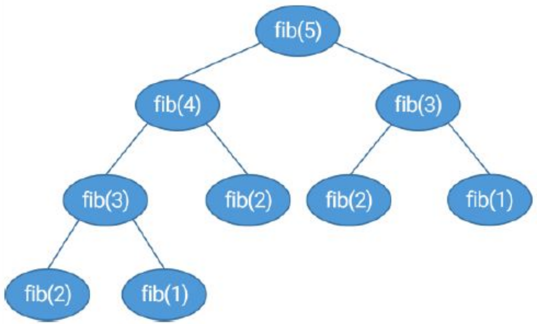
Dynamic Programming Intro

- A strategy for solving a class of problems that can be broken down into smaller repetitive problems that overlap
  - Break the problem into smaller subproblems
  - Solve subproblems and save their results as you solve them
  - If we encounter a subproblem again, just use the saved result
  - Build off the results of subproblems to solve larger instances of the problem
- Typically seen in the context of solving a problem recursively and storing results of recursive calls instead of recomputing
- Using the dynamic programming approach of storing results of sub-problems to avoid re-computation can reduce time complexities from exponential to polynomial time

Fibonacci - Recursive Approach

- The Fibonacci sequence is defined as:
  - fib(n) = fib(n-1) + fib(n-2)
  - fib(0) = 0, fib(1) = 1
  - So fib(2) = 1, fib(3) = 2
- Recursive approach!

```
public int fib(int n) {
    if (n == 0) return 0;
    else if (n == 1) return 1;
    return fib(n - 1) + fib(n - 2);
}
```



->does a lot of recomputation, exponential time complexity

->note how fib(3) and fib(2) are called multiple times and would return the same result

Fibonacci - Memoization Approach

- Instead of doing the recursion directly, we first check if we have already computed that case by checking our Map *memo*
  - If we have not yet computed it, we do so and store it in the Map for future
  - We then just return the value stored in the Map
- Note that many nodes in the previous tree will now become leaves and will be computed in constant time

```java
Map<Integer, Integer> memo;

public int fib(int n) {
    if (n == 0) return 0;
    else if (n == 1) return 1;
    if (!memo.containsKey(n)) {
        int result = fib(n - 1) + fib(n - 2);
        memo.put(n, result);
    }
    return memo.get(n);
}
```

Fibonacci - Tabulation is EVEN BETTER!

- The approach we'll go over is called "tabulation" -- when we solve a dynamic programming problem by using a table
- We use tabulation if you need to calculate answers to all subproblems, and order of calculation matters
- Tabulation is faster than memoization because it is iterative and requires no overhead space
- The code here can be further optimized to only use a length 2 array (or 2 variables), but this is easier to visualize

```
public int fib(int n) {
    int[] f = new int[n + 2];
    f[0] = 0;
    f[1] = 1;
    for (int k = 2; k <= n; k++) {
        f[k] = f[k - 1] + f[k - 2];
    }
    return f[n];
}
```

Longest Common Subsequence (LCS)

- Subsequences :
  - A subset of a string where each character appears in the same order as in the string, but that are not necessarily contiguous
  - Subsequence of BROWN: BRO, BON, BROWN, RWN, O
  - Not subsequences of BROWN: BOR, NWORB, XYZ, NW
- LCS Problem:
  - Given a string X with length n and a string Y of length m, what is the longest subsequence that appears in both strings?

LCS - Algorithm

- Bad! - brute force
  - Iterate over all subsequences in X and check if each subsequence exists in Y
  - Very slow: there are 2^n subsequences in X (each letter can be included or not)
- Good! - dynamic programming
  - Define a 2D array L to store solutions to the subproblems
  - 2D array will be (m+1) x (n+1) big (we add a row and column for empty strings)
  - Subproblem: L[i,j] will store the length of the LCS of X[0...i] and Y[0...j]
  - Set L[0,k] = 0 for all k < m + 1 and L[q,0] = 0 for all q < n + 1 -> LCS of any string and an empty string is an empty string
  - The value of each L[i,j] is:
    - If X[i] == Y[j] -> L[i-1, j-1] + 1 (extend the LCS of X[0...i-1] and Y[0...j-1]
    - If X[i] != Y[j] -> max(L[i-1, j], L[i, j-1]) (use the longest LCS of X[0...i-1] and Y[0..j] or X[0...i] and Y[0...j-1]
    - Both are O(1) operations, fill out (m*n) cells with O(1) operations -> O(mn)
- This algorithm only gives us the length of the LCS

LCS - to find the actual LCS

- Set i = n - 1, j = m - 1
- Let R be the result string
- If X[i] == Y[j], then R = R + X[i]; i--; j--;
- Else if X[i] != Y[j]:
  - If L[i-1,j] >= L[i,j-1], then i--;
  - Else j--
- Reverse R

LCS - Notes

- Multiple LCSs can be found if you traverse up vs. left when L[i-1, j] == L[i, j-1] while constructing the actual LCS
  - On exams, we will either point out which one to do or tell you to be consistent
- Instead of reversing the string to find the LCS, we could alternatively build the string by adding new chars to front