Algorithms

William Schultz

December 1, 2023

1 Graph Search

Can think about general graph search algorithm as consisting of an *explored* set of nodes and a *frontier* set of nodes. The goal is to eventually have the explored set equal to all nodes in the graph. The frontier is a set of nodes that we maintain along the way. Initially, we set the frontier to the starting node of the graph. It is unexplored but currently on our list of nodes that need to be explored i.e. it is on the frontier. We then pick a new node from the frontier set, mark it as explored, and do any other work we might need to do, and then take all of its neighbors and add them to the frontier set.

1.1 Depth-first search

Depth-first search searches deeper in a graph before searching broader. Can do a basic recursive or iterative implementation. Iterative implementation uses a stack to keep track of the frontier nodes, so that we explore deeper nodes first. We can also implement depth first search in a way that lets us recover paths to a node, by storing parent pointers as we go.

1.2 Breadth-first search

Breadth-first search searches all closer nodes before searching farther nodes i.e. it progresses in "levels" of depth. Not a standard way to implement it recursively, but can use a queue to keep track of the frontier nodes.

2 Dynamic Programming

There are 2 main components of a problem that make it amenable to a so-called "dynamic programming" (badly named) approach:

1. **Optimal Substructure**: A global solution can be described in terms of solutions to smaller "local" problems. In other words, it is possible to find a solution to a larger problem by solving smaller problems and combine them in an efficient way.

2. **Overlapping Subproblems**: The global problem can be broken down into smaller "local" sub-problems, and there is some overlap/redundancy between these subproblems, which is where you ideally get the efficiency speedup from.

Note that either (1) and (2), in isolation, don't necessarily permit an efficient, dynamic programming based approach to a problem. For example, we can consider *divide and conquer* type approaches as satisfying the *optimal substructure* property, but don't necessarily satisfy the *overlapping subproblems* property. For example, merge sort solves smaller subproblems (subsequences of an original list) and then merges them into a larger solution. But, in general, these smaller sorting problems cannot be expected to actually overlap at all.

Examples of problems with efficient DP approaches:

1. **Fibonacci**: Compute the n-th Fibonacci number.

2. **Subset Sum**: Given a set (multi-set) S of integers and a target sum k, determine if there is a subset of $X \subseteq S$ such that the sum of integers in X equals k.

3. **Knapsack**: Given a set of n items, each with a weight w_i and values v_i, find a subset of items that fits in a knapsack of capacity B and maximizes the overall value of included items.

4. **Weighted Interval Scheduling**: Given a set of intervals (s_i, e_i, W_i) represent as start and end times s_i and e_i, respectively, and weight W_i, determine the maximum weight set of non-overlapping intervals.

5. **Minimum Edit Distance**: Given two strings S and T over some alphabet of characters Σ, determine the minimum number of insertions or deletions needed to transform S to T.

6. **Matrix Chain Multiplication**: Given a sequence of matrices, determine the most efficient order in which to multiple them

Can also look at some problems as having solution that can be built by a sequence of choices of which elements to add to the solution. This also allows for a more unified view in some cases between a greedy approach and a DP approach. For example, in the *Subset Sum* problem, we can imagine a strategy where we build a solution by picking new elements from the original set to add to our output solution. We might take some kind of greedy approach where we, for example, pick the next smallest value and add it to our output. Clearly, the issue with the greedy approach in this problem is that it can get "stuck", with no next choices that allow the solution to be rectified, even if a solution does exist.

Subset Sum

To understand the idea behind approach for **Subset Sum**, can think about each element of the given list of n integers $S = \{a_1, \ldots, a_n\}$. If we wanted to come up with a naive recursive solution, we could imagine the decision tree for building all subsets of S, where each branch point represents whether we include that element or not in the subset. This is one way to simply generate all possible subsets of a given set. Within this tree, though, at each node, we can imagine we are dealing with a subset of the original set, based on the subset (e.g. suffix) of elements that we have not made a choice about including or excluding. Along with this, we can imagine that each node of the tree also has associated with it the "used up" amount, which is the sum of elements chosen to include based on the path to this position in the tree. Now, even though this tree naively has size (i.e, width) exponential in the number of elements, there are actually a limited number of unique problems to solve in this tree, so there is sufficient overlap between them to make this efficient.

Basically, if our target sum is T, then there are at most T unique "used up" values that can appear at any node in this tree. And, there are most n unique suffixes that can appear as well.

Knapsack

0-1 Knapsack is very similar to Subset Sum i.e., we have to find a subset of n given items that remains under our given capacity and maximizes the sum of the chosen item values. As in **Subset Sum** case, we can imagine solution search tree where we either include or exclude the first element, and the subproblems we recurse on are basically the rest of the elements with a capacity reduced by that of our first element, or the rest of the elements with the original capacity. Again, this tree might grow exponentially, but, if our capacity is C, we actually only have at most C unique possible capacities, and at most n suffixes of elements. Note also that the minor difference from Subset Sum is that, when we combine solutions to recursive subproblems, we want to take the maximum solution (since this is an optimization problem variant), rather than just taking the disjunction.

Algorithms Problem List

Merge Strings Alternately

- **Problem**: Given two strings s_1 and s_2, merge them into one string S such that the output string S interleaves the characters of s_1 and s_2 alternately. If one string is longer than the other, then we append the remaining characters of that string to the end of the output string.

Greatest Common Divisor of Strings

- **Problem**: Given two strings s and t, we say that t "divides" s if $s = t + t + \cdots + t$. That is, s consists of t concateneated with itself 1 or more times. Given two strings s_1 and s_2, we want to find the greatest common divisor x between s_1 and s_2. That is, the largest string x such that x divides s_1 and s_2.

Kids with Greatest Number of Candies

- **Problem**: Given array of kids with some number of candies, and number of extra candides, compute whehter each kid would have max candies after receiving the number of extra candies you have.

Merge k Sorted Lists

- **Problem**: Given a set of k linked lists, each which are individually sorted in ascending order, merge all k lists into one sorted list.
- **Solution Idea**: The basic approach is to essentially just perform the *merge* step of merge-sort. That is, if we are given a set of already sorted lists, we can merge them all into one sorted lists by repeatedly popping the smallest element from the remaining, non-empty lists and appending it to the output list.
- **Key Concepts**:
  - ***Mergesort Merging***
  - ***Linked List Manipulation***

  The essence of the solution is very straightforward as long as you know and understand the ideas behind mergesort i.e. knowing the core idea that you can merge a set of already sorted lists by incrementally choosing the smallest element from each.

Remove duplicates from sorted linked list

- **Problem**: Given a sorted linked list, remove any duplicates from the list.
- **Solution Idea**: Iterate through the linked list, but at each node look ahead to see how many nodes in front of you contain an identical value to your own. Update your current "next" pointer to point to the first node after this block of identical nodes in front of you. Since the list is sorted, you know that any duplicates of the current value must be directly in front of you.
- **Key Concepts**:
  - ***Linked List Traversal***
  - ***Linked List Deletion***
  - ***Duplicate Detection by Sorting***

  The underlying insight in the solution is to recognize that sorting a list can be used an easy mechanism for detecting duplicates. That is, in a sorted list, all duplicates of a particular item will always appear in contiguous "blocks". Once you recognize this fact, then implementing the solution mostly requires a standard application of linked list iteration and linked list item deletion. Namely, that to delete an item n_2 from a linked list that appears in a list as $n1 \rightarrow n_2 \rightarrow n_3$, you simply update the "next" pointer of n_1 to point to n_3 instead of n_2. Recall that a basic linked list node is a $LinkedListNode(val, next)$ structure, where val is the value of that node, and $next$ is a pointer to the next item in the list.

Intersection of two linked lists

- **Problem**: Given two singly linked lists, return the node at which the two lists intersect. If they have no intersection, then return *null*.
- **Solution Idea**: This is similar to a *lowest common ancestor* problem. One approach is to walk backwards to the root from one of the lists and keep track of all nodes seen along the way. Then, walk backwards from the other list and check for the first node you hit that was already seen, and that node should be the intersection point. Note that this uses $O(n)$ space, if n the upper bound on the size of the linked lists.

  It's also possible to do it without using any extra space by using a cleverer 2 pointer approach with a bit of counting. If we walk back to the root in both lists we can record the longer of the two. Then, from this we know the difference in length between the two lists, $diff$. So, we can walk backwards by $diff$ pointers in the longer list, and then walk forwards from there in both lists at the same time, until we hit a point where both pointers are pointing to the same node.
- **Key Concepts**:
  - ***Linked List Traversal***
  - ***Lowest Common Ancestor (?)***

Reverse linked list

- **Problem**: Given a singly linked list, reverse the list.
- **Solution Idea**: Iterate over the list and at each node, re-arrange the *next* pointer so it now points to the previous node rather than the next node.

  $$None \to a \to b \to c$$

  If $curr = a$ and $curr.next = b$, then to do the reversal we want to end up with $a.next = None$ and then step forward, ending up with $curr = b$. So, at each step of the traversal, we keep track of hte previous item we looked at, so that we can reverse the pointer of the current node to point to it. We also need to save a reference to the next node before we update it.
- **Key Concepts**:
  - ***Linked List Traversal***
  - ***Pointer Swapping (?)***

  Need to have a solid grasp of how to traverse a linked list, but also need to have good confidence in how to update points in a few steps (similar to how we swap variables), without overwriting the info we need to continue.

Add two binary strings

Subsets of a list