Algorithms

William Schultz

November 27, 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.

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.

  $$None \rightarrow a \rightarrow b \rightarrow c$$

  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