CS 61B Spring 2018

Final Exam Guerrilla Section

May 5, 2018

Instructions

Form a small group. Start on the first problem. Check off with a helper or discuss your *solution process* with another group once everyone understands *how to solve* the first problem and then repeat for the second problem ...

You may not move to the next problem until you check off or discuss with another group and *everyone understands why the solution is what it is*. You may use any course resources at your disposal: the purpose of this review session is to have everyone learning together as a group.

1 The Tortoise or the Hare

1.1 Given an undirected graph $G = (V, E)$ and an edge $e = (s, t)$ in G, describe an $O(|V| + |E|)$ time algorithm to determine whether G has a cycle containing e.

Remove e from the graph. Then, run DFS or BFS starting at s to find t. If a path already exists from s to t, then adding e would create a cycle.

1.2 Given a connected, undirected, and weighted graph, describe an algorithm to construct a set with as few edges as possible such that if those edges were removed, there would be no cycles in the remaining graph. Additionally, choose edges such that the sum of the weights of the edges you remove is minimized. This algorithm must be as fast as possible.

1. Negate all edges.
2. Form an MST via Kruskal's or Prim's algorithm.
3. Return the set of all edges not in the MST (undo negation). All edges that are not in the MST must produce a cycle. Think about why this is true. A spanning tree spans the entire graph: every node has a path to every other node. By adding any more edges to the graph, we MUST create a cycle.

   We negate the edges because Kruskal's and Prim's traditionally discards the largest edges, since we want the MINIMUM tree. You can see this through the cut property. Thus, if we want minimum sum of removed edges and still use the same Kruskal's algorithm, then we need to make the old largest edges the smalelst ones. We do this by negating.

1.3 Given an undirected, positively weighted graph $G = (V, E)$, a set of start vertices S, and a set of end vertices T, describe an efficient algorithm that returns the shortest path starting from any one vertex in S and ending at any one vertex in T.

Hint: Consider adding dummy nodes to the graph to reduce this problem into something more familiar.

How to approach: We almost ALWAYS use the algorithm as a black box. We don't try to alter the algorithm, but rather the data structure in such a way that we can use the original algorithm on it. We know that Dijkstra's finds the shortest path, so that's probably what we should use. The only problem is, Dijkstra's only has one source node and one destination node... So how do we alter this graph so that we can treat all the start vertices as one node and all the end vertices as another node?

Add a *source* node, s, connected to all start nodes, and a *sink* node, t, connected to all end nodes with weight w (any non-negative constant). Find the shortest path from s to t and then remove s and t from the final solution. This can be done with Dijkstra's algorithm.

2 One Traversal to Rule Them All

2.1 Given a directed, acyclic graph, G, implement `onePath` which determines if G contains a path that visits every vertex exactly once. Briefly justify why the algorithm is correct and state the runtime.

Assume that the graph is implemented with the following API, where nodes are represented by integers.

```
public class Graph {
        /** Returns true if this graph has an edge from u to v. */
        public boolean hasEdge(int u, int v);

        /** Returns a list of vertices, in topological order. */
        public List<Integer> topologicalOrder();
}
```

```
public boolean onePath(Graph g) {
        List<Integer> sorted = g.topologicalOrder();
        for (int i = 0; i < sorted.length - 2; i++) {
                if (!g.hasEdge(sorted[i], sorted[i + 1]) {
                        return false;
                }
        }
        return true;
}
```

The main idea is that if a DAG has a path that traverses every vertex, then its topological sorting must be unique (The path is a straight line), and additionally, every pair of nodes in the topological sorting must have an edge between them. Essentially, the topoligical sort IS the path. $\Theta(|V|+|E|)$, the runtime of topological sorting.

3 Comparison Sorting

3.1 For each of the following scenarios, choose the best sorting algorithm to use and explain your reasoning.

(a) Given a list that was created by taking a sorted list and swapping N pairs of adjacent elements.

Insertion sort, since a list created in such a manner will have at most N inversions. Recall that insertion sort runs in $\Theta(N + K)$ time, where K is the number of inversions. An inversion is any two elements that are in the wrong place relative to one another.

(b) Given that the list must be sorted in-place on a machine where swapping two elements is much more costly than comparing two elements.

Selection sort since, in its most common implementation, selection sort performs N swaps in the worst case whereas all other common sorts perform $\Omega(N \log N)$ swaps.

(c) Given the list is so large that not all of the data will fit into memory at once. As is, at any given time most of the list must be stored in external memory (on disk), where accessing it is extremely slow.

Merge sort is ideal here, since its divide-and-conquer strategy works well with the restriction of only being able to hold a partition of the list in memory at any given time. Sorted runs of the list can be merged in memory and flushed to disk one block at a time, minimizing disk reads and writes.

(d) Given a list of emails ordered by send time, sort the list such the emails are ordered by the sender's name first while secondarily maintaining the time-ordering.

Merge sort. We should avoid quicksort since the usual, in-place implementation (Hoare Partitioning) is unstable and can change the time-ordering when sorting.

(e) Given a randomly shuffled list of Java integers.

Quicksort has better real-world characteristics (can be done in place and has a pretty good average case runtime) than merge sort in this situation and we don't need to consider stability.

(f) Suppose you're designing a secure system that needs to defend against *adversarial inputs*. An attacker can give you any list they choose and understand exactly how your sorting algorithms are implemented.

Since the enemy knows exactly how your sort works, they can purposefully give you an input that has a bad runtime. Merge sort or heap sort have worst-case time complexity in $\Theta(N \log N)$, which is pretty much the best you can do. Quicksort with random pivots would also be acceptable since it's highly improbable to repeatedly select poor pivots on large inputs. Selection sort, or insertion sort would not be acceptable here because of their worst-case $\Theta(N^2)$ runtime which can be controlled by the input list.

3.2 Which sorting algorithms do each of the following illustrate? Your options are merge sort, insertion sort, selection sort, heap sort, and quick sort.

Algorithms illustrated may not conform exactly to those presented in discussion and in lecture. Please note that each of these are snapshots as the algorithm runs, not all iterations of its running.

(a)
```
5103 9914 0608 3715 6035 2261 9797 7188 1163 4411
0608 1163 5103 3715 6035 2261 9797 7188 9914 4411
0608 1163 2261 3715 6035 5103 9797 7188 9914 4411
```

Selection sort. First, we see that this is a partitioning (A term I kinda make up that just means the array is partitioned into sorted and unsorted portions) sort. Once we know it is a partitioning sort, we just need to choose between insertion sort and selection sort. In this case, all the smallest values are on the left, which means it is selection sort.

(b)
```
5103 9797 0608 3715 6035 2261 9914 7188 1163 4411
0608 3715 2261 1163 4411 5103 9797 6035 9914 7188
0608 3715 2261 1163 4411 5103 6035 7188 9797 9914
```

Quicksort. Using the first element as the pivot, the array is partitioned into values greater than and less than 5103. Then, within those halves, we use the first element once again as the pivot.

(c)
```
dze ccf hwy pjk bkw xce aux qtr
ccf dze hwy pjk bkw xce aux qtr
ccf dze hwy pjk aux bkw qtr xce
aux bkw ccf dze hwy pjk qtr xce
```

Merge sort. If we look at the array in halves and fourths, then we see that first, each fourth is relatively sorted. Then, each half. Then, the whole thing.

(d)
```
dze ccf bkw hwy pjk xce aux qtr xpa atm
dze ccf bkw hwy pjk xce aux qtr atm xpa
dze ccf bkw hwy pjk xce aux atm qtr xpa
dze ccf bkw hwy pjk xce atm aux qtr xpa
dze ccf bkw hwy pjk atm aux qtr xce xpa
dze ccf bkw hwy atm aux pjk qtr xce xpa
dze ccf bkw atm aux hwy pjk qtr xce xpa
dze ccf atm aux bkw hwy pjk qtr xce xpa
dze atm aux bkw ccf hwy pjk qtr xce xpa
atm aux bkw ccf dze hwy pjk qtr xce xpa
```

Insertion sort (from the right). This one is kind of tricky, since it starts on the right. But we see that in the first row, its unsorted. Then, in the second row, the last two items are relatively sorted in relation to each other. Next, the last 3 elements are relatively sorted in relation to each other. So on and so forth.

4 Potpourri

4.1 For each of the following, give a data structure or algorithm that you could use to solve the problem or write **impossible** if it is impossible to meet the running time given in the question. For each question, we have one or more right answers in mind, all of which are among the data structures and algorithms listed below. Provide a brief description of how each algorithm or data structure can be used to solve the problem and what, if any, modifications are necessary.

- DFS
- BFS
- Dijkstra's algorithm
- Topological sort
- Kruskal's algorithm
- Linked lists
- Balanced search trees
- Heaps
- Hash Tables
- Tries
- Insertion sort
- Selection sort
- Quicksort
- Merge sort
- Radix sort

(a) Given a list of N words with K characters each, find for each word its longest prefix that is the prefix of some other word in the list in worst-case $O(NK)$ character comparisons.

- Trie: construct a trie of the words and then find the longest prefix for each word by finding the parent of each special end node for a word (or last node with more than one child along path to end of word).
- Radix sort all the words and then compare each word with the one sorted before and after it to find the longest prefix.

(b) Given an undirected, weighted, and connected graph $G = (V, E)$, find the heaviest edge that can be removed without disconnecting the graph in worst-case $O(|E| \log |E|)$ time.

Run Kruskal's to find a minimum spanning tree and then take the heaviest edge not in that minimum spanning tree.

(c) We would like to build a data structure that supports the following operations: `add`, `remove`, and `find` the k^{th} largest element for any k in worst-case $O(\log N)$ comparisons for each operation (where N is the number of elements currently in the data structure).

Use a balanced search tree and additionally store the number of nodes in the subtree for each node.

(d) Given an unordered list of N `Comparable` objects, construct a binary search tree containing all of them in $O(N)$ comparisons.

This is impossible, because this would be tantamount to performing a comparison sort in linear time.

4.2 Given a list of N input words all of length at most K, and M query words, we would like to find, for each query word, the number of input words that match the first $\frac{K}{2}$ letters of the query word. Describe an algorithm that accomplishes this and give its running time as a function of N, K, and M.

- Use a trie where every node of the trie tracks how many of its descendants are complete words. Insert all N input words into the trie. This takes $O(NK)$ time (assuming a constant alphabet size). Then for each query word, find the node in the trie corresponding to the word's first $\frac{K}{2}$ letters and output the number stored in that node (and output 0 if there is no such node). This takes $O(K)$ time for each query word. So the entire algorithm takes $O((N+M)k)$ time. Note that in the best case, all of the query words begin with some letter that is not in the trie at all, in which case this solution will run in $O(NK+M)$ time.
- Use a `HashMap` where the keys are strings of length $\frac{K}{2}$ and the values are integers representing how many of the input words begin with those $\frac{K}{2}$ letters. For each input word w, look up the length $\frac{K}{2}$ prefix of w in the `HashMap`. If it is present, increment the corresponding value. If it is not present, add it as a key with corresponding value of 1. Then for each query word, check if its length $\frac{K}{2}$ prefix is present in the `HashMap`. If so, output the corresponding value. If not, output 0. In the worst case, inserting the input words into the `HashMap` takes $\Theta(NK)$ because in the worst case all input words hash to the same bucket and comparing the inserted word to all previously inserted words takes up to $O(NK)$ time because we have to compare the first $\frac{K}{2}$ letters of each word. Similarly, looking up all the query words takes $\Theta(MNK)$ time. So, in the worst case, this solution will take $\Theta((N+M)NK)$ time. If we assume, however, that the keys are distributed uniformly among the buckets then this solution is also $\Theta((N+M)K)$ time.