CSM 61B — Sorting and Trees

Fall 2020 — Mentoring 12: November 9th, 2020

1 Quicksort Pivot Choice

1.1 For each pivot selection strategy below, what is the best, average and worst case runtime?

The best case and average case runtime for quicksort in each scenario is still in $\Theta(N \log N)$.

(a) Always choose the first value in the list.

If we always select the first value as a pivot, the values in the list will determine the runtime. If we have a sorted or reverse sorted array, selecting the first value will only reduce the problem size by a single value during each call: we essentially have selection sort! The runtime in this worst-case scenario is $\Theta(N^2)$.

But if the values in the list are randomly shuffled, then choosing the first value not degrade into the worst-case runtime.

(b) Always find and choose the median value in the list. Assume finding the median takes $O(N)$ time where N is the length of the list.

The worst case runtime in this scenario will be in $\Theta(N \log N)$ including the time to find the median (which can be found in linear time using the median of medians algorithm). In reality, the additional cost associated with finding the median is usually not worth it over simply selecting a random value.

Even though median-finding takes linear time, quicksort normally needs to spend linear time bucketing values into less-than, equal-to, and greater-than buckets anyways so the asymptotic runtime is not affected.

(c) Always choose a random pivot.

It is possible for this to degrade to $\Theta(N^2)$ runtime in the worst case as well. If we pick a random pivot from an array we have a $\frac{1}{N}$ probability of selecting the smallest value and, after that, a $\frac{1}{N-1}$ of selecting the next smallest, and so forth. The total probability of selecting the smallest value every time is $\frac{1}{N!}$.

Although it is possible for for quicksort to run in $\Theta(N^2)$, it becomes less and less probable as N, or the size of the list, increases.

2 QuickSort vs. Merge Sort

2.1 (a) What are the advantages and disadvantages of quicksort?

Advantages:

- Faster on average by a constant factor than merge sort
- In-place variant for $\Theta(1)$ memory complexity. Including the average-case call stack brings up space complexity to $\Theta(\log N)$
- Multi-pivot quicksort offers greater constant factor optimizations

Disadvantages:

- Most efficient implementations (in-place) are not stable
- Worst-case runtime is in $\Theta(N^2)$, thus making it a poor choice for adversarial datasets
- Not very good for external sorting

(b) What are the advantages and disadvantages of merge sort?

Advantages:

- Stable
- Good for external sorting
- Constant space usage for sorting linked lists

Disadvantages:

- Slower than quicksort on average
- Higher space complexity for array allocation

3 Maximal Spanning Trees

3.1 We have two algorithms, Kruskal's and Prim's, that allow us to find a Minimum Spanning Tree. Consider the problem of finding a Maximum Spanning Tree

(a) Describe a modification to Kruskal's algorithm that would allow us to find a Maximum Spanning Tree of a graph

Negate all the edge weights and find the minimum spanning tree using Kruskal's algorithm. Nothing in Kruskal's algorithm assumes the weights are positive. Therefore, the minimum of the weights negated, is achieved by the maximum of the original weights, and we will have a Maximum Spanning Tree.
Similar logic applies for using Prim's algorithm with negated edge weights.

(b) Can we use a similar approach to modify Djikstra's algorithm to find the Maximum Path between two nodes?

No, because Djikstra's doesn't work with negative edge weights. This is because Djikstra's relies on the assumption that if all weights are non-negative, adding an edge can never make a path shorter. Therefore, we cannot simply negate edge weights and use Djikstra's to find the Maximum path.

4 Radix Sort

Algorithm	Best-case	Worst-case	Stable
Counting Sort	$\Theta(N + R)$	$\Theta(N + R)$	Yes
LSD Radix Sort	$\Theta(W(N + R))$	$\Theta(W(N + R))$	Yes
MSD Radix Sort	$\Theta(N + R)$	$\Theta(W(N + R))$	Yes

Where N is the length of the list, R is the size of the alphabet (radix), and W is the length of the longest word. Extra: MSD radix sort is stable when implemented with additional space for a buffer.

4.1 Run MSD and LSD radix sort on the following DNA sequence such that the output is sorted in alphabetical order ($A < C < G < T$).

Most-Significant Digit

```
ACAG ACAG ACAG ACAG ACAA
CTAG ACAA ACAA ACAA ACAG
ACAA CTAG CCTC CCTC CCTC
TGAG CCTC CTAG CTAG CTAG
CCTC GAGT GAGT GAGT GAGT
GAGT TGAG TGAG TGAG TGAG
```

Least-Significant Digit

```
ACAG ACAA ACAA GAGT ACAA
CTAG CCTC ACAG ACAA ACAG
ACAA ACAG CTAG ACAG CCTC
TGAG CTAG TGAG CCTC CTAG
CCTC TGAG GAGT TGAG GAGT
GAGT GAGT CCTC CTAG TGAG
```

4.2 Performing Radix Sort seems to be a fast sorting algorithm. Why don't we always use it?

Radix sort is only possible when the elements being compared have some radix or base. For strings, they can be broken up into individual characters and separately compared, as we did above. We can also do the same for integers. However, what if we wanted to compare 10 `Cat` objects? We cannot compare `Cat` objects by breaking them up into any smaller pieces or radices.

5 Graph Usage

5.1 You are the king of a large kingdom! In order to manage your kingdom, you have appointed lords to rule towns within your kingdom. Every lord can govern over his town and any town that he is connected to by road. Your job as king is to figure out the optimal way to allocate lords and build roads.

Formally, consider a graph G with vertices V and edges E. Each vertex v represents a town. It has an associated cost c, the cost of installing a lord in the town. Each edge e represents a potential road. It has an edge weight w, the cost of building that road. Devise an algorithm that can efficiently compute which towns to install lords in and which roads to build, such that every town in the kingdom is governed (either has a lord in it or is connected by some number of roads to a town with a lord in it).

We can formulate this problem as a Minimum Spanning Tree problem. We create a dummy node S. We connect S to every vertex with edge weight c, the cost of that vertex. We then find the MST of this modified graph, and that is the solution. Why does this method work? We know that the MST must include S, by definition of spanning tree. Every edge in te MST that is outgoing from S represents a selected town. In the MST, every vertex is either directly connected to S, i.e. a town with a lord, or connected to S by a series of selected edges, i.e. connected to some town with a lord via some roads. Since the MST finds the minimum cost solution, this is our desired arrangement of lords and roads