CS 61B Spring 2018

More Sorting

Discussion 13: April 17, 2018

1 Quicksort

1.1 Sort the following unordered list using stable quicksort. Assume that the pivot you use is always the first element and that we use the 3-way merge partitioning process described in lecture and lab last week. Show the steps taken at each partitioning step.

```
18, 7, 22, 34, 99, 18, 11, 4
```

```
-18-, 7, 22, 34, 99, 18, 11, 4
-7-, 11, 4 | 18, 18 | 22, 34, 99
4, 7, 11, 18, 18 | -22-, 34, 99
4, 7, 11, 18, 18, 22 | -34-, 99
4, 7, 11, 18, 18, 22, 34, 99
```

1.2 What is the worst case running time of quicksort? Give an example of a list that meets this worst case running time.

$\Theta(n^2)$. Running quicksort on a sorted list will take $\Theta(n^2)$ if the pivot chosen is always the first or last in the subarray. In general, the worst case is such that the partitioning scheme repeatedly partitionins an array into one element and the rest. At each level of recursion, you will need to do $\Theta(n)$ work, and there will be $\Theta(n)$ levels of recursion. This sums up to $1 + 2 + \cdots + n$.

1.3 What is the best case running time of quicksort? Briefly justify why you can't do any better than this best case running time.

$\Theta(n \log n)$. The optimal case for quicksort occurs if you can choose a pivot such that the left partition and right partition are of equal sizes. At each level of recursion, you will need to do $\Theta(n)$ work, and there will be $\Omega(\log n)$ levels of recursion. You can't do any better than this best case runtime because that would violate the sorting lower bound from lecture (for comparison based sorts).

1.4 What are two techniques that can be used to reduce the probability of quicksort taking the worst case running time?

1. Randomly choose pivots.
2. Shuffle the list before running quicksort.

2 Comparing Sorting Algorithms

2.1 When choosing an appropriate algorithm, there are often several trade-offs that we need to consider. For the following sorting algorithms, give the expected space complexity and time complexity, as well as whether or not each sort is stable.

	Time Complexity	Space Complexity	Stability
Insertion Sort	$\Theta(n^2)$	1	Yes
Heapsort	$\Theta(n \log n)$	1	No
Mergesort	$\Theta(n \log n)$	$\Theta(n)$	Yes
Quicksort	$\Theta(n \log n)$	$\Theta(\log n)$	No

Note that a lot of these depend on implementation. For merge sort, we use an auxiliary array to do the merging, and that takes $\Theta(n)$ memory. There is an in-place variant, but it is a terrible mess. When merge sorting linked lists, merge sort is still $O(n)$ space, since we create $O(n)$ single item queues. For quicksort, the space complexity comes from the size of the call stack in a recursive implementation.

2.2 For each unstable sort, give an example of a list where the order of equivalent items is not preserved.

```
Heapsort: 1a, 1b, 1c
Quicksort: 1, 5a, 2, 5b, 3
```

Note that if using quicksort that randomizes the array, any array could yield instability.

2.3 In the real world, what are some other tradeoffs we might want to consider when designing and implementing a sorting algorithm?

1. Constant factors in runtime, especially when working with small inputs.
2. Readability when other engineers are using your algorithm.

3 Bounding Practice

Given an array, the heapification operation permutes the elements of the array into a heap. There are many solutions to the heapification problem. One approach is bottom-up heapification, which treats the existing array as a heap and rearranges all nodes from the bottom up to satisfy the heap invariant. Another is top-down heapification, which starts with an empty heap and inserts all elements into it.

3.1 Why can we say that any solution for heapification requires $\Omega(n)$ time?

In order to check that an array satisfies the heap invariant, we have to at least look at every element, which takes linear time.

3.2 Show that the worst-case runtime for top-down heapification is in $\Theta(n \log n)$. Why does this mean that the optimal solution for heapification takes $O(n \log n)$ time?

This means that the optimal solution for heapification takes $O(n \log n)$ time since at least one solution for heapification takes $O(n \log n)$ time.

3.3 In contrast, bottom-up heapification is an $O(n)$ algorithm. Is bottom-up heapfication asymptotically-optimal?

Since the running time of bottom-up heapify is $\Theta(n)$ and any solution for heapification requires $\Omega(n)$, bottom-up heapification is asymptotically optimal.

3.4 *Extra*: Show that the running time of bottom-up heapify is $\Theta(n)$. You should make use of this summation and its derivative:

$$\sum_{i=0}^{\infty} x^i = \frac{1}{1-x} \qquad \sum_{i=0}^{\infty} i x^i = \frac{x}{(1-x)^2}$$

Running time of heapify is:

$$\begin{aligned}
\sum_{i=0}^{\log n} i \frac{n}{2^{i+1}} &= \frac{n}{2} \left(\sum_{i=0}^{\log n} i \left(\frac{1}{2} \right)^i \right) \\
&\leq \frac{n}{2} \left(\sum_{i=0}^{\infty} i \left(\frac{1}{2} \right)^i \right) \\
&= \frac{n}{2} \frac{\frac{1}{2}}{(\frac{1}{2})^2} \\
&= \Theta(n)
\end{aligned}$$

Essentially, the idea is just that each level roughly doubles the work, so the total runtime dependency on n is linear.