CS 61B Spring 2018

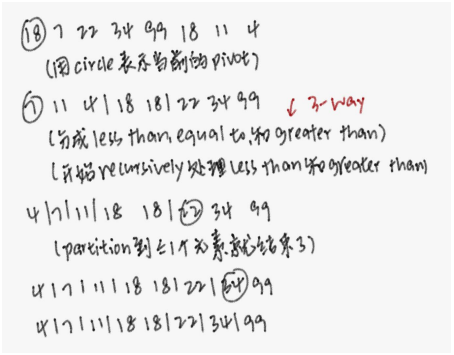
More Sorting

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
```



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

O(N^2): all elements are equal

a more general case: the pivot is always the smallest or the greatest element

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.

O(N log N): the pivot is exactly the median element; this can be justified by the sorting bound we learnt in class.

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

1. randomness
   choose the pivot randomly
   shuffle items before sort
2. smarter pivot selection
3. introspection

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	O(N)~O(N^2)	O(1)	yes
Heapsort	O(N log N)	O(1)	no
Mergesort	O(N log N)	O(N)	yes
Quicksort	~O(N log N)	~O(log N)	no

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

I didn't get the question, because I think as long as there are equivalent items, the order can be destroyed

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

the actual input size and input data type can have a huge impact on our selection of sorting algorithms

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?

we need to at least traverse all the array elements once to make sure that they satisfiy the heap invariants , which means in min heap, all parents should be no greater than their children.

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?

1. the time required for heap construction is big Theta(n log n)
2. because big o is a looser bound than big theta

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

yes

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:

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