Prof. Benyuan Liu
Fall - 2017

ANALYSIS OF ALGORITHM - HW -6 SOLUTIONS

1. **Credits**: Taylor M. Langlois

1. Exercise 9.3-3 (page 223) (10 points)

9.3-3 Show how quicksort can be made to run in $O(n \lg n)$ time in the worst case, assuming that all elements are distinct.

To run in $O(n \lg n)$ worst-case time, the modification would use the deterministic PARTITION algorithm modified to take an element and to partition it around as an input parameter. SELECT would take an array, A, bounds p and r of subarray An and rank i of an order statistic. In time linear in the size of the subarray $A[p \dots r]$ it returns the i^{th} smallest element in $A[p \dots r]$.

BEST-CASE-QUICKSORT (A, p, r)
if $p<r$
  then $i = \lfloor (r-p+1)/2 \rfloor$
    x= SELECT (A, p, r, i)
    q= PARTITION (x)
    BEST-CASE-QUCIKSORT $(A, p, q-1)$
    BEST-CASE-QUCIKSORT $(A, q+1, r)$

For and array of n-elements, the largest subarray BEST-CASE-QUICKSORT recurses on has $n/2$ elements and occurs when $n=r-p+1$ is even, then $A[q+1 \dots r]$ has $n/2$ elements and $A[p \dots q-1]$ has $n/2$ -1 elements. Since BEST-CASE-QUICKSORT always recurses on subarrays, at most, half the size of the original, the recurrence for worst-case running time would be $T(n) \le 2T(n/2)+\Theta(n) = O(n \lg n)$.

Prof. Benyuan Liu
Fall - 2017

2. **Credits**: Victoria Albanese

Victoria Albanese
Analysis of Algorithms

November 21, 2017
Homework 6

2. **Exercise 9.3-5**: 10 points
Suppose you have a "black box" worst case linear-time median subroutine. Give a simple, linear time algorithm that solves the selection problem for an arbitrary order statistic.

Let A be the array with elements A[p…r], and let the ith order statistic be called i. Let the black box subroutine have signature Black_Box(A) and return a partition element.

```
1    Black_Box_Select(A, p, r, i)
2        if p == r
3            return A[p]
4        q = Black_Box(A, p, r)
5        k = q – p + 1
6        if i == k
7            return A[q]
8        else if i < k
9            return Black_Box_Select(A, p, q – 1, i)
10       else
11           return Black_Box_Select(A, q + 1, r, i - k)
```

If we go through this algorithm line by line, we can get a good idea of the runtime. Firstly, lines 2-3 contribute constant time O(1), and handle program return when the subarray size is one. Line 4 calls Black_Box, which is given to complete is linear time, O(n), even in the worst case. The first condition of the if statement returns immediately, with the same effect as lines 2-3, and contributes constant time O(1). The other two statements in the if-clause contribute T(n/2) time, since the black box subroutine returns the median, or n/2 element, meaning that each subarray is of size n/2. This gives the recurrence $T(n) = T(n/2) + O(n)$. This breaks down into the series $O(n) + O(n/2) + O(n/4) + O(n/8) + \ldots 1 = O(n + \frac{1}{2} n + \frac{1}{4} n + \ldots 1) = O(2n)$.

Therefore, this algorithm runs in linear time.

Prof. Benyuan Liu
Fall - 2017

3. **Credits**: Donovyn Pickler

3) Exercise 9.3-6 (10 points)

The k_{th} quantiles of an n-element set are the $k-1$ order statistics that divide the sorted set into k equal-sized sets (to within 1). Give an $O(n\ lg(k))$ time algorithm to list the k_{th} quantiles of a set.

```
                                                        Run-time
                                                        O(n lg(k))
get-quantiles(A,Q,k){
                                                        O(1)
    if(k == 1){                                         O(1)
    Return
    }
                                                        O(1)
    x = A.length                                        O(1)
    y = floor(k/2)                                      O(n)
    z = select(A, floor(i * x/k))
                                                        O(n)
    Partition(A, z)

    Insert(Q, get-quantiles(A.from(1,z),Q,y))           O(lg (k))
    Insert(Q, get-quantiles(A.from(z+1,x),Q,y))         O(lg (k))
                                                        O(1)
    Return z
}
```

Prof. Benyuan Liu
Fall - 2017

4. **Credits**: Ryan Cauble

Problem (9-1):

Given a set of n numbers, we wish to find the i largest in sorted order using a comparison-based algorithm. Find the algorithm that implements each of the following methods with the best asymptotic worst-case running time, and analyze the running times of the algorithms in terms of n and i.

***a.* Sort the numbers, and list the i largest.**

We could sort the numbers using merge sort or heapsort, which both take $\Theta(n \lg n)$ worst-case time Then we put the i largest elements into the output array, taking $\Theta(i)$ time.

In total he worst-case running time will be: $\Theta(n \lg n + i) = \Theta(n \lg n)$

***b.* Build a max-priority queue from the numbers, and call EXTRACT-MAX i times.**

We can implement the priority queue as a heap. Using BUILD-HEAP takes $\Theta(n)$ time.

Then we call HEAP-EXTRACT-MAX i times to get the i largest elements, in $\Theta(i \lg n)$ worst-case time, and store them in reverse order of extraction in the output array.

The worst-case extraction time is $\Theta(i \lg n)$ because i extractions from a heap with $O(n)$ elements takes $i \cdot O(\lg n) = O(i \lg n)$ time, while half of the i extractions are from a heap with $\geq n/2$ elements.

So those $i/2$ extractions take $(i/2)\Omega(\lg(n/2)) = \Omega(i \lg n)$ time in the worst case.

Total worst-case running time will be: $\Theta(n + i \lg n)$.

***c.* Use an order-statistic algorithm to find the i th largest number, partition around that number, and sort the i largest numbers.**

If we use the SELECT algorithm from Section 9.3 to find the i th largest number in $\Theta(n)$ time.

Then we can partition around that number in $\Theta(n)$ time.

Next we would sort the i largest numbers in $\Theta(i \lg i)$ worst-case time with something like heapsort maybe even merge sort.

The total worst-case running time will be: $\Theta(n + i \lg i)$.