CSE 2331 Homework 6 Solutions

1. Let $xs = [11, 5, 24, 13, 6, 2, 9, 14, 4, 7, 8]$.

   (a) Draw the execution of $r \leftarrow \text{PARTITION}(xs, 0, 10, 9)$.

```
partition(xs, i=0, j=10, p=9)

We will display the state of the array after each
loop iteration.

  before: | 11 5 24 13 6 2 9 14 4 7 8 |

  loop 1: | 8 5 24 13 6 2 9 14 4 7 11 |
  loop 2: 8 | 5 24 13 6 2 9 14 4 7 11 |
  loop 3: 8 5 | 24 13 6 2 9 14 4 7 11 |
  loop 4: 8 5 | 24 13 6 2 9 14 4 7 | 11
  loop 5: 8 5 | 7 13 6 2 9 14 4 24 | 11
  loop 6: 8 5 7 | 13 6 2 9 14 4 24 | 11
  loop 7: 8 5 7 | 13 6 2 9 14 4 | 24 11
  loop 8: 8 5 7 | 4 6 2 9 14 13 | 24 11
  loop 9: 8 5 7 4 | 6 2 9 14 13 | 24 11
 loop 10: 8 5 7 4 6 | 2 9 14 13 | 24 11
 loop 11: 8 5 7 4 6 2 | 9 14 13 | 24 11
 loop 12: 8 5 7 4 6 2 | 9 14 | 13 24 11
 loop 13: 8 5 7 4 6 2 | 9 | 14 13 24 11
```

   (b) What is r? $r = 7$ (index of 9) using 1-indexed arrays, or $r = 6$ assuming 0-indexing.

2. We saw in class that deterministic and randomized QUICKSORT both have $O(n^2)$ worst-case time.

   (a) What causes the worst-case behavior?

   Solution. The worst case behavior is caused by consistently chosing pivots that result in imbalanced splits. This can happen if the pivot always ends up at the smallest or largest index in the array. □

   (b) Why does randomizing the pivot help? Hint: Think about the analysis of randomized QUICKSORT.

   Solution. Randomizing the pivot avoids the worst case on average. The probability of being within a constant distance of the ends, for any constant, vanishes asymptotically. Instead, with constant probability, you are at least a fixed fraction from the ends of the list. □

3. Suppose we are given a procedure, PICKPIVOT, that takes an array as input and is *guaranteed* to return an element that is a k-th order statistic for some $\frac{1}{4}n \leq k \leq \frac{3}{4}n$ on arrays of size n in worst-case linear-time.

   NOTE. I did not ask you to figure out PICKPIVOT, but wanted to provide you with a suitable procedure for reference.

   The idea here is known as MEDIAN-OF-MEDIANS. We proceed by chunking the input array into constant size (5 in this case) chunks. We can find the median of a constant size chunk in constant time just by sorting it.[1]

   We put the medians into an array and if it's small enough (constant-sized) we call median to get its median. Otherwise we recurse on the medians array. This is not guaranteed to return the true median. Instead, it returns an approximation of the median value, that is somewhere in the middle 50% of values.

   The relevant recursion is $T(n) = T(n/5) + \Theta(n)$ which expands to a geometric series times n.

```
1  # find the median in Θ(n log n)-time
2  def median(xs):
3      n = len(xs)
4      xs = sorted(xs)
5      return xs[n//2]
6
7  def pick_pivot(xs, l, h):
8      medians = []
9      for i in range(l, h+1, 5):
10         u = min(i+5, h+1) # don't leave xs[l,...,h]
11
12         # at most 5 elements so Θ(1)-time
13         m = median(xs[i:u])
14         medians.append(m)
15
16     n_meds = len(medians)
17     if n_meds > 5:
18         return pick_pivot(medians, 0, n_meds-1)
19
20     #again, medians is at most 5 elements
21     return median(medians)
```

   (a) Give pseudocode demonstrating how PICKPIVOT can be used to improve QUICKSORT and perform worst case analysis of its running time assuming distinct values in the input array.

[1] Since the input size never exceeds 5 it takes at most constant time sort it and extract the median.

Solution. Example pseudocode:

```
def quicksort(xs, l, h):
    if l >= h:
        return

    # find a k-th order statistic
    # for 0.25*n <= k <= 0.75*n
    #   (where n = h-l+1)
    v = pick_pivot(xs, l, h)

    # partition around v
    p = partition(xs, l, h, v)

    quicksort(xs, l, p-1)
    quicksort(xs, p+1, h)
```

Note that since the pivot index is a k-th order statistic for some $0.25n \leq k \leq 0.75n$, it follows that the pivot value is in the middle two quarters of the array after a call to partition. Recall that QUICKSORT splits on this value and that its worst case behavior is due to imbalanced splits. It follows, therefore, that a worst-case (most-imbalanced) split results in recursions of size $T(n/4)$ and $T(3n/4)$.

As a result, we can get a worst-case upper bound by considering what happens if every recursive split is worst possible. In this case,

$$T(n) \leq cn + T(n/4) + T(3n/4).$$

This recursion has two terms, but note that the arguments of the recursive calls on the right always sum to n. Further note that $T(n)$ always does cn work when excluding recursive subcalls. These two facts taken together imply that the total work done at any depth in the recursion tree is at most cn.

We can get an upper bound on the depth of the recursion tree by considering the maximum possible recursion depth. This is given by $T(3n/4)$ which has to process a problem of size $3/4n$ its recursive descendents which must process three-quarters of their respective inputs. In other words, the total recursive depth is at most the smallest k satisfying $n\left(\frac{3}{4}\right)^k \leq 1$.

Solving for k, we see that k is at worst the smallest integer exceeding $\log_{4/3}(n)$. Thus $T(n) \leq cn(k+1) \in O(n \log n)$. To get the lower bound, either consider best-case splits in the middle or note that all correct sorts in the comparison model must be $\Omega(n \log n)$. Either way, it follows that $T(n) \in \Theta(n \log n)$.

□

(b) Give pseudocode demonstrating how PICKPIVOT can be used to find the (lower) median and perform worst case analysis of its running time assuming distinct values in the input array.

Solution. Example pseudocode:

```
1  def find_median(xs, l, h):
2      v = pick_pivot(xs, l, h)
3      p = partition(xs, l, h, v)
4
5      # compute the lower median
6      n = len(xs)
7      m = (n + 1) // 2
8
9      k = p + 1
10     if k == m:
11         return xs[p]
12
13     if k > m:
14         return find_median(xs, l, p-1)
15
16     return find_median(xs, p+1, h)
```

Let $n = (h-l+1)$. Note that since the pivot index is a k-th order statistic for some $0.25n \leq k \leq 0.75n$, it follows that the pivot value is in the middle two quarters of the array $xs[l, \ldots, h]$ after a call to partition. Observe that FindMedian splits on this value and that its worst case behavior occurs when the desired m is on the bigger side. It follows, therefore, that a worst-case split results in recursions of size $T(3n/4)$. In other words, $T(n) \leq T(3n/4) + cn$ where the cn covers the wosrt-case linear-time work of picking the pivot, finding the value, and partitioning.

After one substitution,

$$T(n) \leq T((3/4)^2 n) + cn(3/4) + cn.$$

After two substitutions,

$$T(n) \leq T((3/4)^3 n) + cn(3/4)^2 + cn(3/4) + cn.$$

After $k-1$ substitutions,

$$T(n) \leq T((3/4)^k n) + cn \sum_{i=0}^{k-1} \left(\frac{3}{4}\right)^i.$$

Now note that this terminates when $n(3/4)^k \leq 1$. That is, when $k \geq \log_{4/3}(n)$. Plugging in for k and noting that the sum is a geometric series, we have

$$T(n) \leq T(1) + cn \frac{1}{1 - \frac{3}{4}} = c' + 4cn.$$

Therefore, $T(n) \in O(n)$. Note that we must call PARTITION at least once and therefore $T(n) \in \Omega(n)$. It follows that $T(n) \in \Theta(n)$ in the worst case.

□

(c) What, if anything, changes in your analysis if PICKPIVOT's guarantee only holds asymptotically?[2]

Solution. First, to explain, having saying that PICKPIVOT's guarantee only holds asymptotically means that there exists an n_0 such that for all arrays of size $n \geq n_0$, PICKPIVOT is *guaranteed* to return an element that is a k-th order statistic for some $\frac{1}{4}n \leq k \leq \frac{3}{4}n$.

The analysis remains the same if the results only hold asymptotically. This is because we only are about the behavior in the limit of large n, if the guarantee does not hold for any $n < n_0$ then this is not a big deal because, at worst, this returns bad splits for bounded size arrays, which makes them solvable in constant time. □

4. Suppose we are given a worst-case linear-time (lower) median-finding procedure, HAPPYMEDIAN. In other words, perhaps your answer to 3b. Give a worst-case linear-time algorithm to find the k-th order statistic. Show your analysis.

Solution. Example pseudocode.

```
1  def selection(xs, l, h, k):
2      v = happy_median(xs, l, h)
3      p = partition(xs, l, h, v)
4
5      m = p + 1 - l
6      if m == k:
7          return xs[p]
8
9      if m > k:
10         return selection(xs, l, p-1, k)
11
12     return selection(xs, p+1, h, k-m)
```

To determine the runtime, let $T(n)$ denote the runtime of selection when processing an array of size n. Note here that the array we are processing is $xs[l, \ldots, h]$ and so $n = (h - l + 1)$.

The recursion either proccesses $xs[l, \ldots, p-1]$ and has size $(p-1) - l + 1 = p - l$ when $m > k$ or processes $xs[p+1, \ldots, h]$ and has size $h - (p+1) + 1 = h - p$ when $m < k$.

[2]Assume it is still worst-case linear time.

Note that

$$p - l = \lfloor (n+1)/2 \rfloor - 1 = \lfloor n/2 \rfloor \leq n/2.$$

and

$$\begin{aligned} h - p &= h - \lfloor (n+1)/2 \rfloor - 1 - l \\ &= (h - l) - \lfloor n/2 \rfloor \\ &= n - 1 - \lfloor n/2 \rfloor \leq n/2. \end{aligned}$$

Since both HappyMedian and Partition are $O(n)$-time algorithms, there exists $c \in \mathbb{R}^+$ such that

$$T(n) \leq cn + T\left(\max\left(p - l, h - p\right)\right) \leq cn + T\left(\max(n/2, n/2)\right) = cn + T(n/2).$$

Solving the recurison $T(n) \leq T(n/2) + cn$. After one substitution,

$$T(n) \leq T(n/2^2) + (n/2)c + cn.$$

After two substitutions,

$$T(n) \leq T(n/2^3) + (n/2^2)c + (n/2)c + cn$$

After $k - 1$ substitutions,

$$T(n) \leq T(n/2^k) + cn \sum_{i=0}^{k-1} \left(\frac{1}{2^i}\right).$$

Now note that this terminates when $n/2^k \leq 1$. That is, when $k \geq \log_2(n)$. We could plug this in to the upper bound of the sum to evaluation the solution. However, we can see by the n multiplying the sum that the upper bound is already linear for $k = 1$. Further, this sum is a geometric series which, in the limit of large k, sums to 2. Observe,

$$T(n) \leq T(n/2^k) + cn \sum_{i=0}^{k-1} \left(\frac{1}{2^i}\right) \leq T(1) + 2cn.$$

Therefore, $T(n) \in O(n)$. That is, $T(n)$ is worst-case linear-time.

□