MA407: Algorithms and Computation
Exercise 4

November 8, 2024

Jay Razdan

---

1. **Exercise 4.1**

   (a) What is the running time (in asymptotic notation) of the **Quicksort** version we saw in the lecture for the following input sequences? Explain why.

$$A_n = 1, 2, ..., n-1, n$$
$$B_n = n, n-1, ..., 2, 1$$

   (b) Devise a different rule to choose the **pivot**, than the one we used in the lectures, so that the resulting Quicksort only needs $O(n\log(n))$ many steps for the sorted sequence A_n.
   Is there any pivot rule such that Quicksort needs $o(n\log(n))$ (that is, an asymptotically slower function of n than $n\log(n)$) steps for A_n?

   (c) Let x be a list with distinct integers and with length n. Assume the value v is stored as element $x[i]$. This value v is a median of x if, after sorting x, the value v can be found in the position with index $\lfloor (n-1)/2 \rfloor$ of the list. Describe (in pseudo-code or words) an algorithm for determining the index i of a median $x[i]$ of x (without changing the entries of x or copying the values of x to a different list). What is the running time of your algorithm?

Ans:

(a) The first input sequence pertains to an array which is already sorted in the desired order. Since the **Quicksort** version we saw in the lecture hinged on a "last element" pivot rule, for every call of the partition function i.e. for every recursion, an array of length n is broken down into sub-arrays of lengths $n-1$ and 0. Therefore, the asymptotic running-time dynamic

implied by this relationship is as follows:

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ T(n) &= T(n-1) + \Theta(n) \\ T(n) &= T(n-2) + \Theta(n) + \Theta(n) \\ \Rightarrow T(n) &= \underbrace{T(0)}_{=0} + \underbrace{\Theta(n) + ... + \Theta(n)}_{length=n} \\ T(n) &= \Theta(n^2) \end{aligned}$$

The second input sequence pertains to an array which is sorted in reverse order. Considering the same pivot rule, it remains that for every recursion/call of the partition function, such an array of length n is broken down into sub-arrays of lengths $n-1$ and 0, albeit with alternating left- and right-children if we consider each sub-array to be a node on a tree. As a result, the asymptotic running-time dynamic implied by the relationship of this sequence is like that of the first: $\Theta(n^2)$.
Hence, it is clear that this commonality in asymptotic running-time is due to the pivot rule of selecting the last element in any given sub-array.

(b) To devise a pivot rule for the required asymptotic running time of the log-linear time for a particular sorted sequence of elements such as A_n, we may exploit the sorted nature of the sequence by choosing the median of each sub-array to be our pivot. By choosing the median, which we know has index $n/2$ for any given sub-array of length n (choosing its necessary floor function if n is even), we split such a sub-array evenly each time we call the partition function.
To see why this allows for log-linear time complexity, we can verify this result by establishing the running-time recurrence relation describing this even split each time:

$$T(n) = 2T(n/2) + \Theta(n),$$

which resolves to:

$$\Rightarrow T(n) = \Theta(n \cdot \log_2(n)).$$

However, there is no such pivot rule such that **Quicksort** needs $o(n\log(n))$ steps for the sequence A_n, because the **Quicksort** algorithm, by default, has a best-performing running complexity of log-linear time i.e. it is an element of $\Omega(n\log(n))$. With regards to specifics, we would have to find a split of sub-arrays **better** than an even split, which is impossible since any other split would never allow for exceeding the performance of log-linear asymptotic running time anyway.

(c) We can modify the traditional **Quicksort** algorithm we saw in the lectures to determine the index i of the median v. To do this, it is first important to notice that the index of the median in a correctly sorted list is $\lfloor (n-1)/2 \rfloor$. To this end, we also know that the partition function within the **Quicksort** algorithm will return the correct position of the selected pivot. So, for the list x, we can have a common pivot rule to choose some pivot p, such as the last element of a (sub-)array. Running the partition function will then return the correct index of the selected pivot; if this index is equal to $\lfloor (n-1)/2 \rfloor$, then we have found the median and we call the initial index of this pivot in the list x. If not, we recursively apply this strategy until we have found a pivot whose correctly ordered index is equal to $\lfloor (n-1)/2 \rfloor$.
In regards to asymptotic running time, we first know that the partition function runs in linear time. We must also realize that trying to find the pivot corresponding to the median will also require looping throughout n possible pivots (since the list x is of length n), and therefore linear time. Combining these two together, and also realizing that there is no quicker way of performing either of them quicker than linear time, the overall asymptotic running time of this algorithm is: $\Theta(n^2)$

2. **Exercise 4.2**

Suppose your are given a list A of even length, $len(A) = n$, and you know that $n/2$ of the entries are equal to x, and the other $n/2$ entries are equal to $y \neq x$. You want to find x and y.
A simple deterministic algorithm (not relying on randomisation) for this problem that is guaranteed to find x and y proceeds as follows. First it sets $x = A[0]$. It then goes through the elements at positions $i = 1, ..., n-1$. As soon as $A[i] \neq x$, the algorithm sets $y = A[i]$ and outputs x and y.

(a) Give an implementation of this algorithm in Python, and analyse its running time using Θ-notation.

Now consider the following randomised algorithm for this problem. Repeatedly draw a random entry (with replacement, so you may draw the same entry more than once) until you have seen two distinct numbers for the first time.

(b) Give an implementation of this algorithm in Python.

(c) What is the expected number of random draws performed by this algorithm?

(d) Use your answer to (c) to analyse the expected running time of this algorithm using big-O notation.

(e) Is this a Monte Carlo or a Las Vegas algorithm?

Hint: For part (c), it may be useful to consider the random variable Y which counts the number of random draws, and compute the probability $Pr[Y > k]$.

Ans:

(a)

```
def deterministic(A):
    x = A[0]
    for i in range(1,len(A)):
        if A[i] != x:
            y = A[i]
    return x,y

A = [6,5,5,5,6,5,6,6]
deterministic(A)
```

✓ 0.0s

```
(6, 5)
```

The above deterministic algorithm requires the use of a single for loop - the best case scenario would be that the first two elements are different: $\Theta(1)$ time complexity, however the worst case scenario would require the

algorithm to loop a maximum of $n/2$ times: $\Theta(n)$ time complexity. Since the latter dominates the former, the asymptotic Θ-notation running time of this algorithm is: $\Theta(n)$.

(b)

```
import random

def randomised(A):
    x = 0
    y = 0
    while x == y:
        x = random.choice(A)
        y = random.choice(A)
    return x,y

A = [6,5,5,5,6,5,6,6]
randomised(A)
```

✓ 0.0s

```
(6, 5)
```

(c) Let the random variable Y count the number of random draws of y against a given value of x. The mathematical definition of $E[Y]$ is $:= \Sigma_{k=1} k \cdot P(Y = k)$. Let us use the fact that $P(Y > k)$ also means the probability that we roll only $x's$ for the first k number of draws. So, $P(Y > k) = (1/2)^k = 1/2^k$. This is important as it allows us to write the expectation of Y as: $E[Y] := \Sigma_{k=1} P(Y > k) = \Sigma_{k=1} 1/2^k = 2$. Thus, the expected number of random draws performed by this algorithm is: **2**.

(d) Since the expected number of random draws performed by the algorithm is equal to **2** regardless of list length i.e. input size, the expected asymptotic O-notation running time of this algorithm is: $O(1)$, i.e. constant time.

(e) This is a Las Vegas algorithm as it will always return the correct answer, unlike a Monte Carlo algorithm. Additionally, the algorithm may "give up" i.e. run indefinitely, because there is non-zero probability of having a near-infinite number of draws without success.