Patrick Braun, Class Group 3, Xinyi Xu

MA214 Algorithms and Data Structures LT 2021/22

Homework 4

Instructions: Please submit your solutions via Gradescope by **Friday, 18 February 2022, 10:00am**. Make sure your name, your class group number, and the name of your class teacher is put on **every page** of your submission. Your submission should, ideally, be a **PDF** file.

Exercise 4.1: Master Theorem **3 pts**

Use the Master Theorem to find solutions to the following recurrences. Make sure that you state and justify which case of the Master Theorem you use.

(a) $T(n) = 3 \cdot T(n/4) + n \log_2(n)$

(b) $T(n) = 3 \cdot T(n/2) + n$

(c) $T(n) = 3 \cdot T(n/3) + n/2$

Exercise 4.2: HeapSort **1 pts**

In the lectures last week, we saw the HeapSort algorithm, which sorts a list in place and in worst-case time $O(n \cdot \log_2(n))$. Using the figures from the lecture as a model, illustrate the operation of `BuildMaxHeap` on the list

$$A = [5, 3, 17, 10, 84, 19, 6, 22, 9].$$

Exercise 4.3: Randomised algorithms **3 pts**

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 $i = 2, \ldots, n-1$. As soon as $A[i] \neq x$, the algorithm sets $y = A[i]$ and outputs x and y.

(a) Implement this algorithm in Python, and analyse its running time using big 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) Implement 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 that $\mathbf{Pr}[Y > k]$.

Exercise 4.4: Binary Search **3 pts**

Given an arbitrary list, an algorithm searching for a specific target value in the list cannot do much better than scanning the list elements one by one, which requires $\Omega(n)$ time in the worst case. On the other hand, if the algorithm is guaranteed to be given a sorted list (say, in non-decreasing order), it is possible to do drastically better. An algorithm, with a single comparison, can eliminate half of the entries of the list as possible locations. This is the idea underlying the "Binary Search" algorithm.

(a) Implement a recursive Python function that searches for value `x` in the list `A`, which is sorted in non-decreasing order. Your function should return an index `i`, where `A[i]=x` (or the special Python value `None` if no such `i` exists) and have time complexity $O(\log n)$, where n is the length of the input list. (Do not use slicing, as it is a linear-time operation.)

(b) Implement an iterative version of the binary search in Python, `BinarySearch(A,x)`, where `A`, `x`, and the time complexity are the same as above.

(c) Give a loop invariant for your algorithm in part (b). Prove the correctness of that algorithm using your loop invariant.

Ex 4.1

(a) $T(n) = 3 \cdot T(n/4) + n\log_2(n)$

$\log_b a = \log_4 3 < 1$

<u>Case 3:</u>

As $n\log_2 n = \Omega\left(n^{\log_4 3+\varepsilon}\right) \Rightarrow T(n) = \Theta(n\log_2 n)$

(b) $T(n) = 3 \cdot T(n/2) + n$

$\log_b a = \log_2 3 > 1$ <u>Case 1:</u>

So $n = O\left(n^{\log_2 3-\varepsilon}\right) \Rightarrow T(n) = \Theta\left(n^{\log_2 3}\right)$

(c) $T(n) = 3 \cdot T(n/3) + n/2$

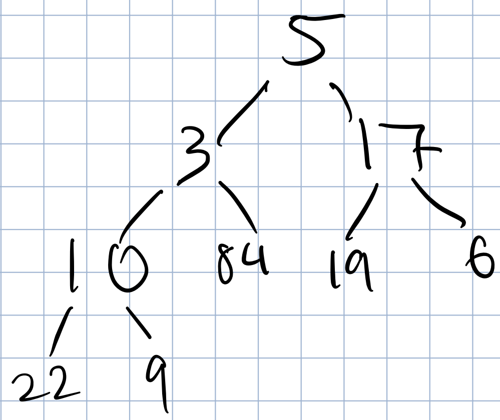
$\log_b a = \log_3 3 = 1$ <u>Case 2:</u>

$f(n) = n = \Theta(n^1) \Rightarrow T(n) = (n \lg n)$

Exercise 4.2: HeapSort **1 pts**

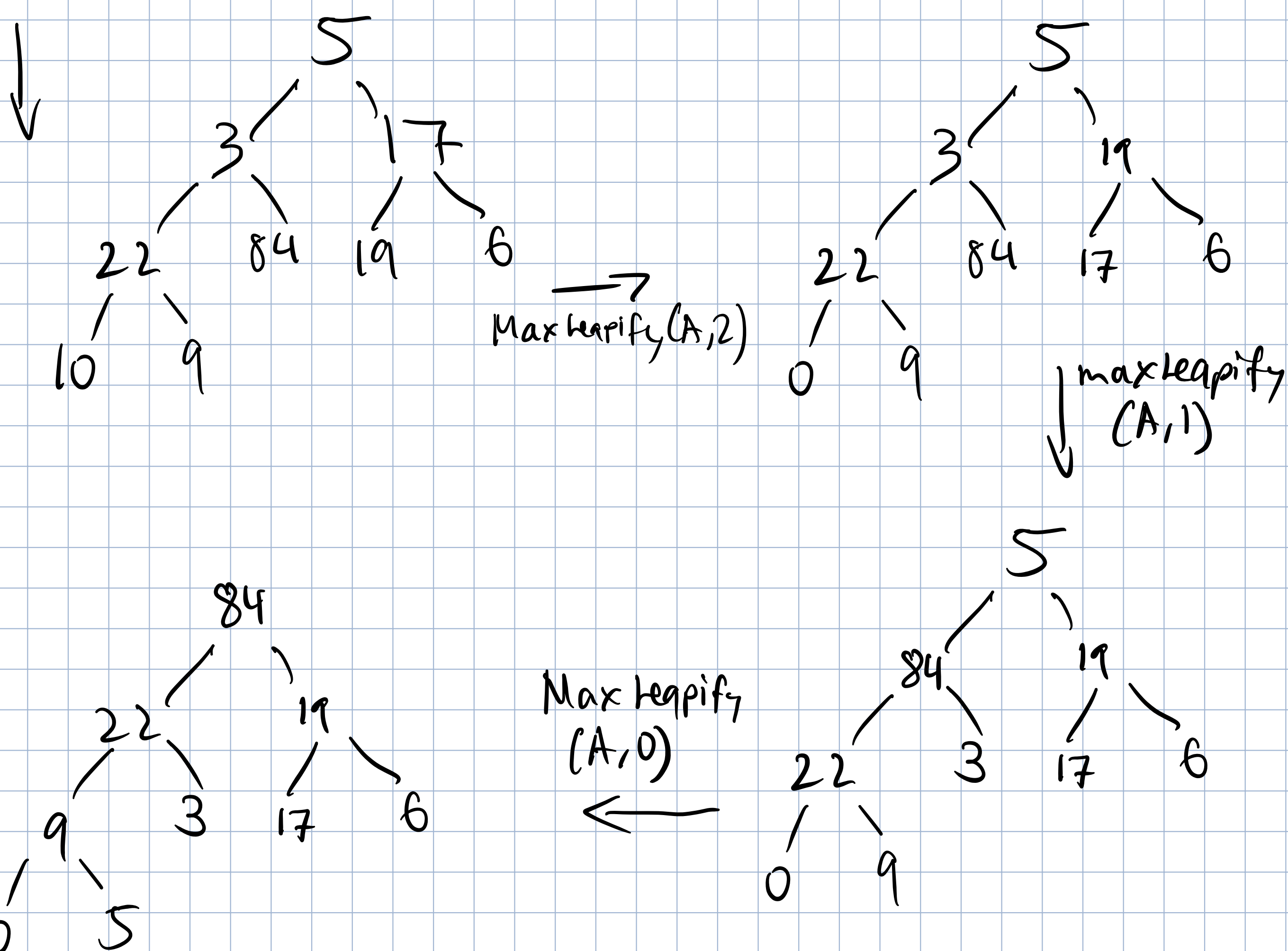
In the lectures last week, we saw the HeapSort algorithm, which sorts a list in place and in worst-case time $O(n \cdot \log_2(n))$. Using the figures from the lecture as a model, illustrate the operation of `BuildMaxHeap` on the list

$$A = [5, 3, 17, 10, 84, 19, 6, 22, 9].$$



As len(A) = 9
We only call max heapify from
A[len(A)//2−1] ... A[0]

So start with maxheapify(A, 3)



Exercise 4.3: Randomised algorithms **3 pts**

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 $i = 2, \ldots, n-1$. As soon as $A[i] \neq x$, the algorithm sets $y = A[i]$ and outputs x and y.

(a) Implement this algorithm in Python, and analyse its running time using big Θ-notation.

```python
def findXAndY(A):
    x = A[0]
    for i in A:
        if i != x:
            return x, i
```

$\Theta(1)$
$\Theta(n/2)$
$\Theta(1)$
$\Theta(1)$

In the worst case half are on each side. Then the algorithm will run $\Theta(\frac{n}{2}) = \Theta(n)$ through the for loop.

Hence $\Theta(n)$.

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) Implement this algorithm in Python.

(b)

```
def findXAndYRandom(A):
    x = A[randint(0, len(A) - 1)]
    new = x
    while(new == x):
        new = A[randint(0, len(A) - 1)]
    return x, new
```

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

Let X be the number of random draws.

$$E(X) = \sum_{i=0}^{\infty} P(X=i)\, i$$

$$= \sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^i i = 1\frac{1}{2} + 2\frac{1}{2^2} + 3\frac{1}{2^3} + 4\frac{1}{2^4}$$

$$= \frac{\frac{1}{2}}{\left(1-\frac{1}{2}\right)^2} = \frac{\frac{1}{2}}{\frac{1}{4}} = 2$$

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

As $\mathbb{E}(X) = 2$ this is not dependent on n, so we have an algorithm of $O(1)$.

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

Las Vegas Algorithm as it will keep trying till we get the right answer.

Exercise 4.4: Binary Search **3 pts**

Given an arbitrary list, an algorithm searching for a specific target value in the list cannot do much better than scanning the list elements one by one, which requires $\Omega(n)$ time in the worst case. On the other hand, if the algorithm is guaranteed to be given a sorted list (say, in non-decreasing order), it is possible to do drastically better. An algorithm, with a single comparison, can eliminate half of the entries of the list as possible locations. This is the idea underlying the "Binary Search" algorithm.

(a) Implement a recursive Python function that searches for value `x` in the list `A`, which is sorted in non-decreasing order. Your function should return an index `i`, where `A[i]=x` (or the special Python value `None` if no such `i` exists) and have time complexity $O(\log n)$, where n is the length of the input list. (Do not use slicing, as it is a linear-time operation.)

```python
def recurseBinarySearch(A, start, end, x):
    if start > end:
        return None

    mid = (start + end) // 2
    if A[mid] == x:
        return mid
    elif A[mid] > x:
        return recurseBinarySearch(A, start, mid - 1, x)
    elif A[mid] < x:
        return recurseBinarySearch(A, mid + 1, end, x)
```

(b) Implement an iterative version of the binary search in Python, `BinarySearch(A,x)`, where `A`, `x`, and the time complexity are the same as above.

```python
def iterativeBinarySearch(A, x):
    i = 0
    j = len(A) - 1
    while i <= j:
        mid = (i + j) // 2
        if A[mid] == x:
            return mid
        elif A[mid] > x:
            j = mid - 1
        elif A[mid] < x:
            i = mid + 1
    return None
```

(c) Give a loop invariant for your algorithm in part (b). Prove the correctness of that algorithm using your loop invariant.

(c) Loop invariant if x is in A and $A[k]=x$ then $i \le k \le j$.

Initialisation Clearly if x is in A then it must be in the range $A[0] \dots A[\mathrm{len}(A)-1]$

Maintenance: Suppose it is true at the start of the iteration. Choose the middle element.

- If mid $= x$ we're done
- If mid is greater, then as the list is sorted it must be that $i \le k < \text{mid}$ so $j = \text{mid}-1$ achieves this
- If mid is less, then as sorted, it must be that $\text{mid} < k \le j$ so $i = \text{mid}+1$ achieves this

Termination
- If we terminate early, we're done.
- Otherwise if we terminate, then $i > j$ so then k must be in the range $j \dots i$ but that is the empty set. So None is correctly returned.