Patrick Braun, Class Group 3, Xinyi Xu

MA214 Algorithms and Data Structures **LT 2021/22**

Homework 1

Instructions: Please submit your solutions via Gradescope by **Friday, 28 January 2022, 11:00am**. Make sure your name, your class group number, and the name of your class teacher is put on your submission. Your submission should, ideally, be a **PDF** file. For example, you can use a scanner app on a smartphone to create PDF files, instead of simply taking pictures of your written solutions.

Exercise 1.1: The one-dimensional peak problem — 1 pt

Consider the problem of finding the index of the peak of a sequence of $n \geq 2$ distinct integers that is first increasing and then decreasing. A peak element is one that is larger than its immediate neighbours. The first and last element just need to be larger than their single neighbour.

Following the example of the sorting problem in the lecture, formally define this computational problem.

Exercise 1.2: A slow iterative algorithm — 4 pt

The one-dimensional peak problem can be solved by an iterative algorithm, which scans through the sequence once.

(a) Describe this algorithm in words. Be careful and make sure your algorithm handles all boundary cases correctly.

(b) Implement your algorithm in Python. And test it on three sequences, one where the peak is at the beginning, one where it is in the middle, and one where it is at the end.

(c) For a sequence of length $n \geq 2$, what is the minimum and maximum number of comparisons that your algorithm has to perform?

Exercise 1.3: A faster recursive algorithm — 5 pt

A faster algorithm for the one-dimensional peak problem can be obtained by following the divide-and-conquer paradigm. The basic idea is to recursively split a problem of size n into a single problem of size roughly $n/2$.

(a) Describe this algorithm in words. Again, be careful that your algorithm handles all boundary cases correctly.

(b) Implement your algorithm in Python. And as above, test it on three sequences, one where the peak is at the beginning, one where it is in the middle, and one where it is at the end.

(c) Let $T(n)$ describe the running time of this algorithm on sequences of length n. Assume (for simplicity) that $n = 2^i$ for some $i \geq 1$. Then the running time is well captured by the following recurrence relation:

$$T(n) = T(n/2) + 1 \qquad \text{for } n \geq 1, \text{ and}$$
$$T(1) = 1.$$

What is the solution to this recurrence relation?

Not part of this question, but good things to think about: How does your answer to (c) change if the 1s on the right-hand side of the recurrence relation were replaced with a 3 or any other constant? Why is it okay to assume that $n = 2^i$ for some $i \geq 1$ if we are interested in obtaining an upper bound on the running time?

Exercise 1.1: The one-dimensional peak problem **1 pt**

Consider the problem of finding the index of the peak of a sequence of $n \geq 2$ distinct integers that is first increasing and then decreasing. A peak element is one that is larger than its immediate neighbours. The first and last element just need to be larger than their single neighbour.

Following the example of the sorting problem in the lecture, formally define this computational problem.

Input: A sequence of $a_1, a_2, \ldots, a_n$ such that $\forall i,j\ a_i = a_j$ if $i \neq j$

$\exists$ unique $i \in \{1, \ldots n\}$ such that, if $j < i$ increasing
$j \geq i$ decreasing

Output: Unique i such that $a_i > a_{i+1}$ $a_i > a_{i-1}$

(For edges only need to satisfy $a_1 > a_2$ or $a_n > a_{n-1}$)

Exercise 1.2: A slow iterative algorithm **4 pt**

The one-dimensional peak problem can be solved by an iterative algorithm, which scans through the sequence once.

(a) Describe this algorithm in words. Be careful and make sure your algorithm handles all boundary cases correctly.

(a) Iterate through each term in $a_1, a_2 \dots, a_n$ one by one. When reaching each term, check if it is larger than all of its neighbours (ie for ends only 1 comparison otherwise 2). If this true return the index of the current term, else move to the next term and repeat. If you reach the last term a_n, return n without checking its neighbour (as we assumed existence of peak).

(b) Implement your algorithm in Python. And test it on three sequences, one where the peak is at the beginning, one where it is in the middle, and one where it is at the end.

My code:

```python
def peak_iter(arr):
    for i, value in enumerate(arr):
        if i == 0 and value > arr[1]:
            return i
        elif i == len(arr) - 1:
            return i
        elif value > arr[i - 1] and value > arr[i + 1]:
            return i


def main():
    example1 = [1, 2, 3, 4, 5, 6, 7, 8]
    example2 = [8, 7, 6, 5, 4, 3, 2, 1]
    example3 = [1, 2, 3, 4, 3, 2, 1, 0]
    ls = [example1, example2, example3]
    for l in ls:
        print(f"Peak in {l} is at index: {peak_iter(l)}")
```

Output:

```
Peak in [1, 2, 3, 4, 5, 6, 7, 8] is at index: 7
Peak in [8, 7, 6, 5, 4, 3, 2, 1] is at index: 0
Peak in [1, 2, 3, 4, 3, 2, 1, 0] is at index: 3
```

(c) For a sequence of length $n \geq 2$, what is the minimum and maximum number of comparisons that your algorithm has to perform?

Best case: First element is the peak ($a_1 > a_2$). Then the algorithm will return 1 after first comparison. So 1 comparison

Worst Case: Last element is the peak ($a_{n-1} < a_n$). Then The algorithm will require 1 comparison for a_1 and 2 comparisons for $a_2, \ldots a_{n-1}$.

So $2 \cdot (n-2) + 1 = 2n - 3$

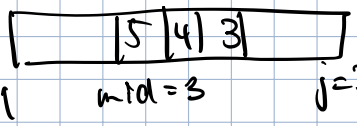
Exercise 1.3: A faster recursive algorithm **5 pt**

A faster algorithm for the one-dimensional peak problem can be obtained by following the divide-and-conquer paradigm. The basic idea is to recursively split a problem of size n into a single problem of size roughly $n/2$.

(a) Describe this algorithm in words. Again, be careful that your algorithm handles all boundary cases correctly.

(a) Consider the sequence $a_1, a_2 \dots a_n$ and let i, j represent the range of values still considered.

So start $i=1, j=n$ (1 indexing). Then check the middle element to see if it is a peak. If yes return the index. Otherwise check which side of the middle is larger. Move i, j such that the indexes represent the half which was larger

(i.e 

let $j = 2$ for next run)

Then return the value of doing the same algorithm with the smaller range of values.

(Also if $i=j$ this is the index to return.

(b) Implement your algorithm in Python. And as above, test it on three sequences, one where the peak is at the beginning, one where it is in the middle, and one where it is at the end.

My code:

```python
def peak_divide_conquer(arr, i, j):
    mid = (i + j) // 2
    # base case
    if i == j:
        return i
    # check for peak
    if arr[mid] > arr[mid - 1] and arr[mid] > arr[mid + 1]:
        return mid
    # need to go to right
    elif arr[mid] > arr[mid - 1]:
        return peak_divide_conquer(arr, mid + 1, j)
    # need to go to left
    else:
        return peak_divide_conquer(arr, i, mid - 1)


def main():
    example1 = [1, 2, 3, 4, 5, 6, 7, 8]
    example2 = [8, 7, 6, 5, 4, 3, 2, 1]
    example3 = [1, 2, 3, 4, 3, 2, 1, 0]
    ls = [example1, example2, example3]
    for l in ls:
        print(f"Peak in {l} is at index: {peak_divide_conquer(l, 0, 7)}")
```

(c) Let $T(n)$ describe the running time of this algorithm on sequences of length n. Assume (for simplicity) that $n = 2^i$ for some $i \geq 1$. Then the running time is well captured by the following recurrence relation:

$$T(n) = T(n/2) + 1 \qquad \text{for } n \geq 1, \text{ and}$$
$$T(1) = 1.$$

What is the solution to this recurrence relation?

$T(n) = T\left(\frac{n}{2}\right) + 1$

$= \left(T\left(\frac{n}{4}\right) + 1\right) + 1 = T\left(\frac{n}{4}\right) + 1 + 1$

$\vdots$

$= T\left(\frac{n}{2^i}\right) + i$

using assumption $n = 2^i \Rightarrow = T(1) + \log_2 n = \log_2 n + 1 = O(\log n)$