Patrick Braun, Class Group 3, Yinyi Xu

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

Homework 2

Instructions: Please submit your solutions via Gradescope by **Friday, 4 February 2022, 10:00am**. 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 2.1: Correctness of the iterative algorithm for the peak problem — 3 pt

Recall the definition of the one-dimensional peak problem from Exercise 1.1 and the iterative algorithm for solving it from Exercise 1.2.

```
1    def peak1d_iterative(A):
2        for i in range(0,len(A)-1):
3            if A[i] > A[i+1]:
4                return i
5        return len(A)-1
```

(a) Formulate a loop invariant for the `for` loop.

(b) Use this loop invariant to argue that `peak1d_iterative()` is correct.

Exercise 2.2: Running time of the iterative algorithm for the peak problem — 3 pt

In this exercise you will examine the worst-case running time of the iterative algorithm from Exercise 1.2 assuming constant cost $c_\ell > 0$ for each line ℓ. Let $n :=$`len(A)` denote the length of the list.

```
1    for i in range(0,len(A)-1):          c1
2        if A[i] > A[i+1]:                c2
3            return i                     c3
4    return len(A)-1                      c4
```

(a) Derive a formula for the best-case running time $T_{best}(n)$.

(b) Derive a formula for the worst-case running time $T_{worst}(n)$.

Hint: Be careful!

Exercise 2.3: Squirrels of the Square Mile — 4 pt

You are consulting for a small computation-intensive investment company, Squirrels of the Square Mile, which has the following type of problem that they want to solve over and over. A typical instance of the problem is the following. They're doing a simulation in which they look at n consecutive days of a given stock, at some point in the past. Let's number the days $i = 1, 2, ..., n$; for each day i, they have a price $p(i)$ per share for the stock on that day. (We'll assume for simplicity that the price was fixed during each day.) Suppose during this time period, they wanted to buy 1000 shares on some day and sell all these shares on some (later) day. They want to know how: When should they have bought and when should they have sold in order to have made as much money as possible?

For example, suppose $n = 3$ and $p(1) = 9$, $p(2) = 1$, $p(3) = 5$. Then the answer would be "buy on day 2, and sell on day 3" as buying on day 2 and selling on day 3 means they would have made £4 per share, the maximum possible for that period.

Clearly, there is a simple algorithm that scales like n^2: try all possible pairs of buy/sell days and see which makes them the most money. Your investment friends were hoping for something a little better.

For simplicity, let's assume we just want to compute the maximum profit per share that can be attained instead of the buy/sell days that achieve this maximum profit.

(a) Describe a divide-and-conquer algorithm (in plain English) that recursively splits a problem of size n into two problems of size $n/2$, and computes a solution to the problem of size n from the solutions to the problems of size $n/2$.

(b) Implement this algorithm in Python.

The running time of this algorithm will scale like $n \cdot \log_2(n)$; which is much better than the running time of the trivial algorithm.

Hint: For the implementation in Python, you may want to start from `peak1d_recursive()` for the general structure. You can use `min(A)` and `max(A)` to find the minimum and maximum of the list `A`.

Exercise 2.1: Correctness of the iterative algorithm for the peak problem **3 pt**

Recall the definition of the one-dimensional peak problem from Exercise 1.1 and the iterative algorithm for solving it from Exercise 1.2.

```
1    def peak1d_iterative(A):
2        for i in range(0,len(A)-1):
3            if A[i] > A[i+1]:
4                return i
5        return len(A)-1
```

(a) Formulate a loop invariant for the `for` loop.

(a) At the beginning of iteration i,

$A[0], A[1] \ldots A[i-1] < A[i]$

(b) Use this loop invariant to argue that `peak1d_iterative()` is correct.

(b) Base Case: At the first iteration i=0 we have that the numbers before A[0] do not contain the peak which is true trivially.

Maintenance: Assume loop invariant is true for specific i. Then $A[0] \ldots A[i-1] < A[i]$

Then if $A[i] > A[i+1]$ then $A[0] \ldots A[i-1] < A[i]$ and decreasing after A[i] so i is the peak and is correct. Otherwise if $A[i] \leq A[i+1]$ then $A[0] \ldots A[i-1] < A[i] < A[i+1]$ so $A[0] \ldots A[i] < A[i+1]$. So maintenance step is correct.

Termination Step. When the for loop terminates

we have that $i = len(A) - 1$. so by definition

$A[0] \ldots A[len(A) - 2] < A[len(A) - 1]$

As there are only $len(A)$ elements this means $A[len(A) - 1]$ is the peak and is correctly returned.

Exercise 2.2: Running time of the iterative algorithm for the peak problem **3 pt**

In this exercise you will examine the worst-case running time of the iterative algorithm from Exercise 1.2 assuming constant cost $c_\ell > 0$ for each line ℓ. Let $n :=$ `len(A)` denote the length of the list.

```
1    for i in range(0,len(A)-1):          c1
2        if A[i] > A[i+1]:                c2
3            return i                     c3
4    return len(A)-1                      c4
```

(a) Derive a formula for the best-case running time $T_{best}(n)$.

(a) In the best case A[0] is the peak.
In this case we have the following runs

```
for i in range(0,len(A)-1):
    if A[i] > A[i+1]:
        return i
return len(A)-1
```

	# of times
c_1	1
c_2	1
c_3	1
c_4	0

So $T_{best}(n) = c_1 + c_2 + c_3$

so $O(1)$

(b) In the worst case, the last element is the peak.
Giving us the following runs.

```
for i in range(0,len(A)-1):
    if A[i] > A[i+1]:
        return i
return len(A)-1
```

	# of times
c_1	n
c_2	n-1
c_3	0
c_4	1

So $T_{worst}(n) = c_1 n + c_2(n-1) + c_4$

$= (c_1 + c_2)n - c_2 + c_4$

so $O(n)$

Exercise 2.3: Squirrels of the Square Mile **4 pt**

You are consulting for a small computation-intensive investment company, Squirrels of the Square Mile, which has the following type of problem that they want to solve over and over. A typical instance of the problem is the following. They're doing a simulation in which they look at n consecutive days of a given stock, at some point in the past. Let's number the days $i = 1, 2, ..., n$; for each day i, they have a price $p(i)$ per share for the stock on that day. (We'll assume for simplicity that the price was fixed during each day.) Suppose during this time period, they wanted to buy 1000 shares on some day and sell all these shares on some (later) day. They want to know how: When should they have bought and when should they have sold in order to have made as much money as possible?

For example, suppose $n = 3$ and $p(1) = 9$, $p(2) = 1$, $p(3) = 5$. Then the answer would be "buy on day 2, and sell on day 3" as buying on day 2 and selling on day 3 means they would have made £4 per share, the maximum possible for that period.

Clearly, there is a simple algorithm that scales like n^2: try all possible pairs of buy/sell days and see which makes them the most money. Your investment friends were hoping for something a little better.

For simplicity, let's assume we just want to compute the maximum profit per share that can be attained instead of the buy/sell days that achieve this maximum profit.

(a) Describe a divide-and-conquer algorithm (in plain English) that recursively splits a problem of size n into two problems of size $n/2$, and computes a solution to the problem of size n from the solutions to the problems of size $n/2$.

(a) Split into 2 subproblems of size n/2.
Call the function recursively. Base case is a list of size 1. In this case, return min=A[0]=max and profit=0.
If not in the base case, calculate current best profit as the maximum of the left and right profits as well as max.-right - min-left.

Also let imin be minimum of the left and right minimums. (same thing for maximum).
Then return min, max, profit.

(b) My code: seems to work correctly.

```python
def find_max_profit(A):
    if len(A) == 1:
        return (A[0], A[0], 0)

    mid = len(A) // 2

    min_left, max_left, profit_left = find_max_profit(A[:mid])
    min_right, max_right, profit_right = find_max_profit(A[mid:])

    current_profit = max(profit_left, profit_right, max_right - min_left)

    current_min = min(min_left, min_right)
    current_max = max(max_left, max_right)

    return (current_min, current_max, current_profit)
```