Exercise Sheet 4: Answers
COMS10018 Algorithms 2024/2025

1 Algorithm Design

Describe an $O(n \log n)$ time algorithm that, given an array A of n integers and another integer x, determines whether or not there are two elements in A whose sum equals x (Hint: Sorting!).

Solution. I will describe two different solutions. *Solution 1* is the solution that I had in mind. During an exercise class in the academic year 2019/2020, a student came up with a simpler and more elegant solution (*Solution 2*)! The advantage of *Solution 1* is that it runs in time $O(n)$ if we are guaranteed that the input array is already sorted, while *Solution 2* requires time $O(n \log n)$ even if the input array is already sorted.

Solution 1. We first sort the array A in time $\Theta(n \log n)$. Assume from now on that A is sorted. Next, we check whether A contains two elements of value $x/2$ in time $\Theta(\log n)$ (using binary search). If there are such elements then we are done. Else, we know that if there is a solution then it consists of two elements x_1, x_2 with $x_1 < x/2$ and $x_2 > x/2$. Let i be the position in array A such that $A[i] < x/2$ and $A[i+1] \geq x/2$. Let $j = i + 1$. Consider now the following loop:

- If $A[i] + A[j] < x$ then add 1 to j.
- If $A[i] + A[j] > x$ then subtract 1 from i.
- If $A[i] + A[j] = x$ then we found a solution and we stop.

We stop this procedure once $i = -1$ or $j = n$ as we then have not found a solution. The runtime of this procedure is clearly $\Theta(n)$, since i and j together "walk" at most a distance of n.

To see why this works, let k_1, k_2 with $k_1 < k_2$ be the indices of a solution, i.e., $A[k_1] + A[k_2] = x$. Observe that, initially, we have

$$k_1 \leq i < j \leq k_2 \ . \tag{1}$$

If the algorithm "misses" the solution k_1, k_2, then there is moment when we updated either i or j and then Inequality 1 is no longer true, i.e., we either updated i to become value $k_1 - 1$ or we updated j to become value $k_2 + 1$.

Suppose first that variable i was updated at this moment. This implies that the algorithm went from the configuration $(i = k_1, j)$ to the configuration $(i = k_1 - 1, j)$. By construction of the algorithm, this only happens if $A[k_1] + A[j] > x$. This however is a contradiction, since $A[k_1] + A[j] \leq A[k_1] + A[k_2] = x$ (since $j \leq k_2$).

Suppose next that variable j was updated at this moment. This implies that the algorithm went from the configuration $(i, j = k_2)$ to the configuration $(i, j = k_2 + 1)$. By construction of the algorithm, this only happens if $A[i] + A[k_2] < x$. This however is a contradication, since $A[i] + A[k_2] > A[k_1] + A[k_2] = x$ (since $i \geq k_1$).

The algorithm therefore cannot miss the configuration (k_1, k_2).

Solution 2. Again, we first sort the array A in $\Theta(n \log n)$ time. Assume from now on that A is sorted. Next, we walk through the array from left to right with a for loop (using variable $i = 0 \dots n-1$). In iteration i, we use a binary search to check whether the array A contains an element with value $x - A[i]$. A binary search takes time $O(\log n)$. Since we do a binary search in each iteration, and there are n iterations at most, the runtime is $O(n \log n)$. This is a very nice and elegant solution. Thanks to the student who came up with it.

✓

2 O-Notation (Difficult)

Prove the following statement:

$$O(\log n) \subseteq O(2^{\sqrt{\log n}}) \subseteq O(n) \ .$$

To this end, identify a value n_0 such that $\log n \le 2^{\sqrt{\log n}} \le n$ holds, for every $n \ge n_0$. While the second of these two inequalities is easy to prove, the first requires an application of the racetrack principle.

Remark: The function $2^{\sqrt{\log n}}$ grows faster than $\log n$ (in fact, faster than any polylogarithm $\log^c n$, for any constant c), but grows slower than n (in fact, slower than any polynomial n^ϵ, for any constant $\epsilon > 0$). The space between polylogarithms and polynomials is therefore non-trivial.

Solution. First, from the definition of Big-O, it follows that (by setting the constants to 1) $O(\log n) \subseteq O(2^{\sqrt{\log n}}) \subseteq O(n)$ holds if we can determine an n_0 such that $\log n \le 2^{\sqrt{\log n}} \le n$ holds, for every $n \ge n_0$.

Next, observe that $\log n = 2^{\log \log n}$ and $n = 2^{\log n}$. It is therefore enough to show that $\log \log n \le \sqrt{\log n} \le \log n$ holds, for every $n \ge n_0$.

We first consider the inequality $\sqrt{\log n} \le \log n$:

$$\begin{aligned} \sqrt{\log n} \le \log n & \text{ is equivalent to} \\ 1 &\le \sqrt{\log n} \\ 1 &\le \log n \\ 2 &\le n \ , \end{aligned}$$

hence, this inequality holds for every $n \ge 2$.

Next, we consider the inequality $\log \log n \le \sqrt{\log n}$. We substitute $\log n$ by $x = \log n$. Then, it is enough to show that $\log x \le \sqrt{x}$, which is equivalent to $\log^2(x) \le x$. We use the racetrack principle to show that this inequality holds for every $x \ge x_0 = 16$. Indeed, first, observe that $\log^2(16) = 16$ so the inequality holds for $x_0 = 16$. It remains to prove that $(\log^2(x))' \le (x)'$ holds for every $x \ge x_0 = 16$. Observe that $(\log^2(x))' = 2\log(x) \cdot \frac{1}{x \ln(2)}$ and $(x)' = 1$. Hence, we need to argue that

$$\begin{aligned} \frac{2 \log x}{x \ln(2)} &\le 1 \ , \text{ which is equivalent to} \\ \log x &\le \frac{x \ln(2)}{2} \end{aligned}$$

holds, for every $x \ge x_0 = 16$. To show this, we use the racetrack principle, again! We first verify that the previous inequality holds for $x = x_0 = 16$. To this end, observe that $\log(16) = 4$

and $16 \ln(2)/2 = 8 \ln(2) \geq 4$ since $\ln(2) \approx 0.693 \geq \frac{1}{2}$. Taking derivatives as required in the racetrack principle, we obtain the condition:

$$\frac{1}{x \ln(2)} \leq \frac{\ln(2)}{2} \text{ , which is equivalent to}$$
$$4.16 \approx \frac{2}{\ln^2(2)} \leq x \ ,$$

which thus holds for every $x \geq x_0 = 16$.

We have thus found a value $x_0 = 16$ such that $\log x \leq \sqrt{x}$. Since $x = \log n$, we have $x_0 = \log n_0$ or $n_0 = 2^{x_0} = 2^{16}$. We can thus pick the value $n_0 = 2^{16}$. ✓

3 Mergesort

The Mergesort algorithm uses the Merge operation, which assumes that the left and the right halves of an array A of length n are already sorted, and merges these two halves so that A is sorted afterwards. The runtime of this operation is $O(n)$.

Suppose that we replaced the Merge operation in our Mergesort algorithm with a less efficient implementation that runs in time $O(n^2)$ (instead of $O(n)$). What is the runtime of our modified Mergesort algorithm?

Solution. Similar to the analysis in the lecture, we sum up the work in each level of the recursion tree. In level i, there are at most 2^{i-1} nodes, and the arrays in level i are of lengths at most $\lceil \frac{n}{2^{i-1}} \rceil$. The runtime in level i on a single node is then $O(\lceil \frac{n}{2^{i-1}} \rceil^2) = O(\frac{n^2}{2^{2(i-1)}})$. We thus obtain:

$$\sum_{i=1}^{\lceil \log n \rceil + 1} 2^{i-1} O(\frac{n^2}{2^{2(i-1)}}) = \sum_{i=1}^{\lceil \log n \rceil + 1} O(\frac{n^2}{2^{i-1}}) = O(n^2) \sum_{i=1}^{\lceil \log n \rceil + 1} \frac{1}{2^{i-1}} \leq O(n^2) \cdot 2 = O(n^2) \ ,$$

where we used the geometric series $\sum_{i=0}^{\infty} \frac{1}{2^i} = 2$.

Observe that, interestingly, the maths show that no $\log n$ factor is introduced here as opposed to the case where the runtime on a single node is $O(n)$. ✓

4 Bubblesort

Bubblesort is a popular, but inefficient, sorting algorithm. It works by repeatedly swapping adjacent elements that are out of order:

Algorithm 1 Bubblesort

Require: Array A of n integers

```
for i = 0 to n − 2 do
  for j = n − 1 downto i + 1 do
    if A[j] < A[j − 1] then
      exchange A[j] with A[j − 1]
    end if
  end for
end for
```

1. What are the worst-case, best-case, and average-case runtimes of Bubblesort?

Solution. We see that the number of times the operations in Lines 3 and 4 are executed is independent of the input, or, in other words, the outer loop always goes from 0 to $n-2$ and the inner loop always goes from $n-1$ downto $i+1$. Hence, the best-case, worst-case, and average-case runtimes of the algorithm are the same.

To analyse the runtime, observe that the operation in Line 4, i.e., exchanging two elements in the array, takes time $O(1)$. The runtime is therefore bounded by the number of times Line 4 is executed. The outer loop goes from $i = 0$ to $n-2$, and the inner loop goes from $j = n-1$ downto $i+1$. We therefore compute:

$$\begin{aligned}
\sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} O(1) &= O(1) \cdot \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = O(1) \cdot \sum_{i=0}^{n-2} ((n-1) - (i+1) + 1) \\
&= O(1) \cdot \sum_{i=0}^{n-2} (n-i-1) = O(1) \cdot \left((n-1)^2 - \sum_{i=0}^{n-2} i \right) \\
&= O(1) \left((n-1)^2 - \underbrace{\frac{(n-2)(n-1)}{2}}_{\leq (n-1)^2/2} \right) \leq O(1) \left((n-1)^2/2\right) \\
&= O(n^2) \ .
\end{aligned}$$

✓

2. Consider the loop in lines $2-6$. Prove that the following invariant holds at the beginning of the loop:
$$A[j] \leq A[k], \text{ for every } k \geq j \ .$$
Give a suitable termination property of the loop.

Solution.

Initialization: We need to show that the property is true prior to the first iteration of the loop. Let $j = n-1$. Then the property translates to $A[n-1] \leq A[k]$ for every $k \geq n-1$. This is trivially true since the only value for k such that $k \geq n-1$ that also lies within the boundaries of the array is $k = n-1$. It is of course true that $A[n-1] \leq A[n-1]$. The property thus holds.

Maintenance: Suppose that the property is true before an iteration j of the loop, i.e., $A[j] \leq A[k]$ holds for every $k \geq j$. We will show that the property also holds before the next iteration. Observe that before the next iteration, the value of j is decreased. We thus need to show that after the current iteration, $A[j-1] \leq A[k]$ holds for every $k \geq j-1$.

Considering the algorithm, there are two cases: Either the if-condition evaluates to true, or it evaluates to false.

Case 1: $A[j] \geq A[j-1]$. (i.e., the if evaluates to false)
In this case nothing happens to the array elements. We need to show that $A[j-1] \leq A[k]$, for every $k \geq j-1$. We already know that $A[j] \leq A[k]$ for every $k \geq j$. Since $A[j-1] \leq A[j]$, the loop invariant is thus also true.

Case 2: $A[j] < A[j-1]$. (i.e., the if evaluates to true)
In this case, $A[j]$ is exchanged with $A[j-1]$. We need to show that after the exchange $A[j-1] \leq A[k]$ for every $k \geq j-1$. Consider thus the state of the array after the

exchange. Concerning $k = j - 1$, this is trivially true (i.e, $A[j-1] \leq A[j-1]$ clearly holds). Concerning $k = j$, this is also true due to the if-statement evaluating to true and the fact that we exchanged the two elements. Concerning all other values of k, i.e., $k \geq j+1$, this follows from the loop invariant being true at the beginning of the iteration.

Termination: We are guaranteed that $A[i] \leq A[k]$, for every $k \geq i$. ✓

3. Consider now the loop in lines $1 - 7$. Prove that the following invariant holds at the beginning of the loop:

   The subarray $A[0, i]$ is sorted and $A[0, i-1]$ consists of the $i-1$ smallest elements of A.

   Give a suitable termination property that shows that A is sorted upon termination.

   Solution.

   Initialization: We need to show that the property is true prior to the first iteration of the loop. At the beginning of the first iteration we have $i = 0$. Then the property translates to "the subarray $A[0 \dots 0]$ is sorted and $A[0, -1]$ consists of the $i-1$ smallest elements of A". This is trivially true, since $A[0 \dots 0] = A[0]$ consists of a single elements, and $A[0 \cdots -1]$ is empty.

   Maintenance: Suppose that the property is true before an iteration i of the loop, i.e., $A[0, \dots, i]$ is sorted and $A[0 \dots i-1]$ are the $i-1$ smallest elements of A. We will show that the property also holds before the next iteration. By the termination property stated in the last exercise, we have that $A[i] \leq A[k]$, for every $k \geq i$, or, in other words, $A[i]$ is the smallest element in $A[i, n-1]$. By the loop invariant, $A[0, \dots, i-1]$ are the $i-1$ smallest elements in increasing order. Hence, the subarray $A[0, \dots, i]$ contains the i smallest elements in A in increasing order. This implies further that the subarray $A[0, i+1]$ is sorted (note that no matter which element is at position $i+1$, the array is sorted).

   Termination: We are guaranteed that A is sorted.

   ✓

5 Optional and Difficult Questions

Exercises in this section are intentionally more difficult and are there to challenge yourself.

5.1 Closest Pair of Points (hard!)

The input consists of two arrays of n real numbers X, Y and represent n points with coordinates $(X[0], Y[0]), (X[1], Y[1]), \dots, (X[n-1], Y[n-1])$. Give a divide-and-conquer algorithm that finds the pair of points that are closest to each other, i.e., the output consists of a two indices i, j such that $(X[i], Y[i])$ and $(X[j], Y[j])$ are the two closest points.