DMFS - Problem Set 2 - resubmission

pwn274

February 2022

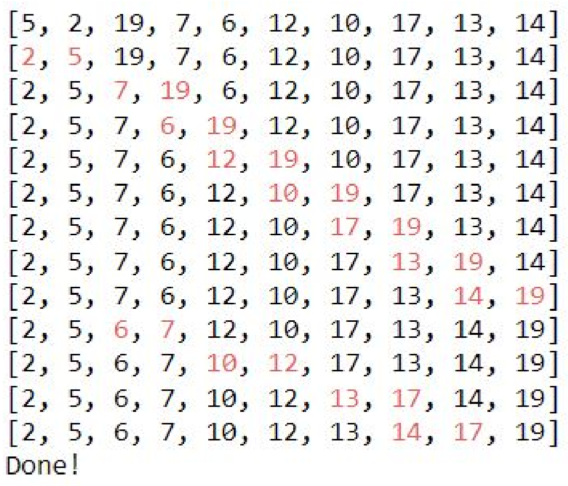
Contents

1

1.1

```
OptimizedBubbleSort (A)
    i := 1
    swapped := true
    while (i <= length(A) and swapped) {
        swapped := false
        for j := 1 upto size(A) - i {
            if (A[j] > A[j + 1]) {
                tmp := A[j]
                A[j] := A[j + 1]
                A[j + 1] := tmp
                swapped := true
            }
        }
        i := i + 1
    }
```

I have sorted it by hand:



```
[5, 2, 19, 7, 6, 12, 10, 17, 13, 14]
[2, 5, 19, 7, 6, 12, 10, 17, 13, 14]
[2, 5, 7, 19, 6, 12, 10, 17, 13, 14]
[2, 5, 7, 6, 19, 12, 10, 17, 13, 14]
[2, 5, 7, 6, 12, 19, 10, 17, 13, 14]
[2, 5, 7, 6, 12, 10, 19, 17, 13, 14]
[2, 5, 7, 6, 12, 10, 17, 19, 13, 14]
[2, 5, 7, 6, 12, 10, 17, 13, 19, 14]
[2, 5, 7, 6, 12, 10, 17, 13, 14, 19]
[2, 5, 6, 7, 12, 10, 17, 13, 14, 19]
[2, 5, 6, 7, 10, 12, 17, 13, 14, 19]
[2, 5, 6, 7, 10, 12, 13, 17, 14, 19]
[2, 5, 6, 7, 10, 12, 13, 14, 17, 19]
Done!
```

Figure 1: I could not get colors to work on numbers in LATEXand my handwriting is horrible, so here is a screenshot. Red entries are compared.

The outer loop makes sure that we can start from every index going until no changes/swappings have been made. Given that the array can be seen as if it is sorted from the right, it is a worst-case $\mathcal{O}(N/2)$, but a best-case $\mathcal{O}(1)$ if the array is already sorted. The inner loop runs will always run through the whole array, making it both best- and worst-case $\mathcal{O}(N)$. Combining these, we see that the worst-case asymptotic time is $\mathcal{O}(n^2)$ while the best-case is $\mathcal{O}(N)$.

For checking correctness, you simply need to realize, that the outer-while loop will only terminate once nothing has been swapped and since it starts from every element from the left and that the largest element always will placed correctly to the right while it locally sort on the way there. So, even in a worst-case scenario, the outer loop will reach the middle, having sorted all the elements.

1.2

I use the Merge and Merge sort pseudocode from the lecture notes. What a horrible waste of time to make us do this by hand on such a large array:

```
The starting array: [5, 2, 19, 7, 6, 12, 10, 17, 13, 14]
Split it into [5, 2, 19, 7, 6] and [12, 10, 17, 13, 14]
Recursively call on [5, 2, 19, 7, 6]
Split [5, 2, 19, 7, 6] into [5, 2, 19] and [7, 6]
Recursively call on [5, 2, 19]
Split [5, 2, 19] into [5, 2] and [19]
Split [5, 2] into [5] and [2]
Merge [5] and [2] into [2, 5]
Merge [2, 5] and [19] into [2, 5, 19]
Split [7, 6] into [7] and [6]
Merge [7] and [6] into [6, 7]
Merge [2, 5, 19] and [6, 7] into [2, 5, 6, 7, 19]
Recursively call on [12, 10, 17, 13, 14]
Split [12, 10, 17, 13, 14] into [12, 10, 17] and [13, 14]
Recursively call on [12, 10, 17]
Split [12, 10, 17] into [12, 10] and [17]
Split [12, 10] into [12] and [10]
Merge [12] and [10] into [10, 12]
Merge [10, 12] and [17] into [10, 12, 17]
Split [13, 14] into [13] and [14]
Merge [13] and [14] into [13, 14]
Merge [10, 12, 17] and [13, 14] into [10, 12, 13, 14, 17]

We will now merge [2, 5, 6, 7, 19] and [10, 12, 13, 14, 17] by making an empty
array M of size {len(MMM)} and defining i = 0 and j = 0.

Now we check L[0](2) <= R[0](10)
It is, so we add 2 to M and increase i to 1.
M is now [2]
Then we check L[1](5) <= R[0](10)
It is, so we add 5 to M and increase i to 2.
M is now [2, 5]
Afterwards we check L[2](6) <= R[0](10)
It is, so we add 6 to M and increase i to 3.
M is now [2, 5, 6]
We check L[3](7) <= R[0](10)
Once again, it is, so we add 7 to M and increase i to 4.
M is now [2, 5, 6, 7]
Finally we check L[4](19) <= R[0](10)
It is not, so we add 10 to M and increase i to 4.
M is now [2, 5, 6, 7, 10]
Then we check L[4](19) <= R[1](12)
Still, it is not, so we add 12 to M and increase i to 4.
M is now [2, 5, 6, 7, 10, 12]
We then check L[4](19) <= R[2](13)
It is not, so we add 13 to M and increase i to 4.
M is now [2, 5, 6, 7, 10, 12, 13]
How far can we go? L[4](19) <= R[3](14)
```

```
It is not, so we add 14 to M and increase i to 4.
M is now [2, 5, 6, 7, 10, 12, 13, 14]
For the last real time, we check L[4](19) <= R[4](17)
It is not, so we add 17 to M and increase i to 4.
M is now [2, 5, 6, 7, 10, 12, 13, 14, 17]
Now we check L[4](19) <= R[5](infinity)
It is, so we add 19 to M and increase i to 5.
M is now [2, 5, 6, 7, 10, 12, 13, 14, 17, 19]
The array is now finally sorted!!
```

1.3

The OptimizedBubbleSort will only require one pass in the outer loop to realize that the list is already sorted, making it run $\mathcal{O}(n)$. MergeSort has no check for any of the 'top-level' 'piles' (better words please), so it will have to check the whole array again, making it even in best case $\mathcal{O}(n \log n)$

1.4

Now we are looking at worst-case for both algorithms, but MergeSort has the same running time of $\mathcal{O}(n \log n)$ while, as I already described, OptimizedBubbleSort has $\mathcal{O}(n^2)$

2

2.1

I will start out by proving, that if all children where to end up with a ball, this will be because they have paired up.
Imagine the opposite: Every child ends up with a ball. As the number of balls per kid are conserved, this means they must have thrown the ball to each other in a circle (well, technically an n-sided polygon). Since all the distances are distinct and they all throw to the closest kid, this means we have $D(k_1, k_2) > D(k_2, k_3) > \cdots > D(k_{n-1}, k_n)$[1] where D is a distance function and k_n is the n'th kid. We now want the last kid to throw it to the first kid again, giving the following contradiction:

$D(k_1, k_2) > \cdots > D(k_n, k_1)$

For this to happen, the distance between the last and the first child will have to be smaller than the first and the second child. This can of course not happen, as in this case, the first child would not have thrown their ball to the second.

Now, we know that the only way for all the children to end up with a ball is for them to pair up, which the proof trivial: There is no way to evenly pair up an uneven number of children - one of them will always end up alone, without a ball, crying!

[1] You will have to excuse my notation, as the $>$-sign is actually overloaded with some $\geq$-meaning. For $D(k_1, k_2) < D(k_2, k_1)$ the distances between the same two children are of course not required to be unique.

2.2

I will prove this by use of induction. The base case here is a single car with enough energy to go a full circle. It is easy to see which car should be chosen as the starting car.[2] Assuming we are able to find the starting car when there is k of them, we can imagine adding a new car giving us $k+1$ total. Taking a car at position n that is reachable by the car before[3] (position $n-1$), we can remove the car in position n giving the car at $n-1$ the extra battery. This changes nothing for the car before the first (at position $n-2$), but if we started with n it would have this new effective charge. As we can now ignore car $n+1$ and we have charge for the full 5km, this means that we are once again restricted to looking at k cars which we, by assumption, know how to solve!

2.3

Once again, I will try to solve this by use of induction. It can easily be seen, that a 2×2 board it trivially solvable as any placement of the block is a solution. This is the base-case. Now, assuming we can solve a board of size $2^k \times 2^k$, I will look at a board-size of $2^{k+1} \times 2^{k+1}$. The detail that need to be realized is the following structure:

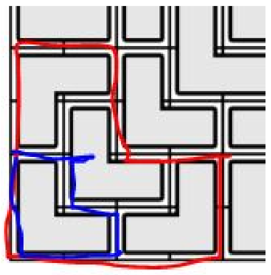


Figure 2: The red shape is exactly twice the size of the blue shape.

Placing the l-shaped pieces in this structure before placing them on the board, effectively halves the size of the problem as the placed pieces are twice the size. This means a $2^{k+1} \times 2^{k+1}$-board can be solved as a $2^k \times 2^k$-board, which we, by assumption, know how to solve.

Since both the base case and the inductive step have been proved as true, by mathematical induction the board is solvable for every k.
QED

[2]not much of a choice, really

[3]As both the distances and the batteries sum to 5km, there will always be a car that is reachable by its neighbor.