DMFS - Problem Set 2

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.

Time complexity is easy as it has two nested loops of size N and N/2 on average making a clear worst-case $\mathcal{O}(n^2)$.

MISSING correctness

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

Given that we have an even amount of children (lets take 50, in our example), all children have a smallest distance to another child. Given that all distances are distinct, one of two things can happen:

- 1) One child has a higher lowest distance than any other children. Here it is trivial to see that this child will throw a ball without receiving one, thereby making the child cry.
- 2) The children are all each others closest neighbor. Here they will pair up and no child will be unhappy.

So, given an even number of kids allows for all the children to end up holding a ball. But given an uneven number of kids, given the fact that all distances are distinct, the second case is not able to happen, as the 51'th kid will either be further from all the other kids or closer to one kid in a pair. Both of these things will make a case 2 to a case 1.

2.2

The best way to look at this problem is to introduce δ_i as the difference between the distance to the next car and the available power. As we know both the distances and the batteries sum to 5 km, we can easily see that the sum over all delta must be 0. Now:

- 1) Imagining the deltas as a list, we can now image starting from the "back car" of any consecutive run of positive or negative δs. This means our problem can be reduced to cars of alternating δs above and below 0. Our list of starting candidates is so far all the first cars in runs of positive δs.
- 2) We can only look at all cars where the deltas when added to the delta in front is 0 or positive. This will always exist, as $\sum_i \delta_i = 0$. After doing this, we have narrowed our list of starting candidates.
- repeat 1) and 2) until there is only two cars left who sum to one with $\delta = 0$. Now we have reached the end!

2.3

Okay this one is kind of easy, once you notice the information that the grid is always of size $2^n \times 2^n$ (which I did not, at first). The beauty of this detail is the following: You can take any four L-pieces and create a new L-piece which is a factor 2 larger as shown here:

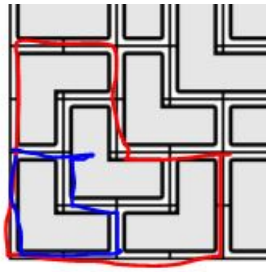

Figure 2: The red shape is exactly twice the size of the blue shape and can always be created.

As this can be done recursively, any $2^n \times 2^n$-board can therefore be seen as a $2^{n-1} \times 2^{n-1}$-board and can be applied recursively. By virtue of the 2×2-board being trivially solvable this is proof that any board is solvable.

Another way this could be shown, is that the following structure us always possible, also halving the board size:

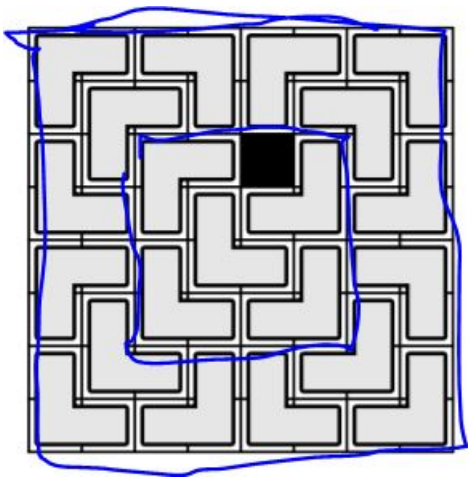

Figure 3: Any $n \times n$ board can always be reduced to a $n/2 \times n/2$ board. As the board sides are always powers of 2, any board can be seen as functioning as a 2×2 grid