KØBENHAVNS UNIVERSITET

Introduktion til diskret matematik og algoritmer: Problem Set 2

Due: Wedneday February 26 at 12:59 CET.
Submission: Please submit your solutions via *Absalon* as a PDF file. State your name and e-mail address close to the top of the first page. Solutions should be written in LaTeX or some other math-aware typesetting system with reasonable margins on all sides (at least 2.5 cm). Please try to be precise and to the point in your solutions and refrain from vague statements. Never, ever just state the answer, but always make sure to explain your reasoning. *Write so that a fellow student of yours can read, understand, and verify your solutions.* In addition to what is stated below, the general rules for problem sets stated on *Absalon* always apply.
Collaboration: Discussions of ideas in groups of two to three people are allowed—and indeed, encouraged—but you should always write up your solutions completely on your own, from start to finish, and you should understand all aspects of them fully. It is not allowed to compose draft solutions together and then continue editing individually, or to share any text, formulas, or pseudocode. Also, no such material may be downloaded from or generated via the internet to be used in draft or final solutions. Submitted solutions will be checked for plagiarism.
Grading: A score of 120 points is guaranteed to be enough to pass this problem set.
Questions: Please do not hesitate to ask the instructor or TAs if any problem statement is unclear, but please make sure to send private messages—sometimes specific enough questions could give away the solution to your fellow students, and we want all of you to benefit from working on, and learning from, the problems. Good luck!

1 In this problem we wish to compare different sorting algorithms.

1a (40 p) The exercises in Chapter 2 of CLRS mention the *bubblesort* algorithm, which can be further optimized as follows:

```
OptimizedBubbleSort (A)
    i       := 1
    swapped := true
    while (i <= size(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
    }
```

Run the optimized bubblesort algorithm by hand on the array

$$A = [5, 2, 19, 7, 6, 12, 10, 17, 13, 14] \tag{1}$$

and show how the elements in the array are moved (similarly to what was done for insertion sort in class). Argue formally why this algorithm is guaranteed to always sort an array correctly. Analyse the time complexity of the algorithm.

Hint: Try to find a nice invariant for the inner while loop to help you argue correctness.

Solution: For the dry-run of the algorithm, let us look at the iterations of the while loop, which we enter for the first time after having set $i = 1$ and `swapped` to true.

1. For $i = 1$, we first flip `swapped` to false and then run the inner for loop for j from 1 to 9. We first compare $A[1] = 5$ with $A[2] = 2$ and swap them, meaning that `swapped` is flipped back to true. For the next comparison we have $A[2] = 5 < A[3] = 19$, so no swap is made. From this point onward, $A[3] = 19$ will be compared to all other elements stored at higher indices in the array, and will be shifted step by step until it reaches index 10, since 19 is the largest element in the array. Once we are done with the for loop, we increment i to 2. At the end of the first iteration the array looks like

$$A^{(1)} = [2, 5, 7, 6, 12, 10, 17, 13, 14, 19] \ . \tag{2}$$

2. Since `swapped` is true and $i = 2$ is less than the length of the array, we enter a second iteration of the while loop in which the inner for loop runs for j from 1 to 8 (after we have assigned `swapped` to false again). In this iteration, we swap places of $A[3] = 7$ and $A[4] = 6$ and of $A[5] = 12$ and $A[6] = 10$ (meaning that `swapped` is also set to true). Then $A[7] = 17$ is compared to $A[8]$ and $A[9]$ and is shifted into position 9, since 17 is the next largest element in the array, after which i is incremented to 3. At the end of the second iteration the array looks like

$$A^{(2)} = [2, 5, 6, 7, 10, 12, 13, 14, 17, 19] \ . \tag{3}$$

3. Since `swapped` is true and $i = 3$ is less than the length of the array, we enter a third iteration of the while loop in which the inner for loop runs for j from 1 to 7. This time, however, we have that $A[j] \leq A[j + 1]$ for all pairwise comparisons, and so `swapped` is never flipped back to true again but stays false. Therefore, the whole algorithm terminates after this iteration.

From this dry-run, we can see that for an array of size n, in the first iteration the largest element in the array is guaranteed to "bubble up" to index n, in the second iteration the next-to-largest element bubbles to index $n - 1$, et cetera. Guided by this, we propose the following invariant:

> *At the start of each iteration of the while loop, the subarray from* $A[n - i + 2]$ *to* $A[n]$ *contains the* $i - 1$ *largest elements in* A *sorted in correct order.*

Note that this invariant is vacuously true in the first iteration for $i = 1$ (since it says that the zero largest elements can be found in an empty part of the array).

In what follows, let us assume for simplicity that all elements in the array are all different, so that there is a unique ith largest element for each $i = 1, \ldots n$. Once we are done with

the analysis, it should be clear that this assumption is not needed, and that all of our argument will go through even if there are duplicates.

Suppose that the invariant is true right when the ith iteration is about to start. Then the $i-1$ largest element are already placed in order in correct position, and the ith largest element resides in some position between 1 and $n-i+1$. During the ith iteration the algorithm will look at all elements $A[1]$, $A[2]$, ..., $A[n-i]$, $A[n-i+1]$, and as soon as the algorithm encounters the ith largest element we see that this element will "win" all comparisons it takes part in and will be shifted all the way to the right, ending up on position $n-i+1$. But this shows that the invariant will also hold at the start of the next iteration. (And if there are duplicates, then one of the largest remaining elements will bubble to position $n-i+1$, and we do not care for which copy this happens.)

It remains to analyse the situation when the algorithm terminates. This can happen either because $i = n+1$ or because `swapped` is false. In the former case, we know from the invariant that the subarray from $A[n-(n+1)+2]$ to $A[n]$ now contains the $(n+1)-1$ largest elements sorted in correct order, which is a somewhat complicated way of saying that the whole array is sorted. In the latter case, we know from the invariant that the subarray from $A[n-i+2]$ to $A[n]$ contains the $i-1$ largest elements sorted in correct order. Moreover, since `swapped` was never set to true during the last iteration, it must be the case that all elements $A[1]$, $A[2]$, ..., $A[n-i+1]$, $A[n-i+2]$ appear in increasing order $A[1] \leq A[2] \leq \ldots \leq A[n-i+1] \leq A[n-i+2]$. (Note that we are using here that i was incremented after the last iteration, so we know that in the last iteration all elements up to $A[n-i+2]$ were looked at). Hence, all of the array must be sorted.

Finally, as to time complexity, the outer while loop can run for at most n times for $i = 1, 2, \ldots, n$, and the inner for loop has i iterations every time. It is not hard to see that this worst-case scenario can arise, for instance, for an array sorted in reverse order. There is a constant number of work being done inside the innermost for loop, and at all other places in the algorithm, so the worst-case running time scales like $\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \Theta(n^2)$.

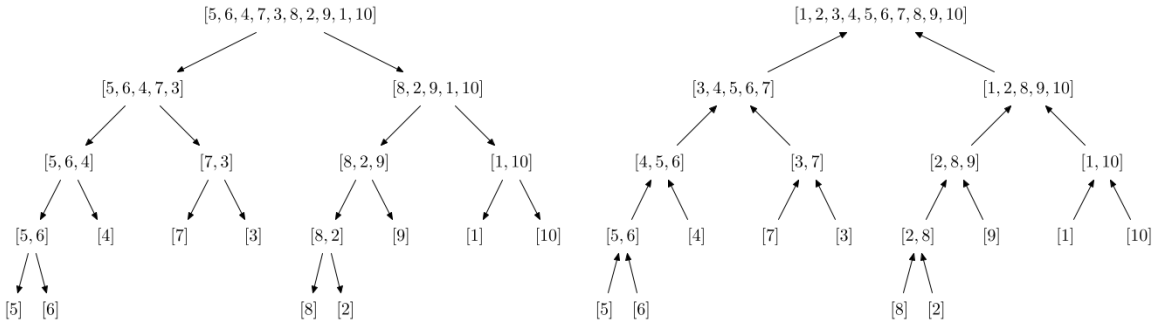
1b (30 p) Run merge sort by hand on the array in (1) (as in the notes for the lectures). Show in every step of the algorithm what recursive calls are made and how the results from these recursive calls are combined, and make sure to explain the final (and most interesting) merge step carefully. (Any clear way of explaining is fine—you do not have to learn how to draw pictures in LaTeX if you do not want to.)

Solution: See Figure 1 for an illustration of what happens in the algorithm.

We first recursively split the list in two part, where the first part is one element larger if the list size is odd, until all lists have size 1. This leads to the split lists in the leaves of the recursion tree in Figure 1a.

In the merge phase we work bottom up from the level above the leaves. Every parent node P takes its two children lists C_1 and C_2, which have previously been sorted, and merges them. We do so by placing to pointers e_1 and e_2 at the start of C_1 and C_2, respectively, and then adding the smallest of these elements to the list under construction after which the corresponding pointer is advanced.

Let us give some examples of how this works, but ignoring the merging of lists of size 1 which is not so exciting (since either the elements are in order or we should just swap them—and, in fact, a more reasonable way to implement merge sort would be to have size ≤ 2 as base case and then swap elements if needed for lists of size 2 rather than making a call to the merge method). For instance, for the two leftmost vertices three levels down we have $e_1 = 2 < 19 = e_2$, so 2 is



(a) Tree of recursive calls splitting list.

(b) Bottom-up sorting of split lists.

Figure 1: Merge sort of array $A = [8, 4, 2, 9, 3, 1, 5, 10, 7, 6]$.

added first and e_1 advances to 5. Since $5 < 19$ we add 5 to the output also. Now C_1 has been emptied, and so we just append C_2 to obtain the list t $[2, 5, 19]$. When $[2, 5, 19]$ is merged with $[6, 7]$, then 2 is the smallest element and goes first, and the next element 5 is also taken from this list. But then 6 and 7 are both smaller than 19, which goes last, and as a result we get the list $[2, 5, 6, 7, 19]$. The recursive calls in the right subtree of the root are dealt with similarly (and this should be described in the solutions, although it is perfectly fine to be a bit brief once you are explaining the merge steps for the third time or so).

In the final step, we need to merge $[2, 5, 6, 7, 19]$ and $[10, 12, 13, 14, 17]$—note that we are specifically asked in the problem statement to explain this merge step carefully, and so we will make sure to do so. Here we start with $e_1 = 2 < 10 = e_2$, so 2 goes first, updating e_1 to 5. Now we still have $e_1 = 5 < 10 = e_2$, so we add 5 to the list under construction, updating e_1 to 6. In the same way we add 6 and 7 from the left-hand side. After this we reach $e_1 = 19$, however, which leads to the whole list of the right-hand side being added, since all elements in this list are smaller than 19. Once the list on the right-hand side has been emptied, 19 is added at the end. This produces the sorted list at the root of the tree in Figure 1b.

1c (20 p) Suppose that we are given another array B of size n that is already sorted in increasing order. How fast do the merge sort and optimized bubblesort algorithms run in this case? Is any of them asymptotically faster than the other as the size of the array B grows?

Solution: Merge sort will be the same, i.e., $O(n \log n)$. The algorithm will always split the input into lists of constant size, which requires a logarithmic number of recursive calls, and merging will always take a linear amount of work per recursion level even if the lists are sorted. (Why?) Another matter is that this might seem overly stupid, and, indeed, there is a *natural merge sort* version of the algorithm that identifies sorted runs in the input and does not split already sorted sublists further. But even without this optimization merge sort is a very efficient algorithm.

Bubblesort just verifies in a single pass through the array in linear time that the array is already sorted, and so will be significantly faster if the input is an array that is already sorted in the correct order.

1d (20 p) Suppose that we are given a third array C of size n that is sorted in *decreasing* order, so that it needs to be reversed to be sorted in the order that we prefer, namely increasing. How fast do the merge sort and optimized bubblesort algorithms run in this case? Is any of them asymptotically faster than the other as the size of the array C grows?

Solution: Merge sort will be the same, i.e., $O(n \log n)$.

Bubblesort will require quadratic time, since the smallest element in the array will only be shifted down by one step in every iteration until it finally reaches the correct index 1 in the very last iteration, and every iteration takes linear time as per the analysis presented above.

2 In 2021 DIKU celebrated its 50th anniversary with a lot of pomp, although slightly less emphasis was given to the fact that it was done one year late due to the Covid pandemic. Even less publicity was given to the public outreach day organized in Fælledsparken for school children by the Algorithms and Complexity Section as part of the anniversary, for reasons that might become clearer after you have studied the problems below.

2a (30 p) In one of the events of the AC Section outreach day, Jakob had arranged so that 51 children[1] were given brightly coloured balls, and were positioned in a field in such a way that all the pairwise distances between the children were distinct. The children were then asked to identify which other child was closest to them and, at a given signal, to throw their ball to this child (and hopefully also receive an incoming ball from somewhere).

This turned out to be a public relations catastrophe. However the children were positioned as described above, every time at least one child ended up without a ball (but instead with tears in the eyes). This did not at all generate the goodwill DIKU was hoping for.

What went wrong? Was Jakob just immensely unlucky? Or can you prove mathematically that it was unavoidable that at least one child would end up without a ball? Would this had been different if Jakob had not insisted on 51 children, but had accepted the proposal by his colleagues to have 50 children? Or if not all distances would have had to be different?

Solution: Firstly, we note that if the distances would not all have had to be different, the children could have arranged themselves at equal distance along a circle and could have thrown their balls clockwise, making everybody happy. Also, if we would have had an even number of children, then they could have been placed in pairs close to each other but far from all other pairs of children, and then each pair of children would just have exchanged balls.

However, if all distances have to be different, and if the total number of children is odd, then there is no way to make everybody happy. Let us prove this by induction for any odd number $n = 2k + 1$ of children.

Base case ($n = 3$)***:*** All pairwise distances between the children are different. The two children with the shorted pairwise distance between them—let us call them Anders and Betina—will throw their balls to each other. The third child, whom we call Christian, will throw his ball to either Anders or Betina, but will not get a ball in return.

Induction step: Our induction hypothesis is that for $n = 2k - 1$ children there is no way to make all children receive a ball.

Suppose now that we have $n + 2 = 2k + 1$ children and consider again the two children Anders and Betina with the shorted pairwise distance, who will throw their balls to each other. We now have two cases:

1. Some other child throws their ball to Anders and Betina. If so, we are done. Clearly, there are now too few balls left for the other children to get one ball each.

[1] Well, because it was a 51st anniversary, after all.

2. Nobody throws a ball to Anders or Betina. If so, this means that we can completely remove Anders and Betina from the picture when considering the rest of the children, meaning that we have exactly the same problem but with $n = 2k - 1$ children. We know by the induction hypothesis that there is no way all of these children will get balls.

It now follows by the induction principle that whenever Jakob's game is played with an odd number of children, at least one child will be left without a ball.

In particular, it was unavoidable that at least one of the 51 children at the DIKU outreach day would be left without a ball.

2b (30 p) In another event, Jakob built a 5-kilometre car track across the park. On this track, 51 electric cars were placed at random locations (but all pointing in the same direction clockwise around the circuit). One car battery was sufficient for exactly one full lap if charged to 100% capacity. However, instead the batteries of all the cars were charged partially in such a way that the total charge of all the batteries together was sufficient for one car to travel exactly the full distance of 5 kilometres. After this, the batteries were distributed to the cars in some random way.

The children were given the challenge to start driving one car in such a way that one full lap of the track would covered. The rules were that if one car travelled far enough to bump into the rear of the car in front, then this next car could continue, and also the battery from the car behind could be shifted to the car in front so that the front car could use any remaining charge (and similarly for any other batteries picked up along the way). If, however, a car would run out of batteries before reaching the car in front, or before the full lap was completed by all the cars together, this was a failure.

This event went slightly better, in that the children were able to figure out a solution to the challenge most of the time. Jakob prided himself with that this was thanks to the fact that he had given the friendly advice, to avoid more embarrassment as in Problem 2a, that a good strategy was to start with the car with the most charge in its battery. Were the children just lucky this time, or can you prove that there is always a solution to this challenge? And is it a good idea to follow Jakob's advice, or does it seem more likely that the children figured out something smarter?

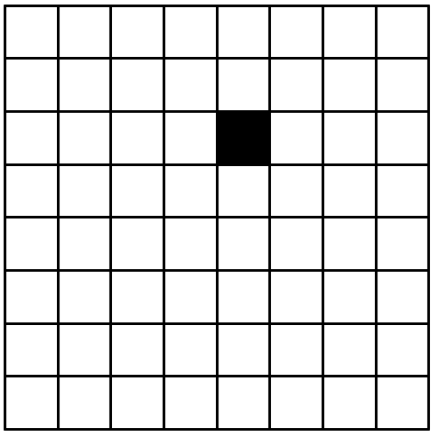
Example: If we for simplicity consider 3 cars on a perfectly circular track, with Car 1 placed at 12 o'clock with a 10% charge, Car 2 placed at 3 o'clock with a 60% charge, and Car 3 placed at 9 o'clock with a 30% charge, then starting with Car 2 is a winning strategy whereas starting with Car 1 or Car 3 leads to failure.

Solution: Jakob is wrong—it is not necessarily a good idea to start with the car with the highest charge. Take the example above, but move Car 3 from 9 o'clock to 1 o'clock. Then starting with Car 2 is a failing strategy.

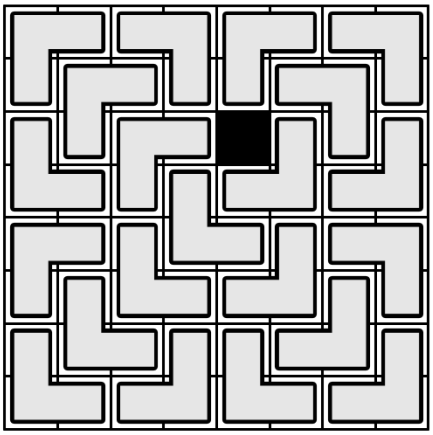
However, it is indeed the case that there is always a solution to this challenge. We will prove this by induction over the number of cars, but we will first make a crucial observation.

There has to exist at least one car that can drive far enough to reach the next car.

Suppose that this is not the case. Go over the cars along the track in clockwise order, and sum up their fractional charges. Since no car can reach the next, each fractional charge is strictly smaller than the distance to the next car measured as a fraction of the total length of the track. This means that the sum of all fractional charges is strictly smaller than the sum of all fractional



(a) Punctured square grid.



(b) Tiling of punctured square grid.

Figure 2: Tiling a punctured 2^n-by-2^n grid with L-shaped tiles (for $n = 3$).

distances along the track. But this latter number is clearly 1, also known as 100%, and the total charge of all batteries is supposed to sum to 100% according to the problem description, and so it is impossible that we would have a strict inequality. This proves the claim in our observation.

We proceed to our proof by induction over the number n of cars.

Base case *($n = 1$):* If we have just one car, then this car has to have a 100% charge and so can drive around the whole track.

Possible second base case *($n = 2$):* Just in case $n = 1$ feels like a weird base case, you can also start with $n = 2$. If so, we note that by our observation above one of the cars can reach the other. Since the sum of the charges of the two batteries is 100%, the second car will be able to complete the whole track.

Induction step: Our induction hypothesis is that the problem is possible to solve for n cars.

Suppose now that we have $n + 1$ cars. By our observation, one of these cars—say car number i—can reach the next car $i + 1$.

But if so, we can mentally consider cars i and $i + 1$ together to be a new super-car with a battery that contains the sum of the charges of the batteries in the two cars and that is placed at the same position as car i. It should be clear that if we can solve the problem with this new super-car, then we can also solve the original problem.

But we see that our new problem is just the problem with exactly the right conditions for n cars, and our induction hypothesis says that there is a solution for any problem with n cars.

It follows by the principle of mathematical induction that the problems is always solvable for any number of cars n (and, in particular, for $n = 51$).

2c (30 p) In the final event of the day, a big 2^n-by-2^n grid was constructed, after which one cell in the grid was removed by placing a black square on it as illustrated in Figure 2a. The children were then given the task to cover all the other cells in the grid by placing L-shaped tiles in such a way that every cell was covered exactly once, and nothing outside of the grid was covered, as shown in Figure 2b.

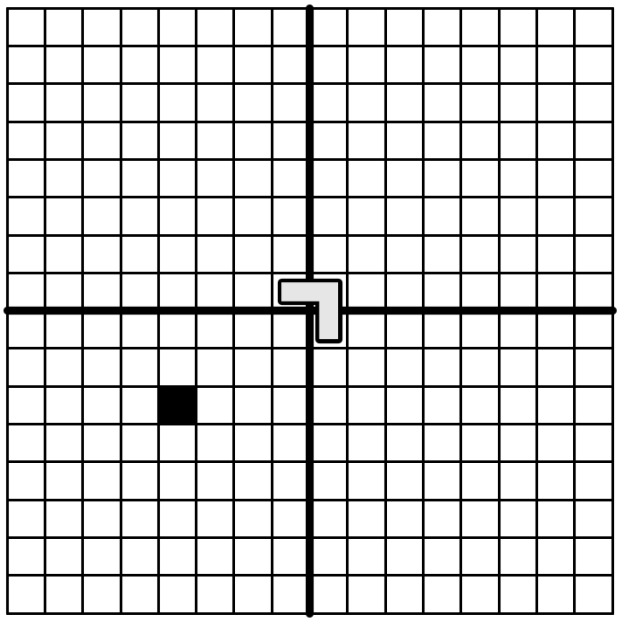


Figure 3: Inductive step in tiling problem.

By now the children were fairly fed up with these strange games, however, and Jakob's colleagues also started getting a bit annoyed, wondering if the strange shape of the tiles was somehow a not-so-subtle attempt to push for a competing foreign university at the other side of the Øresund strait instead, and the day did not end on a festive note at all. Disregarding this unfortunate turn of events, can you prove that it is actually true that for any 2^n-by-2^n grid, regardless of how it is punctured by removing a cell, it is always possible to tile the rest of the grid with L-shaped tiles?

Solution: Let us prove by induction over n that it is always possible to tile a punctured 2^n-by-2^n grid with L-shaped tiles, regardless of how the puncturing is made.

Base case *($n = 1$)**:*** For a 2-by-2 tile it is clear that removing any of the 4 cells will make the remaining 3 cells form an L.

Induction step: Our induction hypothesis is that any punctured 2^n-by-2^n grid can be tiled with L-shaped tiles.

Consider a grid of size 2^{n+1}-by-2^{n+1}, and suppose without loss of generality (because of rotational symmetry) that some cell in the bottom leftmost half of the grid is removed as in Figure 3. Divide the grid up in 4 equal-sized quarters by drawing a horizontal and vertical line after the 2^n first rows and columns, respectively. Place an L-shaped tile where these lines intersect so that the tile touches the 3 non-punctured quarters of the square grid. Now we have 4 punctured subgrids of size 2^n-by-2^n. By the inductive hypothesis, all of these subgrids can be tiled with L-shaped tiles.

It follows by the induction principle that it holds for all positive integers n that any punctured 2^n-by-2^n grid can be tiled with L-shaped tiles.