# DM582 Exercises - Sheet 12

Mads Anker Nielsen  Tobias Samsøe Sørensen

May 12, 2026

This document contains exercises from the course DM582 (spring 2025). Most exercises are from the book *Introduction to Algorithms, 4th edition* by Cormen, Leiserson, Rivest, and Stein (CLRS), the book *Algorithm Design, 1st edition* by J. Kleinberg and E. Tardos (KT), and the book *Discrete Mathematics and its Applications, 8th edition* by K. Rosen.

The solutions given here might differ from the solutions discussed in class. In class, we place more emphasis on the intuition leading to the correct answer. Please do not consider reading these solutions an alternative to attending the exercise classes.

References to CLRS refer to the book *Introduction to Algorithms, 4th edition* by Cormen, Leiserson, Rivest, and Stein.

References to KT refer to the book *Algorithm Design, 1st edition* by J. Kleinberg and E. Tardos.

References to Rosen refer to the book *Discrete Mathematics and its Applications, 8th edition* by K. Rosen.

References to BG refer to the book *Computer Algorithms: Introduction to Design and Analysis, 3rd edition* by Sara Baase and Allen Van Gelder.

This document will inevitably contain mistakes. If you find some, please report them to your TA so that we can correct them.

# Sheet 12

## CLRS Exercise 9.3-1

**Exercise**

In the algorithm `SELECT`, the input elements are divided into groups of 5. Show that the algorithm works in linear time if the input elements are divided into groups of 7 instead of 5.

**Suggested solution**

The analysis here is almost identical. The algorithm still does $\Theta(n)$ work outside of the recursion, noting that we can still sort each group of 7 in constant time. Let $x$ be the median of group medians chosen as the pivot. The change to 7-element groups causes each group with a median of at least $x$ to contain 4 elements that are greater than or equal to $x$. Thus, there are $4\left(\left\lfloor \frac{g}{2} \right\rfloor + 1\right) \geq 2g$ elements that are greater or equal to $x$. Similarly, there are $4\left\lceil \frac{g}{2} \right\rceil \geq 2g$ elements less than or equal to $x$. Since there are $7g$ elements being partitioned, the largest side of any partition contains at most $7g - 2g = 5g \leq \frac{5n}{7}$ elements. So the recursive call on one side of the partition costs no more than $T(\frac{5n}{7})$. The recursive call to find $x$ costs $T(\frac{n}{7})$, so in total, the worst-case running time $T(n)$ is bounded by the recurrence

$$T(n) \leq T\left(\frac{n}{7}\right) + T\left(\frac{5n}{7}\right) + O(n).$$

We show by induction that there exists a constant $c$ such that $T(n) \leq cn$ for all $n > 0$, from which it follows that $T(n) = O(n)$. For the base case, where $n \leq 6$, we can certainly choose $c$ such that $T(n) \leq cn$, since $T(n)$ will be constant for each $n = 1, 2, \ldots, 6$.

Next, consider the inductive step where $n \geq 7$. By the induction hypothesis, $T\left(\frac{n}{7}\right) \leq c \cdot \frac{n}{7}$ and $T\left(\frac{5n}{7}\right) \leq c \cdot \frac{5n}{7}$. Furthermore, the $O(n)$ term in the recurrence is bounded by $c'n$ for some $c'$. If we choose $c \geq 7c'$, then it follows that

$$\begin{aligned} T(n) &\leq c \cdot \frac{n}{7} + c \cdot \frac{5n}{7} + c'n \\ &\leq c \cdot \frac{6}{7}n + c \cdot \frac{1}{7}n \\ &= cn \end{aligned}$$

which is what we wanted.

## CLRS Exercise 9.3-3

**Exercise**

Show how to use `SELECT` as a subroutine to make quicksort run in $O(n \lg n)$ time in the worst case, assuming that all elements are distinct.

**Suggested solution**

We can find the median in linear time with `SELECT`$\left(\left\lceil \frac{n}{2} \right\rceil\right)$ and then partition around it in linear time using `PARTITION-AROUND`. As in the standard `QUICKSORT` procedure, the procedure is then called recursively on both sides of the partition. Since `SELECT` runs in linear time, the work done outside of the recursive calls is still $O(n)$. Furthermore, this approach always produces a maximally balanced partition, and we have seen that quicksort achieves a worst-case running time of $O(n \lg n)$ in this case.

## CLRS Exercise 9.3-6

**Exercise**

You have a "black-box" worst-case linear-time median subroutine. Give a simple, linear-time algorithm that solves the selection problem for an arbitrary order statistic.

**Suggested solution**

Let `SELECT'` denote this new algorithm. The problem is to find the $i$th smallest element in the array $A[p : r]$. We can find the median using the black-box and partition around it, which takes linear time in total. Then we can follow the same procedure as `RANDOMIZED-SELECT` and `SELECT`. Let $q$ be the index of the median after the partition, and let $k = q - p + 1$. If $i = k$, then we can just return the median. If instead $i < k$, then the target element is in the low side of the partition and we can find it by recursively finding the $i$th smallest element in the low side $A[p : q - 1]$. Otherwise, $i > k$ and we can find the $(i - k)$th smallest element in the high side of the partition $A[q + 1 : r]$.

By the choice of the median as the pivot, we know that `SELECT'` is called recursively on an input which contains at most half of the elements in the subarray. Additionally, the work done outside the recursive call is $O(n)$, so the worst-case running time $T(n)$ is bounded by the recurrence

$$T(n) \leq T\left(\frac{n}{2}\right) + O(n).$$

Now, there exists a constant $c$ such that the function represented by the $O(n)$ term is bounded by $cn$. Thus, we can expand the recurrence to get

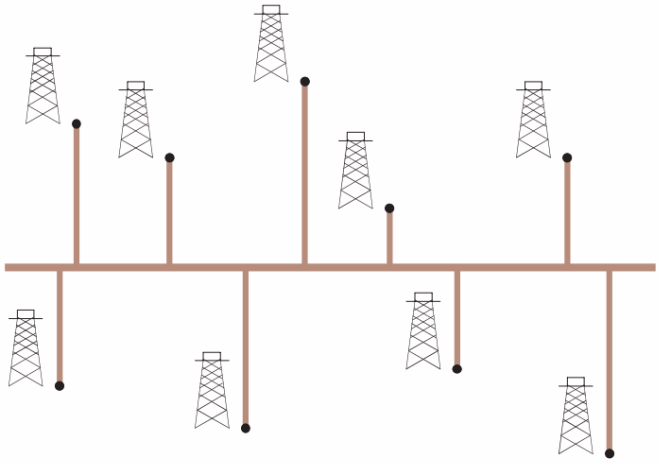
$$T(n) \leq T\left(\frac{n}{2}\right) + cn \leq cn + c\frac{n}{2} + c\frac{n}{4} + c\frac{n}{8} + \cdots + O(1)$$

where the $O(1)$ term denotes the cost of the base case. This is a geometric series and is bounded by $2cn + O(1) = O(n)$, so `SELECT'` runs in linear time.

## CLRS Exercise 9.3-7

**Exercise**

Professor Olay is consulting for an oil company, which is planning a large pipeline running east to west through an oil field of $n$ wells. The company wants to connect a spur pipeline from each well directly to the main pipeline along a shortest route (either north or south), as shown in Figure 9.4. Given the $x$- and $y$-coordinates of the wells, how should the professor pick an optimal location of the main pipeline to minimize the total length of the spurs? Show how to determine an optimal location in linear time. Figure 9.4 shown below.



**Suggested solution**

If $n$ is odd, then the optimal position of the pipeline is at the median (that is, the $\lceil \frac{n}{2} \rceil$th smallest) of all $y$-coordinates of the points. Placing the pipeline here ensures an equal number of points above and below it. Moving the pipeline up or down reduces the distance to $k$ points by some amount while increasing the distance to $k+1$ points by the same amount, so it is optimal.

If $n$ is even, then we have a bit more wiggle room, and can place the pipeline anywhere between the two middle points (which have order statistics $n/2$ and $n/2+1$) since this ensures an equal number of points above and below the pipeline.

In any case, finding the median with `SELECT` gives an optimal location in linear time.

## BG Exercise 5.20

**Exercise**

Consider the problem of determining if a bit string of length $n$ contains two consecutive zeroes. The basic operation is to examine a position in the string to see if it is a 0 or a 1. For each $n = 2, 3, 4, 5$ either give an adversary strategy to force any algorithm to examine every bit, or give an algorithm that solves the problem by examining fewer than $n$ bits.

**Suggested solution**

For $n = 2$, we can construct an adversary strategy that always sets the first examined bit to 0, and the second examined bit to 1 (the second bit could also be set to 0). Since any algorithm will see a 0 as the first examined bit, it must examine the other bit to determine whether there are two consecutive zeroes.

For $n = 3$, we give an adversary strategy. We need to consider all posible actions by an algorithm. An algorithm can choose between which of the three bits to examine first. If the algorithm examines the first bit, the adversary responds with a 1. After this, we notice that we have reduced the problem to the $n = 2$ case, since the 1 bit in the first position doesn't allow for any new 00 pairs compared to $n = 2$. Thus, the algorithm must check the remaining two bits in this case. If the algorithm had instead chosen to examine the third bit first, this case is completely symmetric to examining the first bit first, so it must also check all three bits. Lastly, if the algorithm examines the middle bit, the adversary can respond with a 0, and then set the first and last bit as 1s. This also forces any algorithm to check all three bits.

For $n = 4$, we give an algorithm that examines only two or three bits. The algorithm starts by simply checking the two middle bits. If they are both 0s it halts, having found a pair, and if they are both 1s then we don't need to check the remaining two bits so it also halts in this case. If one of the middle bits is a 1 and the other a 0, then we only need to check the bit adjacent to the known 0 bit, resulting in three checks in total.

For $n = 5$, we give an adversary strategy. First, we argue that any succesful adversary can be modified so that it never creates a 00 pair. If the adversary creates a 00 pair before all bits are checked, then it fails at the task. If the adversary creates a 00 pair when the last bit is checked, it could just as well have responded with a 1 for the last bit, still forcing all bits to be checked. We need to consider only three cases for the first examined bit, since examining bit 1 or 2 first is symmetric to examining bit 5 or 4 first.

1. If the algorithm examines the first bit, the adversary makes it a 0. Now, since we can guarantee that the adversary never makes a 00 pair

anywhere, any algorithm is forced to eventually check the second bit. Therefore, we may assume that the algorithm immediately checks the second bit, and the adversary responds with 1. Thus, there are only three bits remaining, corresponding to the case when $n = 3$, and we can just use that adversary strategy for $n = 3$ which guarantees that the algorithm must check all 5 bits.

2. If the algorithm examines the second bit, the adversary responds with a 0. Then, by the same argument as before, the algorithm is forced to check bits 1 and 3, which are both set to 1 by the adversary. Bits 4 and 5 are then covered by the $n = 2$ case.

3. If the algorithm examines the middle bit first, we set it to a 1. Then we can use use the $n = 2$ strategy for two first and two last bits.

In all cases, any algorithm must examine all 5 bits.

## BG Exercise 5.21

**Exercise**

Suppose you have a computer with a small memory and you are given a sequence of keys in an external file (on a disk or tape). Keys can be read into memory for processing, but no key can be read more than once.

a. What is the minimum number of storage cells needed for keys in memory to find the largest key in the file? Justify your answer.

b. What is the minimum number of cells needed for keys in memory to find the median? Justify your answer.

**Suggested solution**

a. We can use the standard algorithm to find the maximum, which requires only 2 storage cells. One cell is used to store the maximum found so far, and the other is used to load in new values from the file, which are compared to max and swapped if larger. Any algorithm that finds the maximum key must perform some comparisons, and we need to have both keys in memory to compare them, so we need at least 2 cells. Therefore, the algorithm is optimal w.r.t. the number of cells.

b. We give an algorithm that uses $\lceil \frac{n}{2} \rceil + 1$ storage cells. First, allocate an array $A$ of size $\lceil \frac{n}{2} \rceil$ and an additional cell $x$ used for reading in new values. Then, read the first $\lceil \frac{n}{2} \rceil$ keys from the file into $A$. Next, for each key in the file, read the key into $x$, and compare $\max(A)$ with $x$. If $x$ is smaller than $\max(A)$ then replace $\max(A)$ with the value stored in $x$. When this loop terminates, $A$ will contain the $\lceil \frac{n}{2} \rceil$ smallest keys, and the median will be $\max(A)$. To see why, consider a key $k$ among the $\lceil \frac{n}{2} \rceil$ smallest keys, that is compared to $\max(A)$. Now, there are at most $\lceil \frac{n}{2} \rceil - 1$ keys smaller than $k$, so $A$ must contain at least one key larger than $k$, and thus $k$ will be inserted into $A$. Furthermore, the keys removed from $A$ are not among the $\lceil \frac{n}{2} \rceil$ smallest, so $k$ will not be removed.

   For an algorithm that uses fewer storage cells, it won't be able to keep track of half of the keys, and so an adversary would, at least intuitively, be able to place the median among the discarded keys.

## CLRS Exercise 9.3-5

**Exercise**

Show how to determine the median of a 5-element set using only 6 comparisons.

**Suggested solution**

Let $x[1:5]$ be a 5-element array. The following procedure finds the median with 6 comparisons.

1. First, compare $x[1]$ with $x[2]$ and swap them so the smallest is in $x[1]$.
2. Compare $x[3]$ with $x[4]$ and swap them so the smallest is in $x[3]$.
3. Compare $x[1]$ with $x[3]$ and discard the smallest, since it is known to smaller than 3 elements, and thus cannot be the median. Assume $x[1] < x[3]$, so $x[1]$ is discarded, the other case being symmetric.
4. Compare $x[2]$ with $x[5]$ and swap them so the smallest is in $x[5]$.
5. Compare $x[5]$ with $x[3]$ and discard the smaller. Assume $x[5] < x[3]$, the other case again being symmetric.
6. Compare $x[2]$ with $x[3]$, and the median is then $\min(x[2], x[3])$.

This works by first discarding 2 elements that are known to be smaller than 3 other elements. Then we have $x[2]$, $x[3]$ and $x[4]$ remaining and we know $x[3] < x[4]$. Thus, if $x[2] < x[3]$ then $x[2]$ is smaller than 2 elements and must be the median, and conversely if $x[3] < x[2]$ then $x[3]$ is smaller than 2 elements and must be the median.