DM582 Exercises - Sheet 6

Mads Anker Nielsen Tobias Samsøe Sørensen

March 16, 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 6

CLRS, 7-1 (a-b)

Exercise

The version of **PARTITION** given in this chapter is not the original partitioning algorithm. Here is the original partitioning algorithm, which is due to C. A. R. Hoare.

```
HOARE-PARTITION(A, p, r)
 1  x = A[p]
 2  i = p − 1
 3  j = r + 1
 4  while TRUE
 5      repeat
 6          j = j − 1
 7      until A[j] ≤ x
 8      repeat
 9          i = i + 1
10      until A[i] ≥ x
11      if i < j
12          exchange A[i] with A[j]
13      else return j
```

a. Demonstrate the operation of **HOARE-PARTITION** on the array $A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21 \rangle$, showing the values of the array and the indices i and j after each iteration of the while loop.

b. Describe how the **PARTITION** procedure in Section 7.1 differs from **HOARE-PARTITION** when all elements in $A[p : r]$ are equal. Describe a practical advantage of **HOARE-PARTITION** over **PARTITION** for use in quicksort.

Suggested solution

a. See figure 1.

b. The **PARTITION** procedure in Section 7.1 results in an unbalanced partition with all elements on the low side when the elements in $A[p : r]$ are equal. This invokes the worst-case running time of quicksort

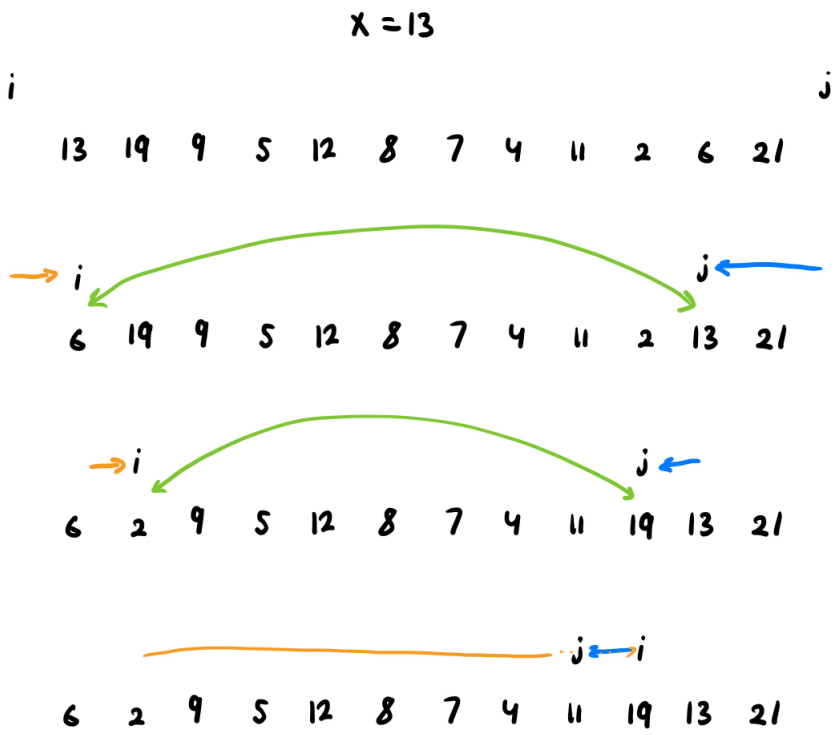

Figure 1

of $\Theta(n^2)$ where n is the length of the subararay. The **HOARE-PARTITION** procedure, on the other hand, results in a balanced partition in this case.

CLRS, Problem 7-2

Exercise

The analysis of the expected running time of randomized quicksort in Section 7.4.2 assumes that all element values are distinct. This problem examines what happens when they are not.

a. Suppose that all element values are equal. What is randomized quicksort's running time in this case?

b. The `PARTITION` procedure returns an index q such that each element of $A[p : q-1]$ is less than or equal to $A[q]$ and each element of $A[q+1 : r]$ is greater than $A[q]$. Modify the `PARTITION` procedure to produce a procedure `PARTITION'` (A, p, r), which permutes the elements of $A[p : r]$ and returns two indices q and t, where $p \le q \le t \le r$, such that

- all elements of $A[q : t]$ are equal,
- each element of $A[p : q-1]$ is less than $A[q]$, and
- each element of $A[t+1 : r]$ is greater than $A[q]$.

Like `PARTITION`, your `PARTITION'` procedure should take $O(r-p)$ time.

c. Modify the `RANDOMIZED-PARTITION` procedure to call `PARTITION'`, and name the new procedure `RANDOMIZED-PARTITION'`. Then modify the `QUICKSORT` procedure to produce a procedure `QUICKSORT'`(A, p, r) that calls `RANDOMIZED-PARTITION'` and recurses only on partitions where elements are not known to be equal to each other.

d. Using `QUICKSORT'`, adjust the analysis in Section 7.4.2 to avoid the assumption that all elements are distinct.

Suggested solution

a. The random swapping operation that distinguishes randomized quicksort from the deterministic version does not change the array. Thus, the algorithm reduces to the deterministic version, which has a running time of $\Theta(n^2)$ when all elements are equal.

b. There are many ways to accomplish this. Below is one suggestion.

```
1  x = A[r]
2  q = t = p − 1
3  for j = p to r − 1
4        y = A[j]
5        if y ≤ x
6              t = t + 1
7              swap A[t] and A[j]
8        if y < x
9              q = q + 1
10             swap A[q] and A[t]
11 swap A[t + 1] and A[r]
12 return (q + 1, t + 1)
```

c. See Figure 2.

```
RANDOMIZED-PARTITION'(A, p, r)
1  i = RANDOM(p, r)
2  exchange A[r] with A[i]
3  return PARTITION'(A, p, r)

RANDOMIZED-QUICKSORT'(A, p, r)
1  if p < r
2       q, t = RANDOMIZED-PARTITION'(A, p, r)
3       RANDOMIZED-QUICKSORT'(A, p, q − 1)
4       RANDOMIZED-QUICKSORT'(A, t + 1, r)
```

Figure 2

d. In the proof of lemma 7.2, we use that if the pivot x chosen in the set $Z_{i,j}$ is not z_i nor z_j, then $z_j < x < z_i$ and thus z_j and z_i end up in different parts of the partition and are thus never compared. For the modified algorithm `QUICKSORT'`, this assertion is still true even if we only assume $z_j \leq x \leq z_i$; if either inequality holds with equality, say

$z_j = x$, then we do not recurse on a subarray containing z_j, and thus z_j is never compared to z_i (similarly if $z_i = x$ or both).

Thus, Lemma 7.2 still holds and the rest of the analysis is identical except we replace $z_1 < z_2 < \cdots < z_n$ with $z_1 \leq z_2 \leq \cdots \leq z_n$ in the proofs of Lemma 7.3 and Theorem 7.4.[1]

[1] We are technically comparing the pivot with each element in the subarray twice in the implementation of `PARTITION'` given here, but this makes no difference to the asymptotic runtime.

CLRS, Problem 7-4

Exercise

Professors Howard, Fine, and Howard have proposed a deceptively simple sorting algorithm, named `stooge-sort` in their honor, appearing on the following page.

```
STOOGE-SORT(A, p, r)
1  if A[p] > A[r]
2      exchange A[p] with A[r]
3  if p + 1 < r
4      k = ⌊(r − p + 1)/3⌋          // round down
5      STOOGE-SORT(A, p, r − k)     // first two-thirds
6      STOOGE-SORT(A, p + k, r)     // last two-thirds
7      STOOGE-SORT(A, p, r − k)     // first two-thirds again
```

a. Argue that the call `STOOGE-SORT(A, 1, n)` correctly sorts the array $A[1 : n]$.

b. Give a recurrence for the worst-case running time of `STOOGE-SORT` and a tight asymptotic (Θ-notation) bound on the worst-case running time.

c. Compare the worst-case running time of `STOOGE-SORT` with that of insertion sort, merge sort, heapsort, and quicksort. Do the professors deserve tenure?

Suggested solution

a. We argue (semi-formally) by induction on the length of the subarray $A[p : r]$ which we denote n. For the sake of simplicity, assume that the elements of the array are distinct.

For $n \leq 2$ the algorithm correctly sorts the array in the first if statement and terminates.

Suppose $n \geq 3$ and let $k = \lfloor (r - p + 1)/3 \rfloor$. All recursive calls are on subarrays with fewer elements, and thus the subarrays are correctly sorted by the induction hypothesis. Suppose some element x is among the largest k elements of the subarray $A[p : r]$. Then x is also among the largest k elements of the subarray $A[p : r - k]$. Thus, after sorting $A[p, r - k]$, x is in the subarray $A[p + k : r]$. Hence, when $A[p + k : r]$ is sorted, x is among the last k element of $A[p : r]$. Since this holds for any x among the largest k elements of $A[p : r]$, we conclude that

the last k elements of $A[p:r]$ are the k largest elements of $A[p:r]$ in sorted order. Thus, sorting $A[p, r-k]$ after the first two recursive calls completely sorts the array.

b. The algorithm perform a constant amount of work and 3 recursive calls on subarrays of size $2/3n$ (ignoring the rounding). Thus,

$$T(n) = 3T((2/3)n) + c$$

describes the running time of the algorithm where c is a constant. This can be solved using the master theorem. Case 1 applies and thus $T(n) \in \Theta(n^{\log_{3/2} 3})$. Note that $\log_{3/2} 3 \approx 2.7$.

c. All of these algorithms have worst-case running time at most $O(n^2)$. The professors might deserve tenure, but probably not because of this algorithm.

CLRS, 9.2-1

Exercise

Show that `RANDOMIZED-SELECT` never makes a recursive call to a 0-length array.

Suggested solution

Suppose we make a recursive call to a 0-length array. We show that the condition $1 \leq i \leq r - p + 1$ is violated. If the call to a 0-length subarray happens on line 8, then we must have $q = p$ and $i < k$. But $k = q - p + 1 = 1$ and thus $i < 1$; a contradiction. Thus, the recursive call to a 0-length array must occur on line 9. Then we must have $q = r$ and $i > k$. But $k = r - p + 1$ and thus $i > r - p + 1$; a contradiction.

CLRS, Problem 9-1

Exercise

You are given a set of n numbers, and you wish to find the i largest in sorted order using a comparison-based algorithm. Describe the algorithm that implements each of the following methods with the best asymptotic worst-case running time, and analyze the running times of the algorithms in terms of n and i.

1. Sort the numbers, and list the i largest.
2. Build a max-priority queue from the numbers, and call `EXTRACT-MAX` i times.
3. Use an order-statistic algorithm to find the i-th largest number, partition around that number, and sort the i largest numbers.

Suggested solution

Using `Merge-Sort` guarantees a running time of $\Theta(n \log n)$ for this approach.

The `Build-Max-Heap` procedure runs in time $\Theta(n)$ and the `Max-Heap-Extract` in time $\Theta(\log n)$ time. We need to call `Build-Max-Heap` once and `Max-Heap-Extract` i times, so the total running time of this approach is $\Theta(n + i \log n)$.

Using `Randomized-Select` for finding the i-th largest element and `Merge-Sort` for sorting, the running time of this approach is $\Theta(n+i \log i)$ in expectation.

Exercise from course webpage

Exercise

Assume that for an oral exam, there are k questions that students can draw from. ~~Kim~~ The hypothetical lecturer finds it tiring to hear the same topic several times, so when a student draws a question, he doesn't put it back again for the next student. Thus, when the first student in the exam sequence draws, there are k questions laid out on the table, when the second student draws, there are only $k-1$ questions on the table, etc. The ith student sees a table with $k-i+1$ questions. At some point, the lecturer restarts the process with all k questions.

The students employ different algorithms for selecting; some try to choose uniformly at random, some take the nearest, some the one furthest away, etc. The lecturer, on the other hand, places the questions in an order chosen uniformly at random, and either does not change the order in between questions disappearing, or sometimes collects all the questions currently on the table and places them uniformly at random again.

Are the students treated fairly in the sense that the ith student gets any of the questions with probability $1/k$, i.e., independent of the person's placement in the exam sequence?

Suggested solution

Yes. First, observe that when the lecturer restarts the process, then the first student after the restart sees a random permutation of all k questions, and thus gets any particular question with probability $1/k$ (regardless of their strategy).

The probability that a particular question x is left on the table when the i-th student draws is the probability that none of the first $i-1$ students drew question x. Since each student draws a question independently and uniformly at random, this probability is

$$\frac{k-1}{k}\frac{k-2}{k-1}\cdots\frac{k-i+1}{k-i+2}=\frac{k-i+1}{k}.$$

Since student i draws uniformly at random from the set of $k-i+1$ questions that are left, the probability that question x is drawn is

$$\frac{1}{k-i+1}\frac{k-i+1}{k}=\frac{1}{k}.$$

CLRS, 7.4-5

Exercise

Coarsening the recursion, as we did in Problem 2-1 for merge sort, is a common way to improve the running time of quicksort in practice. We modify the base case of the recursion so that if the array has fewer than k elements, the subarray is sorted by insertion sort, rather than by continued recursive calls to quicksort. Argue that the randomized version of this sorting algorithm runs in $O(nk + n\log(n/k))$ expected time. How should you pick k, both in theory and in practice?

Suggested solution

Loosely, if quicksort stops when reaching subarrays of size k, then the expected depth of the recursion tree is $\log n/k$, and thus we get and expected running time of $O(n \log n/k)$. Insertion sort is run on the n/k unsorted subarrays of size k. Since insertion sort runs in time $O(k^2)$ on an array of size k, the total contribution to the running time from the calls to insertion sort is $O(n/k \cdot k^2) = O(nk)$. In conclusion, the total running time in $O(nk + n \log n/k)$.

Say the actual running time of the algorithm is $f(n,k) = nk \cdot n\log_2(n/k)$. In theory, we should pick k such that $f(n,k)$ is minimized by solving $\frac{d}{dk}f(n,k) = 0$ for k. In practice, this would probably not be a feasible approach as it requires knowing the exact runtime of the algorithm as a function of k and n. Instead, we are better off determining k experimentally to account for factors such as cache size, memory access times, memory handling by the operating system and hardware, etc. which are hard if not impossible to determine analytically.