Algorithms Example Sheet 1: Core Questions

Exercise 1.C.1 [Check if array is sorted]

(a) Design an algorithm to check if a given array of n elements is sorted.

(b) What is the time complexity of your algorithm? Can you do it more efficiently?

(c) **(I)**Implement your algorithm.

(d) **(I)**Generate some random arrays and use these (together with your sorting check method) to test the sorting algorithms that you will implement (in the coming exercises).

Exercise 1.C.2 [Insertion Sort (B)]

(a) Explain how *insertion sort* works, intuitively.

(b) Illustrate the operation of insertion sort on the array $[31; 41; 59; 26; 41; 58]$.

(c) Write pseudocode for insertion sort.

(d) Argue informally why insertion sort works.

(e) What is the worst-case running time for insertion sort? Can you give an example for this? Why do we prefer examples to be general? (See **[2010P1Q5 (b)]**)

(f) What is the best-case running time for insertion sort?

(g) How can you reduce the number of swaps performed by the algorithm?

[Exercise 1 in Lecturer's handout]

Exercise 1.C.3 [Insertion Sort (I)] Implement the insertion sort algorithm. You can test your implementation on **[LeetCode 912]** (or on **[GeeksForGeeks Insertion Sort]**, or using the sorting benchmark that you created in Exercise 1).

Exercise 1.C.4 [Comparison-based sorting]

(a) What do we mean by a *comparison-based* sorting algorithm?

(b) Explain how you can use any comparison-based sorting algorithm to sort an array in decreasing order (instead of increasing).

Exercise 1.C.5 [Reasoning about correctness of algorithms]

(a) What is a *precondition*?

(b) What is a *postcondition*?

(c) What is an *invariant*? How is it used to prove the correctness of an algorithm?

(d) In which areas of software engineering are preconditions and postconditions useful?

Chapter 2.3

Recommended reading:

- Chapter 3, CLRS
- Handout from Jeff Erickson's book (includes various techniques for solving recurrence relations, most of which are outside the scope of the course)

Exercise 1.C.6 [Asymptotic running times (Introduction - OCaml)]

(a) (Open-ended) Why is the big-$\mathcal{O}$ notation used? Can we not just do empirical computations of the running time?

(b) Use the following method to measure the execution of some code.

```
let measure () =
   let t = Sys.time() in
      (* The code you want to measure *); Printf.printf "%f\n" (Sys.time()
          -. t);;
```

Measure the time it takes to execute `sillySum` (FoCS section 2.5) for values $n = 20$ to $n = 30$. Plot the graph vs input size and compare with the theoretical running time.

(c) What assumptions do we usually make when computing the runtime efficiency of an algorithm?

(d) What is the time complexity of the naive and efficient power function (in FoCS)?

Exercise 1.C.7 [Asymptotic running times (B)]

(a) Why is the Θ, $\mathcal{O}$ and Ω notation useful?

(b) How are each of these formally defined?

(c) In the definition of big-$\mathcal{O}$ notation, what would happen if:

i. we replaced $\forall n > N$ with $\forall n$?

ii. we replaced $\forall n > N$ with $\forall n \geq N$?

iii. we replaced $k \cdot g(n)$ with $g(n)$?

iv. we replaced $k \cdot g(n)$ with $k^2 \cdot g(n)$?

v. we replaced the condition $0 \leq f(n) \leq k \cdot g(n)$ with $f(n) \leq k \cdot g(n)$.

(d) What do we mean by $\mathcal{O}(f(n))$ providing an upper bound on the running time of the algorithm? Is every upper bound useful?

(e) What do we mean by Ω providing a lower bound on the running time of the algorithm? Is any lower bound useful?

(f) How are $o(\cdot)$ and $\omega(\cdot)$ defined?

(g) Prove or disprove $f(n) \in \mathcal{O}(g(n))$ and $f(n) \in \Omega(g(n))$ iff $f(n) = \Theta(g(n))$.

(h) Write down an incorrect definition for $o(n)$ by taking the definition of $\mathcal{O}(n)$ and replacing $\leq$ by $<$. Then find values for k and N that, by this definition, would allow us to claim that $f(3n^2) \in o(n^2)$.

[Exercise 3 in Lecturer's handout]

(i) Let $T_A(n)$ is the running time of Algorithm A and $T_B(n)$ is the running time of algorithm B. If $T_A(n) \in \mathcal{O}(T_B(n))$, would we always prefer A over B in practice? Can you construct a specific example where this is not the case (at least in 2021)?

Exercise 1.C.8 [In-place sorting (B)] What does it mean for a sorting algorithm to be *in-place*? Which of the sorting algorithms that you have covered are in-place? Why is this a desirable feature?

Chapter 2.4 & 2.5

Exercise 1.C.9 [Asymptotic Stirling's formula]

(a) Show that $n! > (n/2)^{n/2}$.

(b) Show that $n! < n^n$.

(c) Deduce that $\log n! = \Theta(n \log n)$.

(d) (optional +) Actually, something stronger holds. Read about Stirling's approximation.

Exercise 1.C.10 [Comparison-based sorting lower bound (B)] Derive the $\Omega(n \log n)$ lower bound for any comparison-based sorting algorithm (use the result from Exercise 9).

Exercise 1.C.11 [Linear Search]

(a) Describe a comparison-based algorithm for finding a given element in an unsorted array.

(b) What is the time complexity of your algorithm? Prove a corresponding lower bound.

Chapter 2.6

Exercise 1.C.12 [Selection Sort (B)]

(a) Intuitively explain how *selection sort* works.

(b) Illustrate the operation of selection sort on the array $[31; 41; 59; 26; 41; 58]$.

(c) Write pseudocode for selection sort. (See **[2010P1Q2 (b)]**)

(d) Argue informally why selection sort works.

(e) What is the worst-case running time for selection sort?

(f) What is the best-case running time for selection sort? Would you choose insertion sort over selection sort?

(g) When looking for the minimum of m items, every time one of the $m-1$ comparisons fails the best-so-far minimum must be updated. Give a permutation of the numbers from 1 to 7 that, if fed to the `selectsort` algorithm, maximizes the number of times that the above-mentioned comparison fails?

[Exercise 7 in Lecturer's handout]

Exercise 1.C.13 [Selection Sort (I)] Implement the insertion sort algorithm. You can test your implementation in problem **[LeetCode 912]** (or **[GeeksForGeeks Selection Sort]**).

Binary search

Recommended reading:

- Read this article on problem solving using binary search.

Exercise 1.C.14 [Binary Search (B)]

(a) Describe the *binary search algorithm.*

(b) Derive the worst-case time complexity for binary search, assuming the array size is a power of 2.

(c) Show that this is the best possible worst-case time complexity for a comparison-based algorithm. (See **[2006P1Q11 (b)]**)

Exercise 1.C.15 [Binary Search (I)]

(a) Implement the binary search algorithm to solve **[LeetCode 704]**.

(b) Given an array consisting of 0s and 1s (in sorted order),

i. Write a binary search program to find the last 0 (or fail if no 0 exists).

ii. Write a binary search program to find the first 1.

You can test your code on **[LeetCode 34]**.

Chapter 2.7

Exercise 1.C.16 [Binary Insertion Sort (B)]

(a) Explain the modification to insertion sort, to efficiently find the correct placement for the current element.

(b) Does this modification improve the time complexity of the algorithm?

(c) Provide pseudocode for the missing part of the code.

Chapter 2.8

Exercise 1.C.17 [Bubble Sort (B)]

(a) Intuitively explain how *bubble sort* works.

(b) Illustrate the operation of bubble sort on the array $[31; 41; 59; 26; 41; 58]$.

(c) Write pseudocode for bubble sort.

(d) Argue informally why bubble sort works.

(e) What is the worst-case running time for bubble sort? Give a general example for when this occurs.

(f) What is the best-case running time for bubble sort? Would you choose insertion sort over bubble sort?

(g) Consider the smallest ("lightest") and largest ("heaviest") key in the input. If they both start halfway through the array, will they take the same time to reach their final position or will one be faster? In the latter case, which one, and why?

Exercise 1.C.18 [Bubble Sort (I)] Implement the bubble sort algorithm. You can test your implementation in problem **[LeetCode 912]** (or **[GeeksForGeeks Bubble sort]**).

Chapter 2.9

Exercise 1.C.19 [Merge Sort (B)]

(a) Intuitively explain how *merge sort* works.

(b) Illustrate the operation of merge sort on the array $[9; 3; 6; 2; 4; 1; 5]$. (See **[2010P1Q5 (a)]**)

(c) Write pseudocode for merge sort.

(d) Argue informally why merge sort works.

(e) What is the worst-case running time for merge sort? Derive this by solving the recurrence relation. You may assume that the array length is a power of 2.

(f) What is the best-case running time for merge sort? Would you choose merge sort over bubble sort?

(g) Can you spot any problems with the suggestion of replacing the somewhat mysterious line `a3[i3] = smallest(a1, i1, a2, i2)` with the more explicit and obvious `a3[i3] = min(a1[i1], a2[i2])`? What would be your preferred way of solving such problems? If you prefer to leave that line as it is, how would you implement the procedure smallest it calls? What are the trade-offs between your chosen method and any alternatives?

[Exercise 11 in Lecturer's handout]

(h) In one line we return the same array we received from the caller, while in another we return a new array created within the mergesort subroutine. This asymmetry is suspicious. Discuss potential problems.

[Exercise 12 in Lecturer's handout]

Exercise 1.C.20 [Merge Sort (I)] Implement the merge sort algorithm. You can test your implementation in problem **[LeetCode 912]** (or **[GeeksForGeeks MergeSort]**).

Chapter 2.10

Recommended reading:

- Chapter 3, CLRS

Exercise 1.C.21 [Almost full binary trees (B)]

(a) What is a *binary tree*? What is the *root* of a binary tree?

(b) What is a *full binary tree*? What is an *almost-full binary tree*?

(c) What are the minimum and maximum number of elements in an almost-full binary tree of height h? Provide an example for minimum and maximum.

(d) Explain how you would store an almost full binary tree in an array. Provide an example.

(e) In this array representation, explain how to access the parent of node with array index i.

Exercise 1.C.22 [Binary heaps (B/P)]

(a) What is a *binary heap*? Give examples of binary trees that are not binary heaps. Can a binary search tree be a binary heap?

(b) Explain how the heapify operation is used to maintain the properties of a binary heap once its root is removed. How many operations does it perform?

(c) Explain why heapifying an entire array takes $\mathcal{O}(n \log n)$ time, where n is the number of elements in the array.

(d) (+) In this sub-question, we will tighten the bound to $\mathcal{O}(n)$.

i. Explain (or prove) why for $|x| < 1$, we have $F(x) := \sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$.

ii. Let $G(x) := \sum_{i=0}^{\infty} i \cdot x^i$. Show that $\frac{d}{dx}F(x) = \sum_{i=0}^{\infty} i \cdot x^{i-1}$ (what assumption are you making?) and that $G(x) = x \cdot \frac{d}{dx}F(x)$.

iii. Deduce that $G(x) = \frac{x}{(1-x)^2}$.

iv. By analysing the recurrence relation for heapifying an array, show that it takes $\mathcal{O}(n)$ time.

Exercise 1.C.23 [Heap Sort (B)]

(a) Intuitively explain how *heap sort* works.

(b) (optional) Illustrate the operation of heap sort on the array $[31; 41; 59; 26; 41; 58]$.

(c) Write pseudocode for heap sort.

(d) Argue informally why heap sort works.

(e) What is the worst-case running time for heap sort? Use the previous exercises.

(f) What is the best-case running time for heap sort? How does heap sort compare to merge sort?

(g) Explain how heap sort relates to selection sort (See **[2006P1Q12 (b)]**)

(h) (optional) Attempt **[2006P6Q1]**.

Exercise 1.C.24 [Heap Sort (I)] Implement the heap sort algorithm. You can test your implementation in problem **[LeetCode 912]** (or **[GeeksForGeeks Heap Sort]**).

Chapter 2.11/2.12

Exercise 1.C.25 [Quicksort (B)]

(a) Describe how Quicksort works, using the following array of numbers as an example: $16, 42, 22, 7, 15, 3$. (See **[2007P1Q11 (a)]**)

(b) Write pseudocode for Quicksort.

(c) Argue informally why Quicksort works.

(d) What is the worst-case running time for Quicksort? Give an example for this. (See **[2006P1Q4 (b)]**, **[2012P1Q5 (c)]**)

(e) What is the best-case running time for Quicksort? (See **[2012P1Q5 (b)]**)

(f) How does Quicksort compare to merge sort and heap sort?

(g) Is it an *in-place* algorithm?

(h) How does pivot picking after the complexity of the algorithm? Why is picking the pivot randomly better than choosing the first element as pivot? (See **[2012P1Q5 (d)]**, **[2016P1Q8 (d)]**)

(i) Why does it matter how you treat elements that are equal to the pivot?

(j) Discuss ways to avoid the worst-case execution of Quicksort. (See **[2008P11Q7 (c)]**)

Exercise 1.C.26 [Quicksort (I)] Implement the Quicksort algorithm. You can test your implementation in problem **[LeetCode 912]** (or **[GeeksForGeeks Quicksort]**).

Exercise 1.C.27 [Quick Select (B)]

(a) Intuitively explain how *quick select* works.

(b) Write pseudocode for quick select.

(c) Argue informally why quick select works.

(d) Assuming that Quicksort chooses the median as a pivot in each step, by solving the linear recurence relation show that quick select takes linear time.

Chapter 2.13

Exercise 1.C.28 [Sorting pairs (I)] You are given an array of `(name, ID)`. In your programming language of choice, show how you can sort this array, first by name and in case of equality by ID.

Exercise 1.C.29 [Stable sorting (B)]

(a) What does it mean for a sorting algorithm to be *stable*? Why would we want this?

(b) Which of the algorithms that you have seen so far are stable?

[Exercise 21 in Lecturer's handout]

(c) How can you convert a non-stable sorting algorithm into a stable? What is the disadvantage of this conversion?

Chapter 2.14

Exercise 1.C.30 [Counting Sort (B)]

(a) Intuitively explain how *counting sort* works.

(b) Write pseudocode for counting sort.

(c) Argue informally why counting sort works.

(d) What is the worst-case running time for counting sort? How does this differ from the complexity of other sorting algorithms?

(e) When would you use counting sort over the other sorting methods?

(f) Is counting sort stable?

Exercise 1.C.31 [Counting Sort (I)] Implement the counting sort algorithm. You can test your implementation in problem **[LeetCode 912]** (or **[GeeksForGeeks Counting Sort]**).

Exercise 1.C.32 [Bucket Sort (B)] Briefly describe how *bucket sort* works.

Exercise 1.C.33 [Radix Sort (B)] Briefly describe how *radix sort* works. (See **[2014P1Q7 (a)]**)