Algorithms Example Sheet 1: Core Questions
Solution Notes

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?

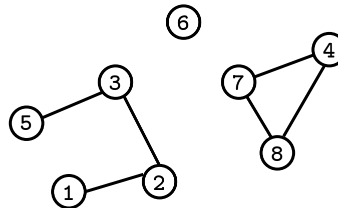
(c) **(I)**Implement your algorithm.

(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).

(a) An array is sorted iff $A[i] \leq A[i+1]$ for every $i \in \{0, 1, \ldots, n-2\}$. We can perform this check using a for-loop over the indices.

(b) The above algorithm requires $n-1$ comparisons in the worst-case and $\mathcal{O}(1)$ time in the best case (if the first two items are in reverse order, then our algorithm will terminate producing `False`).

Proving a lower bound for checking if an array is sorted is a bit more tricky. Assume that we had an algorithm that could decide if an array is sorted using at most $n-2$ comparisons. Let us visualise the pairs that have been compared in a directed graph. For example, if we compared elements with indices 3 and 5, 2 and 1, 3 and 2, 8 and 7, 7 and 4, 4 and 8, the graph would look as follows:

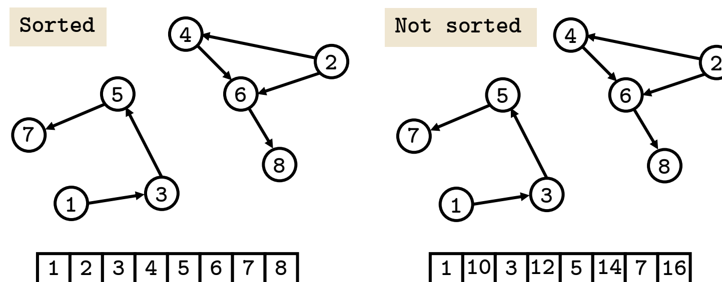


Because the graph has n nodes and $n-2$ edges, it means that there must be at least two components that are not connected (For a formal proof, see later parts of the course on graph theory). We will reveal the entries of the array as the algorithm is comparing elements. For all comparisons asking about indices i and j we respond with $i < j$. Hence, the two disconnected components will be linear chains with some arrows between the elements. This corresponds to:

- a valid assignment that makes A a sorted array, namely $A[i] = i$, and also
- a valid assignment that makes it unsorted (make the elements for the chain that contain index 0, equal to $A[i] = i + n$ and the rest set to $A[i] = i$).

Hence, the algorithm cannot distinguish between these two.

For example, if the comparisons where between 1 and 3, 3 and 5, 5 and 7, 2 and 4, 4 and 6, 6 and 8, 2 and 6, then there exists one sorted array that satisfies the outcomes ($a \to b$ means $b \geq a$) and one unsorted array that satisfies the outcomes.



Hence, the algorithm cannot distinguish between the two and any answer (`True` or `False`) it gives will be wrong (on one of these two outputs).

(c) The following code implements the check:

```
def check_sorted(a):
    for i in range(1, len(a)):
        if a[i - 1] > a[i]:
            return False
    return True


print(check_sorted([]) == True)
print(check_sorted([1]) == True)
print(check_sorted([1, 10, 32, 98]) == True)
print(check_sorted([1, 10, 32, 16]) == False)
```

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]

(a) The part $[0, i)$ is maintained sorted and then item i is inserted in the correct position in $[0, i)$ and the elements after it are moved using swaps.

(b)

$$[31; 41; 59; 26; 41; 58]$$
$$[31; 41; 59; 26; 41; 58]$$
$$[31; 41; 59; 26; 41; 58]$$
$$[26; 31; 41; 59; 41; 58]$$
$$[26; 31; 41; 41; 59; 58]$$
$$[26; 31; 41; 41; 58; 59]$$

(c)
```
public class InsertionSort {

   public static <T extends Comparable<T>> T[] sort(T[] arr) {
      for (int i = 0; i < arr.length; ++i) {
         for (int j = i - 1; j >= 0; --j) {
            if (arr[j].compareTo(arr[j + 1]) > 0) {
               T tmp = arr[j];
               arr[j] = arr[j+1];
               arr[j+1] = tmp;
            } else break;
         }
      }
      return arr;
   }

   public static void main(String[] args) {
      // Integer[] arr = SortUtils.generateRandomSeq(10, 20);
```

```
        Integer[] arr = SortUtils.generateRandomPermutation(20);
        sort(arr);

        System.out.println(SortUtils.isSorted(arr));
        SortUtils.displayArr(arr);

        TimingUtils.computeAvg(
              x -> sort(x),
              SortUtils.generateMultipleRandomSeqs(10, 10000, 1000));
    }

}
```

(d) The array $[0, i)$ is maintained sorted and then the i-th element is inserted correctly, so the array $[0, i]$ becomes sorted. So, inductively the array will be sorted.

(e) The worst-case running time is $\Theta(n^2)$ and it happens for the case when we have a reserve sorted array. The i-th element has to be moved by i positions, so in total $\sum_{i=0}^{n-1} i = n(n-1)/2$ comparisons are required.

(f) The best case running time for insertion sort is $\Theta(n)$ and is achieved when the array is already sorted. Then the i-th element is not moved (i.e. the inner loop terminates in the first iteration).

(g) You can reduce the number of iterations by first finding the position j, where the i-th element should be moved and then moving all elements one position to the left starting with $i - 1$, $i - 2$, until j and then moving $A[i]$ to $A[j]$.

(h) (Solution by Hoa: You can use doubly-linked lists and this way you only need two assignments: to update the pointers of the linked lists.)

(Advanced solution: You can use binary tree that allows moving contiguous subarrays (like the implicit Red-Black tree) and these allow for logarithmic time placements, but also logarithmic time assignments)

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).

The following code solves **[LeetCode 912]**:

```java
class Solution {
    public int[] sortArray(int[] arr) {
        for (int i = 1; i < arr.length; ++i) {
          int j = i - 1;
          while (j >= 0 && arr[j] > arr[j + 1]) {
             int tmp = arr[j];
             arr[j] = arr[j + 1];
             arr[j + 1] = tmp;
             --j;
          }
      }
      return arr;
    }
}
```

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).

(a) A comparison-based sorting algorithm is a sorting algorithm that only performs comparisons between elements. The comparison operator has to be define a pre-order relation between the elements.

(b) You can create a wrapper around the comparison function so that it returns the element is greater than or equal to the other element.

```
boolean new_compare(Object a, Object b) {
   boolean out1 = compare(a, b);
   boolean out2 = compare(b, a);
   // The two objects are the same.
   if (out1 == out2) return out1;
   // Otherwise, reverse the comparison of A <= B.
   return !out1;
}
```

An even simpler approach:

```
boolean new_compare(Object a, Object b) {
   // Instead of returning a < b, it returns b < a.
   return compare(b, a);
}
```

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?

(a) A precondition is an assumption made about the inputs (and state of the program) before a method is called.

(b) A postcondition is what the method guarantees the outputs (and state of the program) will satisfy after the method is called.

(c) An invariant is a property that holds throughout the iterations of a repetitive algorithm (assuming that it was true when the repetitions began). This is similar to the induction hypothesis in natural induction proofs. Invariants help us reason about the execution of for-loops.

(d) Pre-conditions and post-conditions are usually provided at the documentation of the library methods. This is useful for communicating what your method does and for programmers to be able to use it. In some languages (like Java and C++) the preconditions and postconditions described in the standard are a guarantee that any valid implementation of the language will satisfy it (in this and possibly future versions). This is a useful guarantee when writing code that is aimed to stay for years.

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)?

(a) The big-$\mathcal{O}$ notation is used to compare the asymptotic runtime of programs in idealised settings. There are several different machines that could run the same algorithm and one would like to run roughly how efficient the algorithm is with growing input.

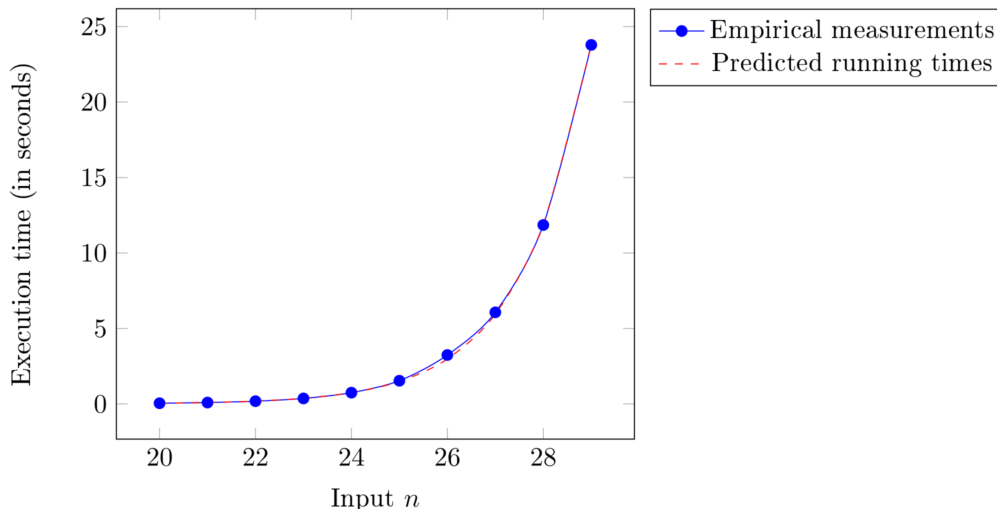
The big-$\mathcal{O}$ estimate can give us an estimate on how to expect resources to grow for larger input sizes.

If we are optimising an algorithm for a particular machine (or architecture) then we should do several empirical measurements to decide on which algorithm to use by examining several inputs that we are expecting.

(b) We use the following code to measure the time complexity:

```
let rec sillySum n =
   if n = 0 then 0
   else n + (sillySum (n - 1) + sillySum (n - 1)) / 2;;

let ignore x = ();

let measure n =
   let t = Sys.time() in
      ignore (sillySum n); Printf.printf "(%d,%f)\n" n (Sys.time() -. t);;

let rec measure_all c r =
   if c < r then (measure c; measure_all (c+1) r);;

measure_all 20 30;;
```

(c) The `sillySum` method has $\Theta(2^n)$ time complexity. We plot the running times versus the input parameter n and the best fit theoretical curve which is $4.43 \cdot 10^{-8} \cdot 2^x$. These are quite close (squared difference is around 0.089).



Question: *How did we pick the constant?* We could have guessed the constant from trial and error, but we could also fit the best curve of the form $a \cdot 2^n$. This is what the following python code does:

```
import numpy as np
import scipy.optimize

x = np.linspace(20, 29, 10)
y = [0.047000, 0.093000, 0.182000, 0.365000, 0.747000, 1.538000, 3.241000,
    6.069000, 11.855000, 23.787000]


def pred(a, v):
```

```python
    return a * np.power(2.0, v)


def log_likelihood(a):
    diff = pred(a, x) - y
    return np.sum(diff * diff)


coeffs = scipy.optimize.fmin(log_likelihood, np.array([0.04]))
preds = pred(coeffs, x)
for (xx, yy) in zip(x, preds):
    print(f"({xx}, {yy})")
print(f"{coeffs[0]} * 2^x")
```

(d) We make several assumptions including: constant-time memory access (not true since the cache reads are faster than reads in main memory), constant-time basic arithmetic operations (addition, multiplication, division) and a few more.

(e) The naive power function takes $\mathcal{O}(n)$ time to compute, while the efficient power function takes $\mathcal{O}(\log n)$ time.

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)?

(a) They focus on the asymptotic growth of resources (time, space, interconnection points etc), which is an estimate independent of the machine being used. This can be helpful for getting an approximate estimate for the resource usage, however in some cases empirical experiments might be required. These notations also allow to formally argue that some algorithm will be say faster than some other algorithm if the input size is sufficient large.

(b) The big-$\mathcal{O}$ notation is defined as

$$f(n) \in \mathcal{O}(g(n)) \text{ iff } \exists k, N > 0. \forall n > N. 0 \leq f(n) \leq k \cdot g(n).$$

The Θ notations is defined as

$$f(n) \in \Theta(g(n)) \text{ iff } \exists k_1, k_2, N > 0. \forall n > N. 0 \leq k_1 \cdot g(n) \leq f(n) \leq k_2 \cdot g(n).$$

The bit-Ω notation is defined as:

$$f(n) \in \Omega(g(n)) \text{ iff } \exists k, N > 0. \forall n > N. 0 \leq k \cdot g(n) \leq f(n).$$

(c) i. This definition would not capture the essence of asymptotics, i.e. that we care about large values of the input and not for very small. For example $n+2$ would not be $\mathcal{O}(2^n)$ since for $n=1$, we have $n+2 = 3 > 2 = 2^n$.

ii. The definition is equivalent to the original one, we just have to adjust $N' = N+1$.

iii. The definition is not equivalent to the original one. Here, constant factors do matter. For example $n \in \mathcal{O}'(3n)$, but $3n \notin \mathcal{O}'(n)$, since there is no value n for which $n > 3n$.

iv. The definition is equivalent to the original one (take $k' = \sqrt{k}$).

v. This change is a bit more subtle. The existence of ≥ 0, guarantees that after some point the function is non-negative. This is useful to get some natural properties such that for f_1 and f_2, $\Theta(f_1(n) + f_2(n)) = \Theta(f_1 + 2f_2)$. Consider $f_1(n) = n - n^2 = \mathcal{O}(n)$ and $f_2(n) = n^2$. Then, $f_1(n) + f_2(n) = n = \Theta(n)$, but $f_1(n) + 2f_2(n) = n^2 + n = \Theta(n^2)$.

(d) It means that if we take all the inputs with parameter n (e.g. all array of length n), the maximum of these running times is going to be $\mathcal{O}(f(n))$. Not every upper bound is useful because some might be very loose. For example, if we take the bubble sort its runtime is $\mathcal{O}(n^2)$, so the upper bound $\mathcal{O}(2^n)$ is valid but too loose. You would probably (under reasonable assumptions) avoid picking a 2^n algorithm, but could pick a $\mathcal{O}(n^2)$ algorithm.

(e) It means that there exists an instance with parameter n, such that the running time is $\Omega(f(n))$.

Some lower bounds are quite easy to get. For example any algorithm that executes at least one line of code has running time $\Omega(1)$, but they are not informative. But for example for bubble sort we would like to know that the running time is $\Omega(n^2)$ on some instances. A tight lower bound (i.e. one that matches the upper bound) is usually what we are aiming for).

(f) The little-o notation is defined as:

$$f(n) = o(g(n)) \text{ iff } \forall k > 0. \exists N > 0. \forall n \geq N. 0 \leq f(n) < k \cdot g(n).$$

The little ω notation is defined as

$$f(n) = \omega(g(n)) \text{ iff } \forall k > 0. \exists N > 0. \forall n \geq N. 0 \leq k \cdot g(n) < f(n).$$

(g)

$$\begin{aligned} f(n) = \Theta(g(n)) &\Leftrightarrow \exists k_1, k_2, N > 0. \forall n > N. 0 \leq k_1 \cdot g(n) \leq f(n) \leq k_2 \cdot g(n) \\ &\Leftrightarrow \exists k_1, k_2, N > 0. \forall n > N. 0 \leq f(n) \leq k_2 \cdot g(n) \wedge 0 \leq k_1 \cdot g(n) \leq f(n) \\ &\Leftrightarrow \exists k_1, N_1 > 0. \forall n > N. 0 \leq k_1 \cdot g(n) \leq f(n) \\ &\wedge \exists k_2, N_2 > 0. \forall n > N. 0 \leq f(n) \leq k_2 \cdot g(n) \quad (N := \max(N_1, N_2)) \\ &\Leftrightarrow f(n) = \Omega(g(n)) \wedge f(n) = \mathcal{O}(g(n)) \end{aligned}$$

(h)

$$f(n) \in o(g(n)) \text{ iff } \exists k, N > 0. \forall n > N. 0 \leq f(n) < k \cdot g(n).$$

Consider $f(n) = 3n^2$ and $g(n) = n^2$. Then we want to find for which n and k, $3n^2 < k \cdot n^2$. By choosing $k = 4$, we have that this holds for all values of $n > 0 = N$, since $3n^2 < 4 \cdot n^2 \Leftrightarrow 0 < n^2$.

(i) Construct an algorithm A that runs a for-loop without a body iterating over $1 \ldots 10^{1000}$ and then runs heapsort on an array. Let B be the bubble sort algorithm. Then $T_A = \mathcal{O}(n \log n)$ and $T_B = \mathcal{O}(n^2)$, but on publicly available classical computers A is expected to be slower than B because inputs cannot be as large as 10^{500} (even if we let the algorithms run for a few years).

Note: There are also other weird algorithms such as a sorting algorithm that runs an exponential 2^n subroutine if the input is sorted, otherwise it runs heapsort. The running time of this algorithm is 2^n, the average running time is $n \log n$ (since $2^n \cdot \frac{1}{n!} < 1$). But in practice it is common to have a sorted array, so probably you would not want this modification. Another idea is to weight the running times by the probability of an instance to occur (like in a Bayesian setting).

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?

An *in-place* sorting algorithm is one that uses $\mathcal{O}(1)$ extra space to sort an array. If out program is limited by memory, then sorting in place would help reduce memory by a factor 2 (usually in this case we might also want to look at external sorting methods).
Insertion sort, selection sort, bubble sort heapsort are all in place algorithms.
Quicksort and merge sort need $\mathcal{O}(\log n)$ memory for the call stack.

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.

(a) $n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1 \geq n \cdot (n-1) \cdot \ldots \cdot n/2 \geq n/2 \cdot n/2 \cdot \ldots n/2 = (n/2)^{n/2}$, where $n/2$ is the integer division of n with 2.

(b) $n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1 \leq n \cdot n \cdot \ldots \cdot n = n^n$.

(c) The first part shows that $\log n! \geq n/2 \log(n/2) = n/2 \log(n) - n/2 \log(2) = \Omega(\log n)$ and the second part shows that $\log n! \leq n \log n = \mathcal{O}(n \log n)$. Hence, $\log n! = \Theta(n \log n)$.

(d) See the FoCS Christmas projects. The idea is to bound $\sum_{i=1}^{n} \log(i)$ using $\int_1^{n+1} \log x \mathrm{dx}$ from above and $\int_1^{n} \log x \mathrm{dx}$ from below.

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).

There are $n!$ permutations for items $1, \ldots, n$. Consider any deterministic comparison-based algorithm A. Then, A makes comparisons between the elements and some swaps based on the outcomes of these. For each permutation, we can trace the path of the permutation through the comparisons. Because each permutation requires a different set of swaps to be sorted, each one must lead to a different leaf in the tree of possible executions (also called decision tree).
Consider starting from the root of the decision tree and following the branch with more outgoing permutations (which must be at least half of the incoming ones by the pigeonhole principle), then there must be at least $\log_2 n!$ comparisons (because each comparison at most halves the number of permutations). Hence, there must be some permutation that requires at least $\log_2 n! = \Omega(n \log n)$ comparisons.

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.

(a) Simply iterate through the array and compare the element against the one you are searching.

```
def find(a, x):
    for v in a:
        if x == v:
            return True
    return False


print(find([1, 10, 23, 7, 5], 7) == True)
print(find([1, 10, 23, 7, 5], 10) == True)
print(find([1, 10, 23, 7, 5], 2) == False)
```

```
print(find([1, 10, 23, 7, 5], 20) == False)
```

(b) The algorithm takes $\Theta(n)$ steps in the worst case, where n is the number of elements in the array, e.g. when the searched element does not occur in the array. In the best case it takes $\mathcal{O}(1)$ (if the element is at the beginning of the list).

We need at least n comparisons. Assume that there is an algorithm A that performs fewer than those comparisons. Then given an array of n elements, A will not check one of the entries of the array. For each entry that the algorithm observes we could set it to 0 and the search element is 1. Hence, A will examine only 0s and will give an answer. If the answer is that there is an element $1<$ then we set the unseen elements to 0. Otherwise, we set them to 1. Hence, A will answer incorrectly in this example.

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]

(a) In the i-th step, selection sort finds the smallest entry in $[i, n)$ (by iterating through that subarray) and swaps it at position i.

$$\begin{gathered}[31; 41; 59; 26; 41; 58] \\ [\mathbf{26}; 41; 59; 31; 41; 58] \\ [26; \mathbf{31}; 59; 41; 41; 58] \\ [26; 31; \mathbf{41}; 59; 41; 58] \\ [26; 31; 41; \mathbf{41}; 59; 58] \\ [26; 31; 41; 41; \mathbf{58}; 59]\end{gathered}$$

(b) See the implementation below.

(c) In the i-th step, selection sort has correctly sorted the subarray $[0, i)$.

(d) In the i-th step selection sort needs $n-i$ steps to find the minimum. Hence the running time is proportional to $1 + 2 + \ldots + (n-1) + n = n(n+1)/2 = \Theta(n^2)$ regardless of the input.

(e) It is the same as in the worst-case. Looking just at the asymptotic complexities, insertion sort has the advantage of doing fewer operations if the array is almost sorted (this is a vague statement), while selection sort will always take quadratic time. On the other hand they have the same average case complexities, but selection sort always performs $\mathcal{O}(n)$ array assignments which on some machines with slow memory access, this could be beneficial. Insertion sort on the other had requires $\Theta(n^2)$ assignments on average.

(f) This happens for the reverse sorted permutation $[7; 6; 5; 4; 3; 2; 1]$ since the items on the right are always smaller than the items on the left.

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]**).

The following implementation passes the task on GeeksForGeeks.

```
class Solution {
   int  select(int arr[], int i) {
        int idx = i, mn = arr[i];
         for (int j = 0; j < i; ++j) {
             if (mn < arr[j]) {
                 mn = arr[j];
                 idx = j;
             }
         }
         return idx;
   }

   void selectionSort(int arr[]) {
      int n = arr.length;
      for(int i=n-1; i>=0; i--) {
         int j = select(arr, i);

         int temp = arr[i];
         arr[i] = arr[j];
         arr[j] = temp;
      }
   }
}
```

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)]**)

(a) The binary search algorithm operates on a sorted array A and given a target value t determines if that value is in the array. It works by maintaining an active subarray $[\ell, r]$ of possible positions where t could lie (initially this is $[0, n-1]$). Then in each step its compares t with the item $A[(\ell + r)/2]$:

- If equal, then we are done.
- If $t > A[(\ell + r)/2]$, then $\ell' := (\ell + r)/2 + 1$.
- Otherwise, $r' := (\ell + r)/2 - 1$.

(b) Each time we do an unsuccessful comparison the active subarray length halves[1]. So in the i-iteration, the length of the array is $n/2^i$. So, if initially the length of the array is $n = 2^k$, then

$$f(n) = f(2^k) = 1 + f(2^{k-1}) = 1 + 1 + f(2^{k-2}) = \ldots = \underbrace{1 + \ldots + 1}_{k} + f(1) = k + 1 = \log_2(n) + 1.$$

Hence, it takes $\Theta(\log_2(n))$ in the worst-case.

[1] Actually would we need to remove -1 and $+1$ from the above definition for this to hold. The asymptotic complexity is not affected by this, but the analysis is simpler.

(c) Assume that there is an algorithm that finds an element in $K < \log_2 n$ comparisons. The target element could be in any of the n positions, so there are n possible outcomes. By considering the flow of all possible computations, one of the two results of the comparison will have at least $n/2$ (by an application of the pigeonhole principle). Since the longest path has length K, we could distinguish at most $2^K < n$ possible outcomes. So for some positions of the target value, the algorithm should not be able to say that it is there.

Hence, we need at least $\log_2 n$ comparisons.

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]**.

(a)
```
class Solution {
    public int search(int[] nums, int target) {
        if (nums.length == 0) return -1;
        int st = 0, en = nums.length - 1;
        while (st < en) {
            int mn = (st + en) / 2;
            if (nums[mn] == target) return mn;
            else if (nums[mn] < target) st = mn + 1;
            else en = mn - 1;
        }
        if (nums[st] == target) return st;
        return -1;
    }
}
```

(b) The tricky part in these problems is to decide how to handle the case when there are two elements remaining. If there are two elements, then we should either examine the element that allows us to reduce the size of the array. This would either be the first one (in which case we need `mn = (st+en)/2`) or the second one (in which case we need `mn = (st+en+1)/2`).

```
class Solution {

    public int find_upper(int[] nums, int target) {
        if (nums.length == 0) return -1;
        int st = 0, en = nums.length - 1;
        while (st < en) {
            // There is a subtle change here.
            // If we are left with just two elements, say [t; t],
            // mn = (st + en)/2 = 0. But looking at the first element,
            // we would not discard it (and would lead to an infinite loop).
            // Looking at the second element allows to discard the first one.
            // and this is what mn = (st + en + 1)/2 = 1 suggests.
            int mn = (st + en + 1) / 2;
            if (nums[mn] > target) en = mn - 1;
            else st = mn;
        }
        if (nums[st] == target) return st;
        return -1;
    }

    public int find_lower(int[] nums, int target) {
        if (nums.length == 0) return -1;
        int st = 0, en = nums.length - 1;
        while (st < en) {
            int mn = (st + en) / 2;
```

```
            if (nums[mn] < target) st = mn + 1;
            else en = mn;
        }
        if (nums[st] == target) return st;
        return -1;
    }

    public int[] searchRange(int[] nums, int target) {
        return new int[]{find_lower(nums, target), find_upper(nums, target)};
    }
}
```

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.

(a) Since the invariant of the insertion sort algorithm is to keep the part $[0, i)$ sorted, we can binary search to find the position of where the next element $A[i]$ should be placed.

(b) This method takes $\mathcal{O}(\log i)$ time to find the correct position, but still has to reallocate $\mathcal{O}(i)$ elements in the worst case. So the $\Theta(n^2)$ running time persists.

(c) The code below implements this idea

```
class InsertionSort {
   static void insert(int arr[],int i) {
      // Binary search to find position to insert.
      int st = 0, en = i - 1;
      while (st < en) {
         int mn = (st + en) / 2;
         if (arr[mn] <= arr[i]) st = mn + 1;
         else en = mn;
      }
      if (arr[st] <= arr[i]) st = i;

      int j = i - 1;
      while (j >= st) {
         int tmp = arr[j];
         arr[j] = arr[j + 1];
         arr[j + 1] = tmp;
         --j;
      }
   }
   static void insertionSort(int arr[]) {
      for (int i = 1; i < arr.length; ++i) {
         insert(arr, i);
      }
   }
}
```

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?

(a) Bubble sort works by repeatedly swapping adjacent pairs that are not in sorted order. It does this by examining consecutive elements from left to right.

(b)

$$
\begin{array}{llllll}
[\underline{31}; & \underline{41}; & 59; & 26; & 41; & 58] \\
[31; & \underline{41}; & \underline{59}; & 26; & 41; & 58] \\
[31; & 41; & \underline{\mathbf{59}}; & \underline{\mathbf{26}}; & 41; & 58] \\
[31; & 41; & 26; & \underline{59}; & \underline{41}; & 58] \\
[31; & 41; & 26; & 59; & \underline{41}; & \underline{58}] \\
[\underline{31}; & \underline{41}; & 26; & 41; & 59; & 58] \\
[31; & \underline{\mathbf{41}}; & \underline{\mathbf{26}}; & 41; & 59; & 58] \\
[31; & 26; & \underline{41}; & \underline{41}; & 59; & 58] \\
[31; & 26; & 41; & \underline{41}; & \underline{59}; & 58] \\
[31; & 26; & 41; & 41; & \underline{\mathbf{59}}; & \underline{\mathbf{58}}] \\
[\underline{\mathbf{26}}; & \underline{\mathbf{31}}; & 41; & 41; & 58; & 59] \\
[26; & \underline{31}; & \underline{41}; & 41; & 58; & 59] \\
[26; & 31; & \underline{41}; & \underline{41}; & 58; & 59] \\
[26; & 31; & 41; & \underline{41}; & \underline{58}; & 59] \\
[26; & 31; & 41; & 41; & \underline{58}; & \underline{59}]
\end{array}
$$

one more vacuous interation

(c) See actual code in the implementation below.

(d) There are several ways to argue about the correctness of Bubble sort.

(Way 1): Bubble sort sorts the array because in each iteration it reduces the number of inverted pairs by 1.
(Way 2): Bubble sort sorts the array because in the i-th iteration it places the i-th largest value to the correct place.

(e) The worst-case time complexity is $\Theta(n^2)$ and it occurs when the input is inversely sorted, i.e. $[n; (n-1); \ldots; 1]$. This is the worst-case because in each iteration the i-th largest value is placed correctly so it will take at most n iterations to sort them all (each of which requires $\Theta(n)$ time).

(f) The best-case time complexity is $\Theta(n)$ and it occurs when the array is already sorted, so the algorithm does not need to make any comparisons.

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]**).

```
class BubbleSort {
  static boolean bubble(int arr[]) {
```

```
        boolean found_pair = false;
        for (int i = 1; i < arr.length; ++i) {
            if (arr[i-1] > arr[i]) {
                int tmp = arr[i];
                arr[i] = arr[i-1];
                arr[i-1] = tmp;
                found_pair = true;
            }
        }
        return found_pair;
    }

    static void bubbleSort(int arr[]) {
        while (bubble(arr));
    }
}
```

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?

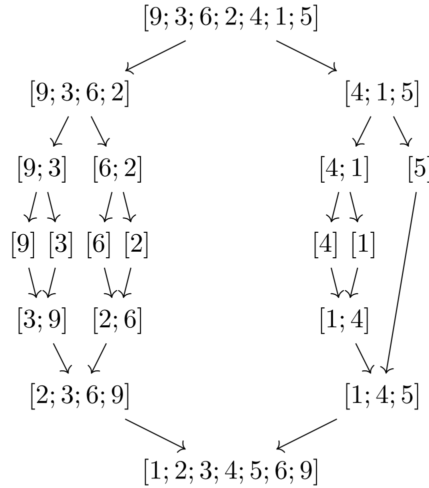
[Exercise 11 in Lecturer's handout]

(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]

(a) Intuitively merge-sort splits the array (or list) in half, recursively sorts each of the two parts and then merges the two sorted parts by iterating through these in sorted order.

(b) One possible way is the following



(c) See implementation below.

(d) The proof is by strong induction.
Base case: Calling `mergesort` for a one-element array will trivially sort it without making any changes to the array.
Inductive step: Assume that `mergesort` produces a sorted output for every array of length $\leq n$. Consider a list of length $n+1$. Then `mergesort` will split the array in half and by the inductive hypothesis the recursive call will return the sorted versions of these two. Now, we need to argue that `merge` works. The invariant that `merge` maintains is that items $[0, i_1 + i_2)$ in the output are sorted and that they are all smaller than $A[i_1]$ and $B[i_2]$ (or ∞ if one does not exist). Hence, appending the minimum of $A[i_1]$ and $B[i_2]$ and incrementing the respective iterator, maintains this invariant.

(e) The best-case time complexity is the same as the worst-case time complexity for mergesort. Let $f(n)$ be the running time for `mergesort` on a list of length n. It takes $f(n/2)$ to sort each of the two halves and it takes $g(n) = \Theta n$ time to merge their outputs. Then, for $n = 2^m$,

$$\begin{aligned}
f(2^m) &= 2 \cdot f(2^{m-1}) + 2^n = 2 \cdot (2 \cdot f(2^{m-2}) + 2^{n-1}) + 2^n \\
&= 2^2 \cdot f(2^{m-2}) + 2^n + 2^n \\
&= 2^3 \cdot f(2^{m-3}) + 2^n + 2^n + 2^n \\
&= 2^n \cdot n = \Theta(m \log m)
\end{aligned}$$

BUT, this is not rigorous. If we wanted to be rigorous, we should have first found a lower and an upper, by explicitly referring to the constants k_1 and k_2 in the definition $g(n) = \Theta(n)$. For example, the lower bound should be changed to,

$$\begin{aligned}
f(2^m) &\geq 2 \cdot f(2^{m-1}) + k_1 \cdot 2^n = 2 \cdot (2 \cdot f(2^{m-2}) + k_1 \cdot 2^{n-1}) + k_1 \cdot 2^n \\
&\geq 2^2 \cdot f(2^{m-2}) + k_1 \cdot 2^n + k_1 \cdot 2^n \\
&\geq 2^3 \cdot f(2^{m-3}) + k_1 \cdot 2^n + k_1 \cdot 2^n + k_1 \cdot 2^n \\
&\geq k_1 \cdot 2^n \cdot n = \Omega(m \log m).
\end{aligned}$$

The upper bound is exactly the same with $\geq \to \leq$ and $k_1 \to k_2$.

Question: *How do we deal with the case that the array length is not a power of 2?*

(f) For general uniformly random arrays that are moderately large you would prefer merge sort, since it will probably be an order of magnitude faster. On the other hand if the arrays are small or almost sorted, there is a chance that bubble sort is faster. However, the exact thresholds that one becomes better than the other will depend on the system being measured.

(g) The problem is that one of the two indices could be out of bounds and the access to the array would be invalid.

(h) This is a software engineering problem. A "reasonable" caller to `mergesort` would probably expect either the array to be sorted in place or that a new array is returned and the old remains unaffected. If both

could happen depending on the input, it is easy for this to lead to a bug when both the returned array and the old array are still used.

This problem can be solved by having a wrapper around this problematic function.

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

Compare the following Java implementation:

```
public class MergeSort {

   static void merge(Integer arr[], int l, int m, int r) {
      Integer tmp[] = new Integer[r - l + 1];
      int i1 = l, i2 = m + 1;
      for (int i = 0; i <= r - l; ++i) {
         if (i2 == r + 1 || (i1 <= m && arr[i1] < arr[i2])) {
            tmp[i] = arr[i1];
            ++i1;
         } else {
            tmp[i] = arr[i2];
            ++i2;
         }
      }
      for (int j = 0; j < tmp.length; ++j) {
         arr[l + j] = tmp[j];
      }
   }

   static Integer[] mergeSort(Integer[] arr, int l, int r) {
      if (r <= l)
         return arr;
      int m = (l + r) / 2;
      mergeSort(arr, l, m);
      mergeSort(arr, m + 1, r);
      merge(arr, l, m, r);
      return arr;
   }
}
```

with the following OCaml implementation:

```
let rec take i = function
| [] -> []
| x::xs ->
   if i > 0 then x :: take (i - 1) xs
   else [];;

let rec drop i = function
| [] -> []
| x::xs ->
   if i > 0 then drop (i-1) xs
   else x::xs;;

let rec merge = function
| [], ys -> ys
| xs, [] -> xs
| x::xs, y::ys ->
   if x <= y then x :: merge (xs, y::ys)
   else y :: merge (x::xs, ys);;

let rec tmergesort = function
| [] -> []
| [x] -> [x]
```

```
| xs ->
  let k = List.length xs / 2 in
  let l = tmergesort (take k xs) in
  let r = tmergesort (drop k xs) in
   merge (l, r);;
```

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.

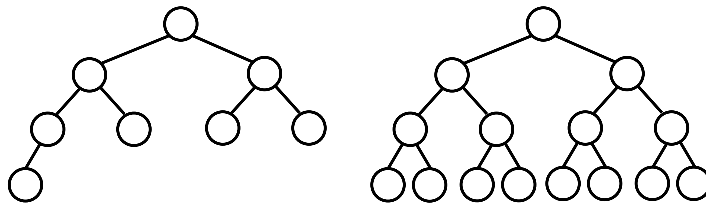
(a) A binary tree is a rooted tree where each node can have at most two children. The root of a binary tree is the node with no incoming edges. (This definition will become clearer when you see graphs)

(b) A full binary tree is a binary tree where the i-th level has 2^i nodes.

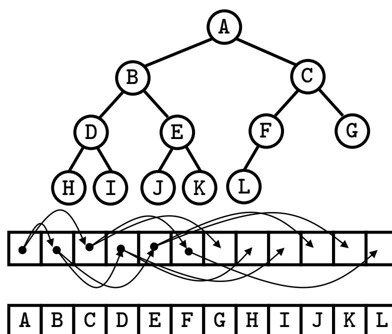
An almost-full binary tree is a tree where in the i-th level there are 2^i nodes, except for the last layer which only has some of these (and all to the leftmost nodes).

(c) For a full binary tree, since every layer has 2^i nodes and there are h layers, there are $2^0 + 2^1 + \ldots + 2^h = 2^{h+1} - 1$.

The minimum number is when the last layer has only one node, so $2^h - 1 + 1 = 2^h$. The maximum number is when the last layer is full, so $2^{h+1} - 1$. Below are the minimum and maximum almost-full binary tree for $h = 4$.



(d) The root node is stored at position 1. If the node is stored at position x, then the children are stored at positions $2x$ and $2x + 1$. The parent of x (if $x \neq 1$) is stored at position $x/2$.



The reason why these encodings lead to unique placements is that considering location x, there is a unique way to write this number in binary $x_1 x_2 \dots x_k$ where 0s correspond to following the left child of a node and 1s correspond to following the right child of a node. By following the same sequence of left/right moves from the root, we should end up in the same node. Hence, these encodings are unique.

Note that for an almost-full binary tree, the array is filled consecutively.

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 $F(x) := \sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$.

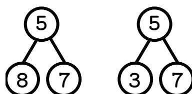
  ii. Let $G(x) := \sum_{i=0}^{\infty} i \cdot x^i$. Show that $\frac{d}{dx} F(x) = \sum_{i=0}^{\infty} i \cdot x^{i-1}$ (what assumption are you making?) and that $G(x) = x \cdot \frac{d}{dx} F(x)$.

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

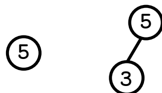
  iv. By analysing the recurrence relation for heapifying an array, show that it takes $\mathcal{O}(n)$ time.

(a) A binary heap is an almost-full binary tree.

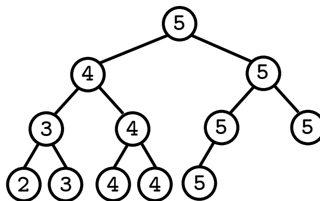
The following binary trees are not heaps: (the rightmost is a BST)



The following BSTs are binary trees:



Question: *When can a BST of distinct keys be a binary heap?* A BST can be a max-heap iff it has no right child. If it has a vertex `v` has a right child, then the BST property says that `v.right`, which violates the max-heap property. There are only two almost full binary trees(up to structure isomorphism), when $n = 1$ and when $n = 2$, that no vertex has a right child. *What if we allow for equal elements?* If we allow for equal elements then we can have a left-recursive chain with arbitrary values, and all right children (and their subtrees) should have the same value as their parent. For example,



(b) The heapify operation works by checking if each node is at least as large as both children. If it is then it terminates, otherwise, it swaps with the smallest of the two and continues recursively there.

(c) The heapify operation takes $\mathcal{O}(h)$. Since $h = \mathcal{O}(\log n)$, doing n heapify operations should take $\mathcal{O}(n \log n)$ time.

(d) i. This is a geometric series. Since $1+x+x^2+\ldots+x^n = \frac{x^n-1}{x-1}$ and for $|x|<1$ we have $\lim_{n\to\infty} x^n = 0$,

$$F(x) = \lim_{n\to\infty}\sum_{i=0}^{n} x^i = \lim_{n\to\infty}\frac{x^n-1}{x-1} = \frac{\lim_{n\to\infty} x^n - 1}{x-1} = \frac{0-1}{x-1} = \frac{1}{1-x}.$$

ii. Assuming that we can change the order of summation and differentiation (when the summation has infinite terms, this is not always possible).

$$\frac{d}{dx}F(x) = \frac{d}{dx}\left(\sum_{i=0}^{n} x^i\right) = \sum_{i=0}^{\infty}\frac{d}{dx}x^i = \sum_{i=0}^{\infty} i\cdot x^{i-1}.$$

Hence, $x\cdot\frac{d}{dx}F(x) = x\cdot\sum_{i=0}^{\infty} i\cdot x^{i-1} = \sum_{i=0}^{\infty} i\cdot x^i = G(x)$.

iii. $G(x) = x\cdot\frac{d}{dx}F(x) = x\cdot\frac{d}{dx}\frac{1}{1-x} = x\cdot\frac{1}{(1-x)^2} = \frac{x}{(1-x)^2}$.

iv. Let's look at the heapifies at level h. There will be 2^h of these and each one will take $\mathcal{O}(\log n - h)$ time (say $\leq k(\log n - h)$). Hence, aggregating their total time complexity is at most

$$\sum_{h=0}^{\log n} 2^h\cdot k\cdot(\log n - h) = \sum_{h=0}^{\log n} 2^{\log n - h}\cdot k\cdot(h) = k2^{\log n}\sum_{h=0}^{\log n} 2^{-h}\cdot(h) \leq kn\sum_{h=0}^{\infty} 2^{-h}\cdot(h) = kn\cdot\frac{2}{(1-2)^2} = 2kn.$$

Hence, it is $\mathcal{O}(n)$ (and actually $\Theta(n)$ since each node is read at least once).

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.

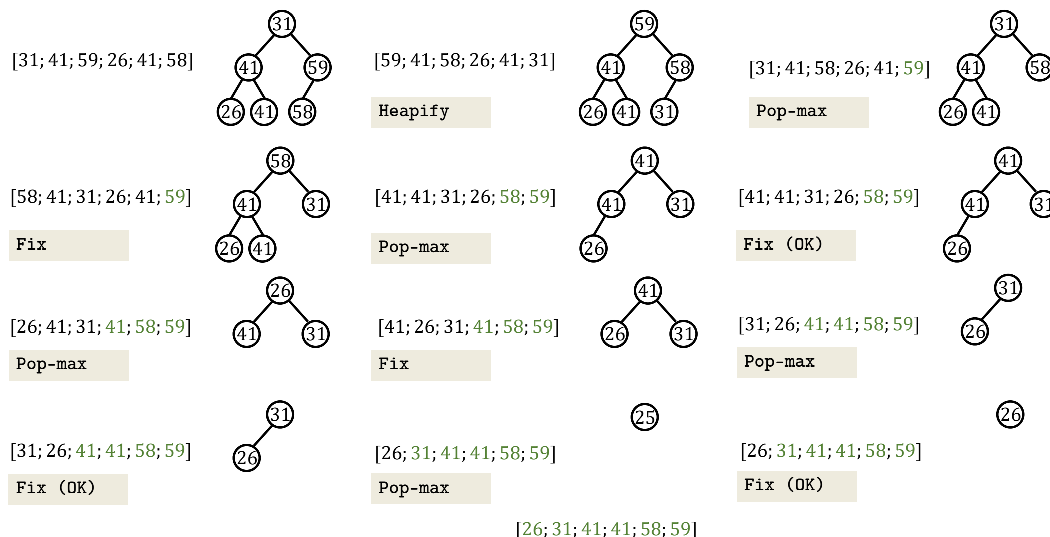
(f) What is the best-case running time for heap sort? How does heap sort compare to merge sort?

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

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

(a) In the i-th step, heap sort maintains elements $[n-i, n)$ in sorted order and the rest of the elements are stored in a max heap. So, the $n-i-1$-st element is the maximum element in the heap. So, it pops this and appends it to the sorted part. By storing the max-heap in the first part of the array, it does the sorting using $\mathcal{O}(1)$ additional memory.

(b) .



(c) See the implementation below.

(d) The explanation above argues about the inductive step. The base case follows because the array is initially sorted. In the end there are no elements in the array and all elements have been placed in order.

(e) It takes $\Theta(n)$ time to construct the heap and each `pop_max` takes $\Theta(\log i)$ time in the worst-case, where i is the number of elements in the queue. Hence, the total running time is $\log n + \log(n-1) + \ldots + \log 1 = \Omega(n \log n)$.

(f) When all elements are equal each `pop_min` takes $\mathcal{O}(1)$ time since no correction is needed. Hence, the time complexity of the algorithm is $\Theta(n)$.

(g) Selection sort maintains $[0, i)$ as being sorted and finds the i-th element by iterating through the rest of the elements and picking the smallest one. This takes $\Theta(n^2)$.

Heap sort maintains $[n-i, n)$ sorted and then picks the maximum element from the remaining elements. Since each `pop_min` takes $\mathcal{O}(\log n)$ time, heapsort takes $\mathcal{O}(n \log n)$ time. A variant of heapsort that maintains $[0, i)$ sorted and adds the minimum of the remaining elements would also work, but it would require $\mathcal{O}(n)$ extra memory (or to store the heap in reflected form).

See official solution notes

(h) .

See official solution notes

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]**).

```java
package sort2;

public class HeapSort {

   static void heapify(Integer[] arr) {
      for (int i = arr.length - 1; i >= 0; --i) {
         shiftDown(arr, arr.length, i + 1);
      }
   }

   static void shiftDown(Integer[] arr, int len, int i) {
      while (true) {
         int left = (2 * i <= len) ? arr[2 * i - 1] : Integer.MIN_VALUE;
         int right = (2 * i + 1 <= len) ? arr[2 * i] : Integer.MIN_VALUE;
         if (arr[i - 1] >= left && arr[i - 1] >= right) return;
         int swapIdx = 2 * i;
         if (arr[i - 1] <= right && right >= left) {
            ++swapIdx;
         }
         swap(arr, i - 1, swapIdx - 1);
         i = swapIdx;
      }
   }

   static Integer[] heapSort(Integer[] arr, int l, int r) {
      heapify(arr);
      for (int i = arr.length - 1; i >= 0; --i) {
         swap(arr, i, 0);
         shiftDown(arr, i, 1);
      }
      return arr;
   }

   static void swap(Integer[] arr, int i, int j) {
      int tmp = arr[i];
      arr[i] = arr[j];
```

```
        arr[j] = tmp;
    }

    public static void main(String[] args) {
        // Integer[] arr = SortUtils.generateRandomSeq(10, 20);
        Integer[] arr = SortUtils.generateRandomPermutation(20);
        heapSort(arr, 0, arr.length - 1);

        System.out.println(SortUtils.isSorted(arr));
        SortUtils.displayArr(arr);

        TimingUtils.computeAvg(x -> heapSort(x, 0, x.length - 1),
                SortUtils.generateMultipleRandomSeqs(10, 10000, 1000));
    }

}
```

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.

```
import java.util.ArrayList;
import java.util.Collections;

public class SortingPairs {

   public static class NameAndId implements Comparable<NameAndId> {
      String name;
      int id;

      public NameAndId(String name, int id) {
         this.name = name;
         this.id = id;
      }

      @Override
      public int compareTo(NameAndId other) {
         if (name == other.name) return Integer.compare(id, other.id);
         return name.compareTo(other.name);
      }

      @Override
      public String toString() {
         return "(" + name + ", " + id + ")";
      }
   }

   public static void main(String[] args) {
      ArrayList<NameAndId> arr = new ArrayList<>();
      arr.add(new NameAndId("Bob", 14));
      arr.add(new NameAndId("Bob", 17));
      arr.add(new NameAndId("Bob", 12));
      arr.add(new NameAndId("Alice", 19));
      arr.add(new NameAndId("Thomas", 14));
      arr.add(new NameAndId("Alice", 11));
      Collections.sort(arr);
      for (NameAndId v : arr) {
         System.out.println(v);
      }
   }

}
```

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?

(a) A sorting algorithm is stable if $A[i] = A[j]$ and $i < j$ then in the output array $A[i]$ appears before $A[j]$.

This is a property that standardises the sorting output which means that we can interchange stable sorting methods without expecting any output changes.

(b) The following algorithms are stable: insertion sort (and binary insertion sort), mergesort, counting sort, bucket sort, radix sort

(c) For each element $A[i]$, we can construct the pair $(A[i], i)$ and define comparison by first and in case of equality by second element. This means that whenever $A[i] = A[j]$, we prefer the element with smaller index. Hence, any correct sorting algorithm will sort them the same way a stable sorting method would. The disadvantage is that this algorithm requires $\Theta(n)$ extra storage for the indices.

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?

(a) Counting sort is used to sort an array A consisting of naturals in the range $[0, U]$. We construct an array C of length $U + 1$, where initially all entries are set to 0 and then iterate through the array elements and for $A[i]$ increment $C[A[i]]$ by one. So after the iteration is over $C[x]$ contains the number of times element x appears in the array. So we just loop through 0 to U and add element x, $C[x]$ times to the output array.

(b) See the implementation below.

(c) The output array will consist of non-decreasing entries since we are looping from 0 to U (and can never step back). Each element in the output array will appear exactly the number of times it appeared in A.

(d) The worst-case (which is also the best) running time is $\Theta(U + n)$. The difference is that this also depends on the maximum element U of the array (not just the length n of the array). If $U = o(n \log n)$, then counting sort is asymptotically faster than any comparison-based sorting method.

(e) This is an open-ended question where the answer depends on the range of the elements compared to the length of the array and the magnitude of the maximum element.

(f) It can be implemented to be stable. To do this we maintain an array of linked list and each element we append to the end. So, when we iterate through the array we add elements $A[i]$ to the end of the corresponding $C[A[i]]$ list. Hence, when creating the output array if $A[i] = A[j]$ and $i < j$, then $A[i]$ will be output first.s

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]**).

```java
public class CountingSort {

  static Integer[] countingSort(Integer[] arr) {
    // Find the maximum in the array.
    // Here we would need to check that none of
    // the entries are negative or null.
    Integer u = max(arr);
    // Create the counting table.
    int[] c = new int[u + 1];
    for (int i = 0; i < arr.length; ++i) {
      ++c[arr[i]];
    }
    // j iterates over the entries [0, U].
    int j = 0;
    for (int i = 0; i < arr.length; ++i) {
```

```
            // Find the next element.
            while (c[j] == 0) ++j;
            --c[j];
            arr[i] = j;
        }
        return arr;
    }

    static Integer max(Integer[] arr) {
        Integer mx = null;
        for (int i = 0; i < arr.length; ++i) {
            if (mx == null || arr[i] > mx) {
                mx = arr[i];
            }
        }
        return mx;
    }

    public static void main(String[] args) {
        // Integer[] arr = SortUtils.generateRandomSeq(10, 20);
        Integer[] arr = SortUtils.generateRandomPermutation(20);
        countingSort(arr);

        System.out.println(SortUtils.isSorted(arr));
        SortUtils.displayArr(arr);

        TimingUtils.computeAvg(x -> countingSort(x),
            SortUtils.generateMultipleRandomSeqs(10, 10000, 1000));
    }

}
```

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

Bucket sort splits the range of the elements into B (these might or might not be uniformly sized). Then iterates through the elements of the arrays and places them in the appropriate bucket. Then each bucket is sorted using say a quadratic sorting algorithm. Then the buckets are merged

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

See official solution notes