1 Warming Up

1.1 Problem and Problem Instance

“Find the smallest number in an array of $n$ integers $a_1, a_2, ..., a_n$.”
What is the input? What is the output?
Is this a problem or a problem instance? If you believe this is a problem, give an instance of this problem.

1.2 Lower Bound vs. Upper Bound

1.2.1 Analyse lower bound and upper bound of an algorithm

Problem: “Does there exist an element 0 in an integer array $a_1, a_2, ..., a_n$?”
Algorithm: Sweep from the left to right until the end of the array. If it sees 0, return yes; otherwise, return no.
Lower bound: What is the best case for this algorithm? How long does the best case take?
Upper bound: What is the worst case for this algorithm? How long does the worst case take?

1.2.2 Big O and its relatives

Show that

$$\log(n!) = \Theta(n \log n).$$

1.3 Effective vs. Efficient

Are selection sort, bubble sort and insertion sort effective algorithms for Minsort problem?
In terms of time, is any algorithm above significantly more efficient to sort a very large unsorted list than the others? Besides time, what else can you improve to make the algorithm more efficient? Give an example.

2 Analysing Algorithms

2.1 Iterative algorithm

Here is an algorithm that, given two sorted lists \( A[1] < A[2] < \ldots A[n] \) and \( B[1] < B[2] < \ldots < B[n] \), decides whether there is a pair of indices \( i, j \) with \( 1 \leq i \leq n \) and \( 1 \leq j \leq n \) so that \( A[i] = B[j] \).

\[
\text{Intersect}(A[1..n], B[1..n]: \text{sorted list of integers})
\]

1. \( I \leftarrow 1, J \leftarrow 1, \text{Found} \leftarrow \text{False} \).
2. While \( I \leq n \) and \( J \leq n \) and \( \text{Found} = \text{False} \) do:
3. IF \( A[I] = B[J] \) THEN \( \text{Found} \leftarrow \text{True} \).
4. IF \( A[I] > B[J] \) THEN \( J++ \) ELSE \( I++ \)
5. Return \( \text{Found} \).

a) Suppose \( A \) is \{1,4,6,8\} and \( B \) is \{2,3,5,6\}, show that how the algorithm works with this problem instance.
   b) Prove that this algorithm is correct, i.e., that it returns true if and only if such a pair \( i, j \) exists.
   c) Give a time analysis, up to order, for this algorithm. Be sure to explain your answer.

2.2 Recursive Merge

\[
\text{RMerge}(A[1..k], B[1..l]: \text{sorted list of integers})
\]

1. IF \( k = 0 \) return \( B[1, \ldots, l] \)
2. IF \( l = 0 \) return \( A[1, \ldots, k] \)
4. ELSE return \( B[1] \circ \text{RMerge}(A[1, \ldots, k], B[2, \ldots, l]) \)

a) Suppose \( A \) is \{1,4,6\} and \( B \) is \{2,3,5\}, show that how the algorithm works with this problem instance.
   b) Prove \( \text{RMerge}(A[1, \ldots, k], B[1, \ldots, l]) \) is a sorted array containing all elements from both sorted array \( A \) and \( B \) by induction on \( n = k + l \).
   c) Give a time analysis for this algorithm.

2.3 MergeSort

\[
\text{MergeSort}(A[1, \ldots, n])
\]

1. IF \( n = 1 \) Return \( A \)
2. \( B[1, \ldots, n/2] \leftarrow \text{MergeSort}(A[1, \ldots, n/2]) \)
3. \( C[1, \ldots, n/2] \leftarrow \text{MergeSort}(A[n/2 + 1, \ldots, n]) \)
4. Return \( \text{Merge}(B[1, \ldots, n/2], C[1, \ldots, n/2]) \)

a) Suppose \( A \) is \{4,3,2,1\}, show that how the algorithm works with this problem instance.
   b) Prove \( \text{MergeSort}(A[1, \ldots, n]) \) return a sorted array with exactly the elements in \( A[1, \ldots, n] \).
   c) Give a time analysis for this algorithm.
3 Two pointers

Problem: You are given a sequence of \( n \) positive integers - \( a_1, a_2, \ldots, a_n \). Find a continuous subsequence \( a_l, a_{l+1}, \ldots, a_r \), such that the sum of all numbers in it is less than \( K \) and its length is as large as possible.

1. Develop an algorithm that solves this problem.
2. Prove its correctness.
3. Show (prove) its time complexity.

4 Counting inversions

Problem: You are given an array of \( n \) elements. Find the number of inversions in this array. The inversion is a pair \((a_i, a_k)\), such that \( i < k \) and \( a_i > a_k \).

1. Develop an algorithm that solves this problem.
2. Prove its correctness.
3. Show (prove) its time complexity.

5 Quicksort

Problem: Quicksort is a very famous and simple sorting algorithm that is used in many applications. The problem is the following – we have an array of \( n \) elements: \( A_0, A_1, \ldots, A_{n-1} \) that we need to rearrange in a way that it becomes sorted in an increasing order, i.e. \( A_0 \leq A_1 \leq \ldots \leq A_{n-1} \). To accomplish this, we will do the following: let’s choose a random element from the array. We will call this element a pivot. Now, let’s break, or partition, the remainder of the array into two groups (maybe empty): the first part contains all elements that are less than or equal to the pivot, the second part contains all elements that are strictly larger than the pivot. We will call this process partitioning. Let’s look at the example:

\[
10 \ 3 \ 5 \ 0 \ 12 \ 17 \ 7 \ 4 \ 5 \ 19
\]

The array contains 10 elements. Let’s choose a random element. Let it be the third element of the array. Its value equals 5.

Now let’s break the array into two parts as was described above:

\[
3 \ 5 \ 0 \ 4 \ 5 \ 10 \ 12 \ 17 \ 7 \ 19
\]

Red-color elements belong to the first group, while blue-color ones belong to the second group. The pivot element’s color is green.

The elements in both groups can be arranged in any way inside their group. As long as the conditions that all elements in the left group are less than or equal to the pivot and that all elements in the group are strictly larger than the pivot are satisfied, any partition will be valid. For example, another valid partition is:

\[
0 \ 3 \ 4 \ 5 \ 5 \ 7 \ 10 \ 12 \ 17 \ 19
\]

As we can see, if all the elements in both groups were sorted in an increasing order, the resulting array after the partitioning would be sorted too. We will use this observation to sort the array. We will call the sorting method quicksort. First, we choose a random element in the array and perform partitioning. Then, we apply the quicksort recursively to each of the resulting groups. The recursion stops when there is only one element in the array.

Apparently, as a result, we will get a sorted array.
1. Develop an algorithm that performs partitioning. Try to make it as efficient as possible.

2. Prove the correctness of the partitioning.

3. Show (prove) the partition algorithm’s time complexity.

4. Prove the correctness of the quicksort.

5. Show (prove) the worst case running time of the quicksort.

6. Show (prove) the average running time of the quicksort. (the average running time of an algorithm is the number of operations averaged over all possible inputs)