1 Warming Up

1.1 Problem and Problem Instance

“Find the smallest number in an array of \( n \) integers \( a_1, a_2, \ldots, a_n \).”

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.

Solution:

Sol: Input: the array \( a_1, a_2, \ldots, a_n \). Output: the smallest number

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 \( a_1, a_2, \ldots, a_n \).”

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?

Sol:

Best case: \( a_1 \) is 0. Constant time. \( O(1) \).

Worst case: no element is 0 in the array. \( O(n) \).

1.2.2 Big O and its relatives

Show that \( \log(n!) = \Theta(n \log n) \).

Hint: Show the upper bound by comparing with \( n^n \), lower bound by comparing with \( ((\frac{n}{2})^2) \).

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.

Sol:

Yes, they are effective because they are correct and terminate.

No, they all take \( O(n^2) \) time.

Efficiency = Output / Input

Improve Efficiency:

1. Output + / Same Input, e.g. sorting more numbers in the same time (if this is the Input measure) or in the same space or with the same energy consumption
2. Output the same / Input -, e.g. sort the same number of integers in less time with less memory requirements, with less energy consumption

Time is not the only measure, e.g. Space and Energy efficient algorithms.
2 Analysing Algorithms

2.1 Iterative algorithm


Intersect($A[1..n], B[1..n]$: sorted list of integers)

1. $I \leftarrow 1, J \leftarrow 1, Found \leftarrow False$
2. While $I \leq n$ and $J \leq n$ and $Found = False$ do:
5. Return $Found$.

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.

**Sol:**
b)

We need to prove two things: First, that if the algorithm returns True, the lists intersect, both containing some value $X$; and second, that if the lists intersect, the algorithm returns True.

The first is simple. The only line where $Found$ gets set to True is line 3, and it only gets set to true if $A[I] = B[J]$. Setting $X = A[I]$, $X$ is in both lists, and so the lists intersect.

The other direction is a bit more difficult. Assume $X$ is in both lists, $X = A[i]$ and $X = B[j]$ for some $1 \leq i, j \leq n$. We'll show as a loop invariant that, if the algorithm has not returned True already, that $I \leq i$ and $J \leq j$. This is true at the start of the algorithm, since $I = 1 \leq i$ and $J = 1 \leq j$. Assume at the start of an iteration $I \leq i$ and $J \leq j$. If $I = i$ and $J = j$, $A[I] = A[i] = X = B[j] = B[J]$. Thus, the If in line 3 is true, and the algorithm halts returning True. If $I = i$ and $J < j$, then $B[J] < B[j] = X = A[I]$ and the algorithm increments $J$, not changing $I$. Since the new value of $J$ is at most $j$, we still have $I \leq i$ and $J \leq j$. The case $I < i$ and $J = j$ is identical. Finally, if $I < i$ and $J < j$, either one we increment, we will still have $I \leq i$ and $J \leq j$. Thus, by induction on the number of iterations, either we have returned True or $I \leq i$ and $J \leq j$.

Now, after $2n$ iterations, the algorithm must terminate, since each iteration that we don’t terminate, we increment $I$ or $J$ and we stop when either reaches $n + 1$. When this happens, by the loop invariant $I \leq i \leq n$ and $J \leq j \leq n$, so we cannot terminate because either $I$ or $J$ exceed $n$. Thus, we must terminate because $Found$ is True, at which point we return True.

Alternative format: Say $X = A[i] = B[j]$. Assume we didn’t return True, to get a contradiction. At the end $I$ or $J$ must exceed $n$, and since we only increment them by at most one each iteration, they must take on all possible values in between. So there must be a point where $I = i$ or a point where $J = j$. Consider the first time when either event happens. Without loss of generality, assume its when $I = i$ (since the $J = j$ case is symmetrical). When this happens, $J \leq j$, since either it happens in the beginning when $J = 1 \leq j$, or $J$ hasn’t yet reached $j$ since the $I = i$ case happened first. Then, while $J < j$, $B[J] < B[j] = X = A[i] = A[I]$, so we will increment $J$ and not change $I$ until $J = j$. At this point, $B[J] = B[j] = X = A[i] = A[I]$, so the algorithm will set $Found$ to True and terminate. This contradicts our assumption that the algorithm did not return True.

c) Time analysis: Each operation within the while loop takes constant time, as does each operation before it. As mentioned above, the while loop can execute at most a total of $2n$ times before the algorithm terminates. Thus, $T(n) \in O(n)$. If the lists do not intersect, at least one counter needs to get incremented $n$ times, so the worst-case time is also $O(n)$.
2.2 Recursive Merge

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

1. IF k = 0 return B[1,...,l]
2. IF l = 0 return A[1,...,k]

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 RMerge(A[1,...,k], B[1,...,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.

Sol:
b) Prove by induction on n, the total input size.
(1) Base case: When n=0. Then both lists are empty. RMerge returns an empty list, which is correct.
(2) Induction: Assume n ≥ 1 and RMerge(A[1,...,k], B[1,...,l]) returns a sorted list containing all elements from either list whenever k + l = n - 1.

We want to prove: RMerge(A[1,...,k], B[1,...,l]) returns a sorted list containing all elements from either list whenever k + l = n.

Case 1: One of the lists is empty. The first or second line of RMerge returns the other list, which is correct.


Similarly, RMerge(A[1,...,k], B[1,...,l]) also works when k + l = n if A[1] ≥ B[1].

Conclusion, RMerge(A[1,...,k], B[1,...,l]) works when k + l = n.
c) Time analysis: According to the recurrence relation, T(n) = T(n - 1) + c. Solving it we have T(n) ∈ O(n).

2.3 MergeSort

MergeSort(A[1,...,n])

1. IF n = 1 Return A
2. B[1,...,n/2] ← MergeSort(A[1,...,n/2])
3. C[1,...,n/2] ← MergeSort(A[n/2 + 1,...,n])
4. Return Merge(B[1,...,n/2], C[1,...,n/2])

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

Sol:
b) (1) Base case: When n=1, the list has only one element, it is sorted already, the MergeSort works;
(2) Induction: Assume that MergeSort sorts n = 1, 2, 3, ..., k elements.

We want to show MergeSort sorts n = k + 1 elements. This is true because MergeSort(A[1,...,(k+1)/2]) returns the sorted first n/2 elements, and MergeSort(A[(k+1)/2+1,...,(k+1)]) returns the sorted rest half
elements. Finally, Merge returns a sorted list with elements exactly in either array. So MergeSort will sort $n = k + 1$ elements.

c) **Time analysis**: According to the recurrence relation, we have

$$T(n) = 2T\left(\frac{n}{2}\right) + cO(n).$$

By solving it we have

$$T(n) \in O(n\log n).$$

**Note:**

Master theorem is used to solve recurrence relation that takes the form

$$T(n) = aT\left(\frac{n}{b}\right) + f(n).$$

See more about Master theorem in wiki or discussion section.
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.

Solution:

1. Algorithm: We will solve this problem with a popular technique called two pointers. Let’s have two variables \( l = 0 \) and \( r = 0 \). Let’s scan our sequence from left to right. Variables \( l \) and \( r \) will denote the beginning and the end of the current subsequence respectively and let \( S(l, r) \) be its sum. At each step, if \( S(l, r) \) is less than \( K \), we have found the subsequence of the length \( r - l + 1 \). Otherwise, while \( S(l, r) \geq K \) and \( l \leq r \), we will increase the value of variable \( l \). At the end of each step, we increase the value of \( r \). During the process, we also have variable \( ans = 0 \) which we will update every time we find a new valid sequence: \( ans = \max(ans, r - l + 1) \). The algorithm stops when \( r = n \).

2. Correctness: Let’s prove that the algorithm finds the largest possible subsequence. Let \( a_{l'}, a_{l'+1}, \ldots, a_{r'} \) be the optimal answer for the problem. At some time of our algorithm, \( r \) will be more than or equal to \( l' \). We will have \( l \leq l' \) and \( r \leq r' \) and \( l' \leq r \). At any time while \( l' \leq r \leq r' \), \( l \) will not exceed \( l' \). Indeed, if at some point \( l \) were bigger than \( l' \), it would mean that there was \( r \leq r' \), such that \( S(l', r) \geq K \). However, this is not possible as all numbers are positive. Therefore, \( S(l', r) \leq S(l', r') < K \). Consequently, there will be a moment during our algorithm, when \( l \leq l' \) and \( r = r' \). As sequence \( a_{l'}, a_{l'+1}, \ldots, a_{r'} \) is optimal, it means that \( l = l' \). So our algorithm finds the optimal answer for this problem.

3. Time complexity: Time complexity of the algorithm is \( \Theta(n) \). Indeed, at each step, we increase either \( l \) (in a loop) or \( r \). Both \( l \) and \( r \) can not exceed \( n \). Each step works in \( \Theta(1) \) time. There are no more than \( 2n \) steps. Therefore, the complexity is \( \Theta(n) \).
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.

Solution:

- 1. **Algorithm:** There is a simple brute force solution, where we just look at all possible pairs.
   - 2. **Correctness:** As we enumerate all possible pairs, we will be able to find all inversions.
   - 3. **Time complexity:** As the number of pairs equals \( \frac{n(n-1)}{2} \), the time complexity will be \( \Theta(n^2) \).

- 1. **Algorithm:** Another approach is to use divide-and-conquer. We will break our array into two parts, such that their sizes do not differ by more than one: \( B = A[0, ..., \lceil \frac{n}{2} \rceil - 1] \) and \( C = A[\lceil \frac{n}{2} \rceil, ..., n - 1] \). We will recursively find the number of inversions in arrays \( B \) and \( C \) and then find all inversions that have one element in \( B \) and the other one in \( C \) (the recursion stops if there is only one element in the array; the result will be zero for this case). For this purpose, let’s assume that elements in \( B \) and \( C \) are sorted. We will scan both arrays from left to right. Initially, \( i = k = 0 \). Then, if we compare elements \( B_i \) and \( C_k \):
   - (a) if \( B_i \leq C_k \), then \( B_i \) doesn’t form inversions with any elements \( C_j \), such that \( k \leq j \). Increase \( i \) by one.
   - (b) if \( B_i > C_k \), then \( C_k \) form inversions with all elements \( B_j \), such that \( i \leq j \). Increase \( k \) by one.
   - 2. **Correctness:** Let’s prove the correctness by the induction:
     - The base case is when there is only one element in the array. The result is zero, which is true.
     - Hypothesis: Let’s assume that the algorithm finds correctly the number of inversions for all arrays of the size less than \( n \).
     - Induction step: Every time we have \( B_i > C_k \), \( B_i \) will be the first element in \( B \) that satisfies this condition (the smallest element in \( B \) that is bigger than \( C_k \)). So, if we take any pair \((B_j, C_k)\), such that \( B_j > C_k \), then \( i \leq j \) and \( B_i \leq B_j \). So for every \( C_k \), we find all possible numbers in \( B \), with which \( C_k \) forms inversions. According to the hypothesis, we will correctly find all inversions in arrays \( B \) and \( C \), as their sizes are less than \( n \) \((\max(\lceil \frac{n}{2} \rceil, n - \lceil \frac{n}{2} \rceil) < n \text{ for } n > 1)\). Then, if we add all these numbers, we will get the total number of inversions in the array.
   - 3. **Time complexity:** As we can see, at each step we need both arrays \( B \) and \( C \) to be sorted. For this purpose we can use a merge sort algorithm. As you can see, our method is just a slight modification of the merge sort, which also allows us to find the number of all the inversions. By adding a few operations that work in constant time to the merging process of the two arrays in the merge sort, we can solve our problem. So the time complexity of this approach equals the time complexity of the merge sort, i.e. \( \Theta(n \log n) \).
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: $A_0, A_1, ..., A_{n-1}$ that we need to rearrange in a way that it becomes sorted in an increasing order, i.e. $A_0 \leq A_1 \leq ... \leq A_{n-1}$. 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 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 that 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)

Solution:

1. Algorithm: Let’s describe how partitioning works. Let’s look at the same example with the same pivot element:

10 3 5 0 12 17 7 4 5 19
First let’s swap the first element of the array and the pivot element. We will get:

```
  5  3  10  0  12  17  7  4  5  19
```

Now, let’s have two pointers \( l \) and \( r \), which will initially point to the beginning and to the end of the array respectively (\( l = 0, r = n - 1 \)). And let’s denote the pivot element as \( p \).

```
  5  3  10  0  12  17  7  4  5  19
     l      r
```

First, we try to increase \( l \) as much as possible. We increment pointer \( l \), if \( A[l] \leq p \).

```
  5  3  10  0  12  17  7  4  5  19
     l      r
```

Then, we try to decrease \( r \) as much as we can. We decrement pointer \( r \), if \( p < A[r] \).

```
  5  3  10  0  12  17  7  4  5  19
     l      r
```

Then, if \( l < r \), we swap elements \( A[l] \) and \( A[r] \).

```
  5  3  5  0  12  17  7  4  10  19
     l      r
```

If \( l < r \), we also repeat all these three operations again (increasing \( l \), decreasing \( r \) and swapping \( l \) and \( r \) again, if \( l < r \)). As soon as \( l > r \), we swap elements \( A[r] \) and \( p \) and stop partitioning.

```
  5  3  5  0  12  17  7  4  10  19
     l      r
```

```
  5  3  5  0  4  17  7  12  10  19
     l      r
```

```
  5  3  5  0  4  17  7  12  10  19
     l      r
```

```
  5  3  5  0  4  17  7  12  10  19
     r      l
```

```
  5  3  5  0  4  17  7  12  10  19
     r      l
```

```
  4  3  5  0  5  17  7  12  10  19
     r      l
```

8
2. **Correctness (partitioning):** Let’s prove that partitioning is correct. Indeed, after we increase \( l \) and decrease \( r \), all elements to the left of \( A[l] \) are less than or equal to \( p \), and all elements to the right of \( A[r] \) are strictly larger than \( p \). After this process, we get that \( A[l] > p \) and \( A[r] \leq p \). If \( l < r \), we swap \( A[l] \) and \( A[r] \) and get that \( A[l] \leq p \) and \( A[r] > p \). If \( l > r \), we swap \( A[r] \) and \( p \), which are both less than or equal to \( p \) and stop partitioning. As a result, we get that all elements to the left of \( p \) are less than or equal to \( p \), and all the elements to the right of \( p \) are strictly larger than \( p \).

3. **Time complexity (partitioning):** The time complexity for partitioning will be \( \Theta(n) \). Indeed, at each step we increase \( l \) or decrease \( r \). Each step takes \( O(1) \) time. Both \( l \) and \( r \) can’t be increased/decreased more than \( n \) times, so there are not more than \( 2n \) steps. On the other hand, the number of iterations will be at least equal to \( n \), as we continue partitioning till \( l > r \).

4. **Correctness (quicksort):** Now, let’s prove that the quicksort is correct.

   The base case is when we have only one element. One element is an ordered array, so the base case is correct.

   The hypothesis is that the quicksort sorts correctly all arrays of the size less than \( n \).

   Now, there is the inductive step. For an array of size \( n \), we, first, choose a pivot element \( p \). Then, we perform partitioning bases on this element. As a result, the array will be divided into two groups: the left group in which all elements are less than or equal to \( p \), and the right group in which all elements are strictly larger than \( p \). The sizes of these groups are less than \( n \) (because they at least do not contain the pivot element). Then, we use the quicksort on them recursively. According to the hypothesis, both these groups of elements will be sorted correctly. As a result, the initial array will consist of the left part which is sorted correctly, the pivot element and the right part which is also sorted correctly. Therefore, the whole array will be sorted correctly.

5. **Time complexity (quicksort):** Let \( T(n) \) be the number of operations the quicksort makes for the array of size \( n \). Then,

   \[
   T(0) = \Theta(1) \\
   T(n) = T(i) + T(n - i - 1) + \Theta(n)
   \]

   where \( A[i] \) is a pivot element.

   Let’s say, that at each iteration, the pivot element is the smallest element in the array. Then, we have:

   \[
   T(n) = T(0) + T(n - 1) + \Theta(n) \\
   T(n) \geq T(n - 1) + n - 1 \\
   T(n) \geq T(n - 2) + n - 2 + n - 1 \\
   \vdots \\
   T(n) \geq 1 + 2 + \ldots + n - 2 + n - 1 \\
   T(n) \geq \frac{(n-1)n}{2}
   \]

   Therefore, \( T(n) \in \Omega(n^2) \)

   So in the worst case, the quicksort will take quadratic time to sort the array. But this case happens rarely. Let’s see how fast the quicksort works on average (the expected time complexity).

6. **Time complexity (quicksort):** Let partitioning take \( cn \) operations and \( T(n) \) be the average running time. Then, we will have the following recurrent equation:

   \[
   T(n) = \frac{1}{n} \sum_{i=0}^{n-1} (T(i) + T(n - i - 1) + cn)
   \]
\[ T(n) = \frac{2}{n} \sum_{i=0}^{n-1} T(i) + cn \]
\[ nT(n) = 2 \sum_{i=0}^{n-1} T(i) + cn^2 \]
\[ (n - 1)T(n - 1) = 2 \sum_{i=0}^{n-2} T(i) + c(n - 1)^2 \]
\[ nT(n) - (n - 1)T(n - 1) \approx 2T(n - 1) + 2cn \]

\[ \frac{T(n)}{n+1} \approx \frac{T(n-1)}{n} + \frac{2c}{n+1} \]

\[ \sum_{i=1}^{n} \frac{T(i)}{i+1} \approx \sum_{i=1}^{n} \left( \frac{T(i-1)}{i} + \frac{2c}{i+1} \right) \]

\[ \frac{T(n)}{n+1} \approx \frac{T_0}{1} + 2c\sum_{i=1}^{n} \frac{1}{i} - 1 \]

\[ \frac{T(n)}{n+1} \approx 2c(\ln n - 1) \]

\[ T(n) \approx c'n \ln n \]

\[ T(n) \in \theta(n \log n) \]