# Design and Analysis of Computer Algorithms

11059005 蕭耕宏

## Assignment 5

Due: June 16, 2025 (before class)

Please use A4-sized papers to write your answers for the homework sets.

1. Suppose that you are given an array of strings. Different strings may have different number of characters. The total number of characters over all the strings is n. Show how to sort the strings in O(n) time. (We assume that the order is the lexicographic order.)

   **Solution:**

   **Basic Idea:**
   Radix sort can be used to sort strings in O(n) time, starting from the rightmost position fo the logest string. For each position we use counting sort to srot the strings based on the character at that position (or a special "null" charactere for strings shorter than the current position). Since counting sort is linear, and we process at most L position where L is the length of the longest string, the total time is proportional to the total number of characters n.

   **Detailed Approach:**
   1. Find the maximum string length L, the length of the longest string in the array.
   2. Pad shorter strings conceptually with a special "null" character to make them all of length L.
   3. For each position k from L-1 down to 0:
      - use counting sort to sort the strings based on the character at position k.
      - the character set is assumed to be finite.
   4. counting sort detail:
      - create a count array fro the character set.
      - for each string, increment the count of its character at position k
      - compute cumulative counts to determine positions

      place strings in a new array based on their character at position k preserving the relative order from the previous iteration (stable sort).

   **Correctness:**
   Sorting right to left ensures that more significant in lexicographic order positions are processed later, preserving the correct order. Also, counting sort is stable, meaning that it maintains the relative order of strings with the same character at the current position. Lastly, since we process each character position independently and in linear time, the overall complexity remains O(n).

   **Example:**
   Input: ["cat", "dog", "car", "do"] (n = 3 + 3 + 3 + 2 = 11 characters)

   Max length: L = 3 (length of "cat", "dog", "car").

   Position k = 2:

   Characters: "cat"[2] = 't', "dog"[2] = 'g', "car"[2] = 'r', "do"[2] = null. Counting sort order: null < 'g' < 'r' < 't'. Result: ["do", "dog", "car", "cat"].

   Position k = 1:

Characters: “do”[1] = ‘o’, “dog”[1] = ‘o’, “car”[1] = ‘a’, “cat”[1] = ‘a’. Counting sort (stable): [“car”, “cat”, “do”, “dog”].

Position k = 0:

Characters: “car”[0] = ‘c’, “cat”[0] = ‘c’, “do”[0] = ‘d’, “dog”[0] = ‘d’. Counting sort (stable): [“car”, “cat”, “do”, “dog”]. Output: [“car”, “cat”, “do”, “dog”].

2. Describe an O(n)-time algorithm that determines the k numbers in S which are closest to the median of S where the given set S has n distinct number and $k \leq n$ is a given integer.

**Solution:**

**Basic Idea:**
Firstly, we cana’t fully sort the array (which takes O(n log n) time), but we can use Quick Select algorithm to find the median efficiently. After finding the median, we compute the distance of each element from it and select the k elements with the smallest distances, again using a selection method to maintain linear time.

**Detailed Approach:**

1. Use QuickSelect algorithm to find the median of S. For n elements, the median is the $\lfloor \frac{n}{2} \rfloor$-th smallest element using 0-based indexing. QuickSelect finds this in O(n) time on average by partitioning the array around a pivot.
2. Compute the absolute difference between each element in S and the median. Store these distances in a list pairs of (distance, element).
3. Use QuickSelect on the list of pairs to find the (k-1)-th smallest distance (0-based index) to determine the k closest elements. Collect all elements whose distance is less than or equal to this k-th smallest distance. If ties result in more than k elements, we can drop the extra elements arbitrarily since the problem does not specify how to handle ties.

**Correctness:**
QuickSelect guarantees that we can find the median in linear time, and since we are only computing distances and selecting k elements based on these distances, the overall complexity remains O(n). The selection of k closest elements is also efficient since we are not fully sorting the distances but only finding the k smallest ones.

**Example:**
Consider S = [1, 3, 5, 7, 9, 11] , n = 6 , and k = 3 .

Find the Median:

Median position is floor(6/2) - 1 = 2 (0-based). QuickSelect finds the 2nd smallest element: 5 (sorted: [1, 3, 5, 7, 9, 11]).

Calculate Distances:

|1-5| = 4 , |3-5| = 2 , |5-5| = 0 , |7-5| = 2 , |9-5| = 4 , |11-5| = 6 . Distances: [4, 2, 0, 2, 4, 6].

Find the k-th Smallest Distance:

k = 3 , so find the 2nd smallest distance (index 2 in 0-based sorted order). Sorted distances: [0, 2, 2, 4, 4, 6]. The 2nd smallest is 2.

Select Elements:

Elements with distance <= 2 : 3 (dist 2), 5 (dist 0), 7 (dist 2). Result: [3, 5, 7], exactly k = 3 , so no further adjustment needed.

Output: The 3 numbers closest to the median are [3, 5, 7] .

3. Let A and B are two sorted arrays and each is of size n. Give an O(log n)-time algorithm to find the median of all 2n elements in A and B.

**Solution:**

**Basic Idea:**
Since merging the arrays A and B would take O(n) time, we can use a binary search approach to find the median directly without merging. The median of the combined arrays is the (n-1)-th element in 0-based indexing of the merged array. We can use binary search on the smaller array to find the correct partitioning that gives us the median.

**Detailed Approach:**

1. Use binary search on the index i(where 0 ≤ i ≤ n) of array A to decide how many elements from A go to the left part. Compute the corresponding index j in array B as j = n - i.

Initially set low = 0 and high = n. Compute i = (low + high) / 2 and j = n - i.

2. Let $A_{left} = A[i-1]$ (largest element in A's left part, or - $\infty$ if i = 0)

   Let $A_{right} = A[i]$ (smallest element in A's right part, or + $\infty$ if i = n)

   Let $B_{left} = B[j-1]$ (largest element in B's left part, or - $\infty$ if j = 0)

   Let $B_{right} = B[j]$ (smallest element in B's right part, or + $\infty$ if j = n)

3. when $A_{left} \leq B_{right}$ and $B_{left} \leq A_{right}$, compute the median.
   $\frac{\max(A_{left}, B_{left}) + \min(A_{right}, B_{right})}{2}$

   Otherwise, if $A_{left} > B_{right}$, then we need to move left in A, so set high = i - 1. if $B_{left} > A_{right}$, then we need to move right in A, so set low = i + 1.

**Correctness:**
The condition $A_{left} \leq B_{right}$ and $B_{left} \leq A_{right}$ ensures that we have correctly partitioned the arrays such that all elements in the left part are less than or equal to all elements in the right part. The median is then computed as the average of the maximum of the left parts and the minimum of the right parts, which is correct for both even and odd total lengths. Using $-\infty$ and $+\infty$ allows us to handle edge cases where the partition index is at the boundaries of the arrays.

**Example:**
Consider $A = [1, 3, 5]$ and $B = [2, 4, 6]$, with $n = 3$. The merged array would be $[1, 2, 3, 4, 5, 6]$, and the median is `frac`$\{3 + 4\}\{2\} = 3.5$.

- **Initial Attempt:**
  - $low = 0$, $high = 3$, $i = 1$, $j = 2$.
  - $A_{left} = A[0] = 1$, $A_{right} = A[1] = 3$.
  - $B_{left} = B[1] = 4$, $B_{right} = B[2] = 6$.
  - Check: $1 < 6$ (true), $4 < 3$ (false). Since $4 > 3$, increase $i$.
- **Next Iteration:**
  - $low = 2$, $high = 3$, $i = 2$, $j = 1$.
  - $A_{left} = A[1] = 3$, $A_{right} = A[2] = 5$.

- $B_{left} = B[0] = 2$, $B_{right} = B[1] = 4$.
- Check: $3 < 4$ (true), $2 < 5$ (true). Conditions satisfied.
- Median $= \frac{\max(3,2)+\min(5,4)}{2} = \frac{3+4}{2} = 3.5$.

4. Consider the algorithm Select 2 we discussed in class where the input elements are divided into group of 5. Will this algorithm work in linear time if they are divided into groups of 7? How about groups of 3?

   **Solution:**

   **Basic Idea:**
   The Select 2 algorithm is median of medians selection algorithm that guarantees linear time complexity by recursively selecting the median of groups of a fixed size. The choice of group size affects the number of recursive calls and the partitioning process.

   MM-Select

   1. Divide the elemtns into $\frac{n}{r}$ groups
   2. Find median of each group, $m_i$
   3. Find the median of among all the $m_{i'}s$ by calling MM-Select itself.
   4. Partition the elemtns by the partition element mm and assume the mm is at position j.
   5. If k is j then return the element, else, call MM-Select itself depending on the relation between k and j.

   $T(n) = \begin{cases} T(\frac{n}{5})+T(3\frac{n}{4})+cn & \text{if } n \geq 24 \\ cn & \text{if } n \leq 24 \end{cases}$

   The problem is to determine if changing the group size to 7 or 3 still allows the algorithm to run in linear time.

   Therefore, we need to analyze how the group size affects the number of elements that are guaranteed to be greater than or less than the median in each recursive step.

   **Detailed Approach:**

   **Groups of 7:**
   To build time complexity, we can start with step 1, 2 and 4.

   Step 1 is divide the elements into groups of 7, which gives us $\frac{n}{7}$ groups, taking O(n) time.

   Step 2 is to find the median of each group, which takes O(1) time (since r is constant), so O(n) in total.

   Step 4 is to partition the elements based on the median of medians, which takes O(n) time using a linear partitioning algorithm like QuickSelect.

   For step 3, time to find mm among $\lfloor \frac{n}{7} \rfloor$ is O(n/7). Step 5 requires extra attention.

   number of group $m = \lfloor \frac{n}{7} \rfloor$

   number of elements $\leq mm \geq 4 \times \lceil \frac{m}{2} \rceil \approx \frac{2n}{7}$

   number of elements $\geq mm \geq 4 \times \lfloor \frac{m}{2} \rfloor \approx \frac{2n}{7}$

   $T(n) = \begin{cases} T(\frac{n}{7})+T(\frac{2n}{7})+cn & \text{if } n \geq 49 \\ cn & \text{if } n \leq 49 \end{cases}$

   Assume $T(n) \leq 7cn$

   Base case: $T(n) = cn \leq 7cn$

Inductive step:

$T(n) = T\left(\frac{n}{7}\right) + T\left(\frac{2n}{7}\right) + cn$

$\leq 7c\left(\frac{n}{7}\right) + 7c\left(\frac{2n}{7}\right) + cn$

$= cn + 2cn + cn = 4cn$

$\leq 7cn$ (holds for n >= 49)

Therefore, the algorithm works in linear time for groups of 7.

**Groups of 3:**
For groups of 3, we can follow a similar analysis.

Step 1 is divide the elements into groups of 3, which gives us $\frac{n}{3}$ groups, taking O(n) time.

Step 2 is to find the median of each group, which takes O(1) time (since r is constant), so O(n) in total.

Step 4 is to partition the elements based on the median of medians, which takes O(n) time using a linear partitioning algorithm like QuickSelect.

For step 3, time to find mm among $\left\lfloor \frac{n}{3} \right\rfloor$ is O(n/3).

Step 5 requires extra attention.

number of group $m = \left\lfloor \frac{n}{3} \right\rfloor$

number of elements $\leq mm \geq 2 \times \left\lceil \frac{m}{2} \right\rceil \approx \frac{n}{3}$

number of elements $\geq mm \geq 2 \times \left\lfloor \frac{m}{2} \right\rfloor \approx \frac{n}{3}$

$T(n) = \begin{cases} T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + cn & \text{if } n \geq 9 \\ cn & \text{if } n \leq 9 \end{cases}$ By checking coefficients 1/3 + 2/3 = 1, we can see that the algorithm runs in O(nlog n) time.

Therefore, the algorithm **does not** work in linear time for groups of 3.