# Multi select – Algorithm and Data Structure

A yellow rectangular box with purple numbers and a purple arrow pointing to a array

Description automatically generated

Indice

[Multi select – Algorithm and Data Structure 1](#_Toc149572684)

[Analisi del problema 3](#_Toc149572685)

[Pseudo Codice algortimo 4](#_Toc149572686)

[Analisi di complessità (Worst case) 4](#_Toc149572687)

[Risultato ottenuto 4](#_Toc149572688)

## Analisi del problema

Vogliamo implementare un algoritmo di sorting, in particolare un MultiSelect. Il funzionamento generale di questo algoritmo prevede che dato un’array input di ***M*** elementi e un indice ***K***,esso restituisca un array lungo ***M*** con in posizione ***K*** un numero. Alla sinistra di questo numero saranno presenti in ordine ***sparso*** numeri minori o uguali a ***M[K]***. E alla sua destra numeri in ordine ***sparso*** maggiori o ugualli a ***M[K]***.

La versione che dobbiamo implementare presenta però un ulteriore complicazione. Infatti tra i dati di input è presente un fattore ***N*** il quale determina il numero di ***K***. Più precisamente sarà importante determinare la migliore sequenza di ***K*** tra le possibili per minimizzare il numero di mosse necessaire al completamento dell’algoritmo.

Questo siccome il numero di mosse ultimate viene determinato da:

* La prima iterazione avrà sempre un numero ***M*** di mosse.
* In base al ***K*** scelto genereremo due sottoinsiemi di dimensioni variabili. Questo determina il numero di mosse al passaggio successivo.
* La sequenza di ***K*** determina poi il proseguimento dell’algortimo e le dimensioni dei sotto insiemi.
* Il numero totale di mosse corrisponderà a ***M*** +

Ecco un esempio visivo di quanto spiegato fino ad ora:

A group of people standing on a podium

Description automatically generated

Figura 1. Possibile risultato di una multi-select con N=3 e K=[2,4,10].

Esempio di input output del programma:

A white rectangular object with black numbers

Description automatically generated

## Pseudo Codice algortimo

## Analisi di complessità (Worst case)

## Risultato ottenuto

## Difficoltà riscontrate

To solve William's problem efficiently, we need to find the best sequence of using the **select** algorithm on the array and its sub-arrays so that we minimize the total number of algorithm steps. This problem can be approached as an optimization problem, where we try different sequences and calculate the total steps for each sequence, then choose the one with the minimum steps.

The key to solving this problem is to identify which index **K** to choose first, and then recursively apply the **select** algorithm to the resulting sub-arrays. We can use a recursive function to try out different sequences and use memoization to store already computed results for sub-problems to avoid redundant computations.

Here's a high-level approach to implement the solution in C++:

1. **Sort the Indices**: First, sort the indices **K**. This will make it easier to handle the sub-arrays.
2. **Recursive Function**: Implement a recursive function that takes the current sub-array bounds and the remaining indices to process. This function should try running **select** on each remaining index and recursively call itself on the resulting sub-arrays.
3. **Memoization**: Use a map or a 2D vector to store the results of already solved sub-problems to avoid redundant calculations.
4. **Minimize Steps**: In the recursive function, accumulate the number of steps taken and choose the sequence that results in the minimum number of steps.
5. **Base Case**: When there are no more indices left to process, return 0.
6. **Output**: Call the recursive function from the **main()** and output the result.

This code uses a map to memoize the results of sub-problems, and the recursive function **minSteps** tries all combinations of selecting the **K** index for the select algorithm. Note that the input indices in **K** are assumed to be zero-based.