DESIGN AND ANALYSIS OF ALGORITHM

Project

Submitted by
Corentin GIGOT (ES04084)
Morgat COCHENNEC (ES04094)
Zoë SAVI (ES04034)
Léa FOUGERA-LEMPEREUR (ES04032)
Group name: First call

DEPARTMENT OF COMPUTER SCIENCE
FACULTY OF COMPUTER SCIENCE AND INFORMATION TECHNOLOGY

SEM 2 2022-2023

Original scenario :

Our scenario is such that we want to find the number of a person in UPM, whether it is a professor or a student from his name/firstname. We can also search for a name from a phone number or simply search for the number and name of all teachers.
We are therefore creating a phone directory specific to the UPM to facilitate contact between students and campus staff.

Why finding an optimal solution for this scenario is important ?

In order to increase the directory's effectiveness, productivity, accessibility, dependability, and scalability, an ideal solution must be found. Indeed, effectiveness guarantees that looking up names or phone numbers is done swiftly and effectively.
This program makes it simple for instructors and students to communicate and provide a quick access.
On the other hand, accessibility is crucial, and the user interface needs to be straightforward and accessible to everyone. Furthermore, dependability is necessary to avoid inaccurate search results, which would compromise the optimisation of the solution. Finally, the programme must be expandable, allowing for the addition of features or the implementation of extra data to provide a longer lifespan for the programme and to accommodate shifting campus demands.

Suitability of sorting, DAC, DP, greedy and graph algorithms by stating their strengths and weaknesses :

Algorithms		Strengths	Weaknesses
Divide and Conquer		Divide the problem into independent subproblems to decrease the time complexity (O(log(n))) Useful for sorted structures	Need a sorted structure of data
Dynamic programming		Divide the problem into linked subproblems to decrease the time complexity (O(n^2))	Need often more memory time
Greedy method		Easier than other algorithms to make it work	Optimal solution each calculation step is different than optimal solution of entire list. Won't be useful to find a precise answer.
Graph method		Take into account relation between entities	Difficulty of implementation and scalabity. (Complexity of input size increase in an exponential manner)

Method used for our scenario :

We must use divide and conquer to solve our problem for these reasons :

Graph method is useful when you can establish relationships between entities (here there are none), moreover input list in term of phone book can be really wide which doesn't help with Graph method's time complexity. That's why Graph method is not efficient in this case. Greedy method computes an optimal solution for a partial problem. This can lead to imprecise outcomes. In other words, this may not match to find a precise answer like a phone number.
Dynamic programming is more interesting but effective when subproblems overlap between them. Or in our case, subproblems are not linked. That's why Divide and conquer should be an optimal answer for our problem. By the way, we will use a specific type of divide and conquer named Binary search, which focuses on a smaller part of the array each loop, in fact it does not compute all subproblems, hence we will reduce our time complexity.
In our code, we use Quicksort to order the database to illustrate different useful features (search by name, first name, phone number,...), but normally, the database is already sorted and so the time complexity should be only O(log(n)) instead of O(nlog(n)).

Algorithm paradigm by emphasising which part needs recurrence and the function for the optimization :

The "divide and conquer" paradigm serves as the foundation for the algorithm paradigm for binary search.
The fundamental principle of binary search is to compare the sought item with the middle item after recursively splitting the range of values into two equal portions. The item's location in the left or right side of the range can be determined based on this comparison.
In a binary search, the search range must be split into two halves continually until the searched object is discovered or the range is decreased to zero. This recursive split reduces the amount of things to verify and increases the search's efficiency by allowing only half of the range to be reviewed at each stage.
The recursive (or iterative) function, which compares the sought item to the middle item in the current range, is the primary function in binary search. After deciding which area of the range should be investigated next, this function repeats the procedure with the appropriate half. Up until the sought object is discovered or the range is finished, recursion is used to divide the range into two portions at each step.
Each search step in binary search optimises the search range by halving it, making it feasible to locate the searched object rapidly.

Algorithm specification :

In order to develop our scenario described above, we need to use data stored in a file. These data are professor or student first names, last names, phone numbers as well as the role of each person on campus. In terms of constraints, each person from the UPM should sign a paper authorizing them to reveal their data in this program, so we have taken fictitious data in order to carry out this program. In addition, phone numbers must be registered in the same form: +66 12345678 and the data in our file in a specific order: surname, first name, function, telephone number. Finally, we seek to know the results in as little time as possible.

Analysis of the algorithm’s correctness as well as time complexity by using asymptotic notation :

Time complexity of our program is the time complexity of quick sort + binary search.

Quick sort Time complexity is : O (n log(n))
Best case : O (n log(n)). When the partitioning algorithm always chooses the middle element or near the middle element as the pivot, the best case scenario happens.
Average case : O (n log(n)). This occurs when the array elements are in a disordered sequence that isn’t increasing or decreasing properly.
Worst case : O (n^2). The worst-case situation is when the partitioning algorithm picks the largest or smallest element as the pivot element every time.

Binary Search time complexity is : O log(n).
The best-case time complexity would be O(1) when the central index would directly match the desired value.
The worst-case scenario could be the values at either extremity of the list or values not in the list. So the time complexity for the worst case is O log(n).
The average case would be O log(n) when element that you search is not at either extremity of the list but in it.
So as we can see, Binary Search is negligible compared to quicksort time complexity, so our program follows quicksort time complexity.
But, in our code, we use Quicksort to order the database to illustrate different useful features (search by name, first name, phone number,...), but normally, the database is already sorted and so the time complexity should be only binary search time complexity which is O(log(n)) instead of O(nlog(n)).