Structure de données et algorithmes
Projet 1: Algorithmes de sélection

25 février 2025

The goal of the project is to implement, theoretically analyze and empirically compare different selection algorithms, i.e. those allowing to find the `k`-th smallest value in a set.

Algorithme de sélection

The selection problem consists in identifying in an array of size `N` the `k`-th smallest value, given a comparison function between values. We will assume in this project that the initial array is not necessarily sorted and that it is allowed to permute its elements. The function to be implemented `Select(A,k)` will take as argument the array `A` and the value of `k` and will return the position in `A` of the `k`-th smallest value of `A`.

A naive way to solve the selection problem is to sort the entire array and simply return `k`. This solution is however generally inefficient because it does too much work compared to what is strictly required. We propose to study 4 algorithmic solutions to this problem[1]:

1. SelectionSelect
2. HeapSelect
3. QuickSelect
4. FRSelect

The first three algorithms are adaptations of the sorting algorithms of the same name to make the selection. The fourth is a variant supposed to be faster of the `QuickSelect` algorithm. These four algorithms are described below.

SelectionSelect. The `SelectionSelect` algorithm is an adaptation of selection sort to solve this problem. The idea of selection sort is to determine the minimum element of the array, swap this element with the first element of the array, and then proceed in the same way to sort the rest of the array. For the selection problem, we can simply apply this idea until we have determined the position of the `k`-th smallest element in order to avoid unnecessary calculations.

HeapSelect. ¡The `HeapSelect` algorithm is based on the idea of heapsort to make the selection. Heapsort proceeds in two steps: the first consists of creating a heap from the array, while the second arranges the elements at the end of the array from largest to smallest by iteratively extracting the element at the top of the heap. If we modify the algorithm to sort the array in descending order, using a min-heap rather than a max-heap, we can therefore interrupt the second step when the `k`-th smallest element is the one at the top of the heap.

[1]It should be noted that an optimal theoretical solution to the problem is given by an algorithm called "Median of Medians", which is also a variant of the `QuickSelect` algorithm. This algorithm, although $\Theta(n)$ in the worst case, is however relatively slow in practice.

QuickSelect. The `Quickselect` algorithm is an adaptation of `Quicksort` for the selection problem. The idea of this algorithm is, like for quicksort, to distribute the elements of the array according to a pivot value (arbitrarily chosen in the array). Once the partition is done, rather than making the two recursive calls of `Quicksort`, it is possible to do without one or both calls in the case of selection. Indeed, if the position of the pivot `q` is equal to `k`, the algorithm can directly return `q` as the index searched. If `q` < `k` (resp. `q` > `k`), the `k`-th value is to the right (resp. to the left) of the pivot and a single recursive call can be made on the corresponding subarray to find the `k`-th value.

FRSelect. The Floyd-Rivest algorithm[2] is an improved version of the `QuickSelect` algorithm. This variant potentially reduces the size of the subarray more quickly than the `QuickSelect` algorithm. The idea to achieve this is to more intelligently determine the pivot element to place it as close as possible to the element `k` sought. To do this, the algorithm delimits a subset of size S (with $S \ll N$) of the initial array and it recursively searches for the k'-th element in this subset with k' adjusted with respect to the size of S to represent the same quantile as `k` with respect to N. The element thus selected will then be the pivot value used for the partitioning, which can be carried out as in the case of the `QuickSelect` algorithm. The size S of the subarray considered is adapted to the size of the array by a rather complicated formula, which allows to obtain good theoretical and practical performances. To understand the details of this algorithm, we invite you to consult the sources given in the footnote.

Implémentation

You must fill 4 files each implementing one of the four solutions:

- `SelectionSelect.c`;
- `HeapSelect.c`;
- `QuickSelect.c`
- `FRSelect.c`

These four files must provide the `select` function as defined in the `Select.h` file (provided). You can add as many additional functions as needed (which must be declared statically).

The prototype of the `select` function is as follows:

```
size_t select(void *array, size_t length, size_t k,
int (*compare)(const void *array, size_t i, size_t j),
void (*swap)(void *array, size_t i, size_t j))
```

where `array` is a pointer to the start of the array to be sorted (of type `void *`, so that it can be applied to any type of array), `length` is its length, `k` is the position in the order of the element to find, `compare` is a function returning an integer respectively smaller, equal or greater than 0 if the element at position `i` in `array` is smaller, equal or greater than the element at position `j` and `swap` is a function that allows you to swap the elements of the array at positions `i` and `j`. For example, you can use this function on an array of integers as follows:

```
void swapInt(void *array, int i, int j) {
   int temp = ((int*)array)[i];
   ((int*)array)[i] = ((int*)array)[j];
```

[2] See this document https://people.csail.mit.edu/rivest/pubs/FR75b.pdf or the wikipedia page https://en.wikipedia.org/wiki/Floyd-Rivest_algorithm.

```
   ((int*)array)[j] = temp;
}

int compareInt(void *array, int i, int j) {
   return (((int*)array)[i] - ((int*)array)[j]);
}

int array[5] = {4, 2, 1, 3, 0};
size_t pos = select(array, 5, 3, compareInt, swapInt);
```

As stated above, the `select` function can swap the elements of the array but only by using the `swap` function given as an argument. The elements must be compared only with the `compare` function. The `select` function must return the position of the `k`-th smallest element in the array after calling the function. We will assume that the argument `k` is between 0 and `length` $- 1$ inclusive. Called with `k` $= 0$ (respectively `k` $=$ `length` $- 1$), the function must thus return the minimum (respectively maximum) element of the array.

Your implementations must be *as efficient as possible* while following the instructions given. The implementation of the algorithms `SelectionSelect` and `HeapSelect` does not call for any particular comments. For the implementation of `QuickSelect`, you have several degrees of freedom regarding the choice of the pivot and the partitioning function. For the latter, think of a way to avoid as much as possible the occurrence of the worst cases. For the implementation of `FRSelect`, strictly follow the pseudo-codes proposed in the two sources given above regarding the choice of the pivot. You are free, however, in the choice of the implementation of the partitioning function.

Your files will be compiled with `gcc` and the following compilation flags:

```
--std=c99 --pedantic -Wall -Wextra -Wmissing-prototypes
```

This implies that the project must be made in the C99 standard. The presence of *warnings* will negatively impact the final rating. A file that does not compile will receive a zero rating.

A `Makefile` file is provided that creates 4 executables from your implementations using a `main.c` file also provided. These executables first check the correctness of the `select` function on an array of size 1000 and then run a test on the four types of arrays specified below for arrays of size 10000 and the calculation of the median (`k` $= 5000$). You can test other array sizes and values of `k` by giving them as arguments on the command line. You are of course free to modify this file `main.c` to answer the questions below.

Theoretical analysis

In the report, you are asked to answer the following questions:

1. Give in a table the time and space complexities in the worst case and the best case of your 4 implementations as a function of the size `N` of the array and the value of `k`. Briefly justify each of the cases, specifying if necessary the specificities of your implementations. For the algorithm `FRSelect`, you can consult external sources to determine these complexities (by citing them).

2. By adapting the complexity analysis of the average case of `Quicksort` seen in the course and under the same assumptions, show that the average complexity of `Quickselect` is $\Theta($`N`$)$ in the case where `k` is set to 0 (search for the minimum).

3. A selection algorithm is stable if it respects the order of identical elements, that is, if it takes into account the initial order of these identical elements to determine the `k`-th smallest. For each of the 4 algorithms, specify whether they are stable or not, briefly justifying your answer.

Experiments

1. Measure the computation times and the number of comparisons of the 4 algorithms in the case of random arrays, arrays ordered in ascending and descending order, and arrays containing only identical values for the search of the median ($\mathtt{k} = \lfloor \mathtt{N}/2 \rfloor$) and the 10th percentile ($\mathtt{k} = \lfloor \mathtt{N}/10 \rfloor$). Report these computation times in two tables (one for each value of `k`) in the format below:

Type of array	random			increasing			decreasing			constant		
Size	10^4	10^5	10^6	10^4	10^5	10^6	10^4	10^5	10^6	10^4	10^5	10^6
SelectionSelect HeapSelect QuickSelect FRSelect												

2. Comment on these results. For each type of table:
   - compare the evolution of computation times as a function of the size of the table to the theoretical complexities,
   - comment on the relative order of the different algorithms,
   - discuss the impact of the parameter `k`.

Notes:

- The functions to generate the 4 types of arrays are provided in the file `IntArray.c` and the file `main.c` performs an experiment for each type of array.
- The times reported must be average times established on the basis of at least 10 experiments. This is particularly important in the case of a random array.
- Specify in your comments the specificities of your implementations that have an influence on the theoretical results (for example, the choice of the pivot and the partitioning function used by the `QuickSelect`).

Date of submission and submission

The project must be completed ***alone or in pairs*** by ***Sunday, March 23, 2025 at 11:59 p.m.*** at the latest. The project must be submitted via Gradescope (http://gradescope.com, see course code on Ecampus)

The following files must be submitted:

1. your report (6 pages maximum) in PDF format and named `report.pdf`. Be brief but precise and respect the numbering of the (sub-)questions;
2. the four files `SelectionSelect.c`, `HeapSelect.c`, `QuickSelect.c` and `FRSelect.c`.

Respect the file extensions as well as the names of the files `*.c` (including case). Do not include any additional files.

A project not submitted on time will receive an overall score of zero. Submitting the project assumes that you have read the rules regarding plagiarism recalled on Ecampus. In case of plagiarism[3] proven, the group will be assigned a zero grade for the entire course.

The correction criteria are specified on Ecampus.

Good work!

[3]Anti-plagiarism tests will be carried out.