Juan Bustamante
Sebastián Jaramillo

Engineering method

Problematic situation:
A prominent bank wishes to speed up the process in one of its busiest offices. To accomplish this, they want a software tool capable enough to manage clients flow and attention; clients database manipulation and its proper visual representation.

We are using the engineering method to achieve a solution that covers those requirements.

Step 1. Problem identification

1.1 Bank needs:

- The bank needs to **attend** its clients **in the order they get to the office**.
- The bank needs to **attend handicapped clients according to their priority**.
- The bank needs to **organize** its clients inside the building with **queues**: **one for standar-abled** clients and **another for those with special needs**.
- The bank needs to **collect** information about its clients.
- Clients need to be able to **deposit** and **withdraw** money from their bank accounts.
- Clients need to be able to **pay** their credit card debt.
- Clients need to be able to **cancel** their bank accounts.
- Clients need to be able to **undo** the last action of its bank account.
- All of this needs to be **done through two employees, one for each queue.**

1.2 The problem:
The bank needs a new software to help its employees attend the clients, their needs, and collect information.

Step 2. Information gathering

To reach certain requirements, we should consider how other businesses from the same field solved similar problems.

2.1 Credit cards amount per bank account:

> BBVA USA offers only one credit card per bank account, although they have different kinds of credit cards with different benefits each. We'll ignore this as our client does not offer this option.

2.2 Accounts per client:

> Entities such as BBVA, CaixaBank and Openbank give its clients freedom on their number of accounts.

Step 3. Ideas for a solution:

Some of the operations the program will do are sorts. These sorts will be replicated to the next variables:

- Client's `name`: A string value representing the name of the client. Might not be unique.
- Client's `id`: A numerical value representing the name of the client. Unique.
- Client's `money`: A numerical value representing the sum of all their saved money, from all their bank accounts. Might not be unique.
- Client's `vinculatedTime`: A date value representing how old is the client's account. Might not be unique.

We propose a combination of four out of these six sorting algorithms to manipulate the data.

3.1 Algorithms propositions

3.1.1 Bubblesort:

- Time complexity of $\Theta(n^2)$ in the average; not the best in matters of time.
- Minuscule memory usage, its worst space complexity is $O(1)$

3.1.2 Mergesort:

- Uniform performance in all cases. Its best, average and worst time complexity are all the same: $\Theta(n \log(n))$. Quite fast.
- Large memory usage, in the worst case is as big as the amount of data: $O(n)$.

3.1.3 Quicksort:

- As fast as merge in the best and average case, but really slow in its worst. Its time complexity is $\Theta(n \log(n))$ and $O(n^2)$ respectively.
- Efficient memory usage, the worst case of its space complexity is $O(\log(n))$

3.1.4 Heapsort:

- In matters of time this algorithm acts like Merge sort, being its best, average and worst time complexity $\Theta(n \log(n))$.
- What makes it different from the last two is its small, constant memory use: $O(1)$

3.1.5 Selection Sort:

- A slow $\Theta(n^2)$ for all of its time cases (best, average and worst), but it's compensated by its little memory usage.
- At its worst case this sort memory usage is a lite $O(1)$.

3.1.6 Radixsort:

- Quick algorithm specialized in sorting numbers, its all-case time complexity is $\Theta(nk)$, where k is the maximum number of digits a number can have.
- Uses more memory than the data it is trying to sort, its all-case space complexity is $O(n+k)$, where k is the same as announced above.

3.2.1 Solutions

3.2.1 Sol. 1 - Time based decisions:

The Bank's critical need is to speed up the process of attending its clients and manipulating their data. It is reasonable then, to use the fastest algorithms in our list, those will be: 3.1.2 Mergesort, 3.1.3 Quicksort, 3.1.4 Heapsort and 3.1.6 Radixsort. And, as IDs are going to be treated as numerical values, this solution will also allow us to sort them with Radixsort.

3.2.2 Sol. 2 - Memory based decisions:

A bank in this level is managing a great flow of clients every day, and therefore, the program will do as well. With this solution we aim at memory efficiency using algorithms with low-level memory usage, even though they may be slower than others. As we don't know the memory capacity of the banks machines, this alternative will more likely work on more of them, no matter how much available space they have. The proposed algorithms are 3.1.1 Bubblesort, 3.1.4 Heapsort, 3.1.5 Selection Sort and 3.1.3 Quicksort.

3.2.3 Sol. 3 - Balanced decisions:

The previous solutions will not work if (1) the machine(s) does not have enough memory available, and thus, can not execute some algorithms; and (2) if the machine(s) does not have enough processing power to sort and deliver the data in reasonable times.

This solution is conformed to minimize those risks taking Sol. 1, discarding the most memory-demanding algorithm, and replacing it with one from Sol. 2. We end up then having to choose between Bubblesort and Selection Sort (from Sol. 2), and discharging Selection Sort because of its all-cases time complexity of $\Theta(n^2)$ (best for bubblesort is $\Theta(n)$).

After this, our third solution is confirmed then like: 3.1.2 Mergesort, 3.1.3 Quicksort, 3.1.4 Heapsort and 3.1.1 Bubblesort.

Step 4. Transition from ideas to preliminar design:

4.1 Discharging alternatives:

4.1.1 Solution 2:

After giving it some thought, we realized that Sol. 2 may not be the most suitable one for this case, and Discharging it is reasonable. Here are some reasons why:

- It is very unlikely for today's machine to encounter memory problems, even manipulating large data sets.
- The application will be used to attend clients in real time, prioritizing speed over memory optimization in our algorithms should come first.

Step 5. Solution evaluation:

We now find ourselves with two solutions to choose from. We are defining some criteria to select the best possible solution.

5.1 Criteria:

To evaluate we will *only* consider average cases.

- A: Number of algorithms which **time requirements** are below (n^2):
  - [1] One point
  - [2] Two points
  - [3] Three points
  - [4] Four points
- B: Number of algorithms which **its time complexity** are the same for worst and average case:
  - [1] One point
  - [2] Two points
  - [3] Three points
  - [4] Four points
- C: Number of algorithms which **its time complexity** for the best case is better than its complexity for the average case:
  - [1] One point
  - [2] Two points
  - [3] Three points
  - [4] Four points
- D: Number of algorithms which **space requirements** are constant (n):
  - [1] One point
  - [2] Two points
  - [3] Three points
  - [4] Four points

Criteria \ Solution	A	B	C	D	Total
Solution 1	4	3	0	2	9
Solution 3	3	3	1	3	10

5.2 Selection:

Both solutions are clearly at a similar level. However, based on our criteria, 3.2.3 Sol. 3 - Balanced decisions is best for this case, and is the one to be implemented.