Lesson 8. Algorithms

November 29, 2022

1 Summary

- Introduction to computational complexity
- Searching algorithms
- Sorting algorithms

2 Introduction to computational complexity

- Computational complexity is the way we measure how good an algorithm is to solve a given problem or how difficult is to solve a given problem using computers
- We usually distinguish between *time* and *memory* (space) complexity, but usually time complexity is more important (I can always buy more memory for my computer but I cannot buy time)
- Usually there is a trade-off between time and space
- We measure how the time/memory grows with the size of the problem
- Depending on that we speak about classes of complexity: most known classes are P and NP
    - P is the set of all problems that can be solved in polynomial time. That means that the time it takes the problem to be solved depends on some polynomial of the size of the problem, usually its number of elements `n`. For example $t = n^2 + n$, $t = n^4 + 3 * n^3$. In the first case, if the size of the problem is multiplied by 10, the time will grow by $10^2 + 10 = 110$ (the time to solve a problem 10 times bigger is multiplied by 110). In the second case it will be multiplied by 13,000. In any case, despite the grade of the polynomial P problems are considered solvable
    - NP is the set of problems for which no polynomial solution is known, but once a solution is found, checking that it is really a solution can be done in polynomial time
    - The most important open question in computer science is P == NP ? (big prize for the one solving it)
- To specify the computational complexity the big-O notation is used. For example we say that an algorithm is $O(n^2)$ (to be read "order of n^2") and that means that the time grows with the square of the size of the problem. The previous examples were $O(n^2)$ and $O(n^4)$ (we take only the highest grade polynomial, ignoring the smallest ones)

3 Searching algorithms

- Algorithms for searching for an element in a list

- Linear search and binary search
- Linear search: I walk through all the list until I find the element I am looking for. The (worst case) complexity is $O(n)$ (linear). It is the only algorithm that can be used in non-sorted lists.
- Binary search: It can only be used in sorted lists. I check if the element I am looking for is at the middle of the list. If not, as the list is sorted I will know whether it lies at the first of second half of it, so I can check the middle of that part of the list, doing it again until either I find the element or I notice it is not in the list. Complexity: $O(log\ n)$ (2-based log)

4 Sorting algorithms

- Two families: internal and external sorting
- Internal sorting: we sort using the same list plus maybe an auxiliary variable
- External sorting: we sort using another list
- Internal sorting: simple methods (bubble, selection sort and insertion sort) and advanced methods (quick sort, heap sort, shell short…)
- Simple methods are $O(n^2)$ and advanced methods $O(n * log\ n)$

4.1 Bubble sort

- Simplest, but not so efficient, sorting algorithm
- I use a loop to compare first element to second one, if they are not in order I swap them. I do the same with second and third, until I have compared all. If I am sorting in ascending order the biggest will be at the end. Then I start another loop, comparing again first to second, etc.
- Example, sorting[6,3,5,4,1,2]

First iteration (first comparison round)

3,6,5,4,1,2

3,5,6,4,1,2

3,5,4,6,1,2

3,5,4,1,6,2

3,5,4,1,2,6

Second iteration

3,5,4,1,2,6

3,4,5,1,2,6

3,4,1,5,2,6

3,4,1,2,5,6

Third

3,4,1,2,5,6

3,1,4,2,5,6

3,1,2,4,5,6

Fourth

1,3,2,4,5,6

1,2,3,4,5,6

Fifth

1,2,3,4,5,6

The number of comparisons in each iteration is $n + n - 1 + n - 2... + n$ which is equivalent to n^2, so it is $O(n^2)$

4.2 Selection sort

- We directly select (doing linear search) the smallest element of the list (or the biggest if we sort in descending order) and swap it with the first element of the list. Next we search for the second smallest one, swapping it with the second element of the list, and so on

Example: sorting [6,3,5,4,1,2]

1,3,5,4,6,2

1,2,5,4,6,3

1,2,3,4,6,5

1,2,3,4,6,5

1,2,3,4,5,6

- Its complexity is also $O(n^2)$ as we need to perform $n-1$ linear searches with $O(n)$ complexity each

4.3 Insertion sort

- We take advantage of the fact that a 1-element list is always sorted. So we create a sublist just with the 1st element. Then we insert the second element in its position, before or after the first. We do the same with the third element in the list of the 1st and 2nd, and so on.
- Example: sorting [6,3,5,4,1,2]
- On the left the sorted elements sublist

```
6   | 3,5,4,1,2


3,6   | 5,4,1,2


3,5,6   | 4,1,2


3,5,4,6 | 1,2
```

```
3,4,5,6 | 1,2


3,4,5,1,6 | 2

3,4,1,5,6 | 2

3,1,4,5,6 | 2

1,3,4,5,6 | 2


1,3,4,5,2,6

1,3,4,2,5,6

1,3,2,4,5,6

1,2,3,4,5,6
```

- We need to place n elements comparing to 1, 2, 3… elements each time so its complexity is also $O(n^2)$
- Even if the three algorithms have the same complexity, bubble sort is slower than the other two ones. With the provided implementations selection sort is slightly faster than insertion sort