Algorithm Engineering

Federico Matteoni

A.A. 2021/22

Index

0.1 Introduction

Teacher: Paolo Ferragina
Exam: written + oral. Midterms in November and December, with exercises.
Classes will be recorded on Microsoft Teams. Also the book "The Magic of Algorithms" is very important: you must be used to talk about these things, not just be able to solve exercises.

Course **Design** and **analysis** of algorithms but also insights about **implementation**, with reference to libraries and considerations about what happens when using certain algorithms. The case of use is **big data**.

Algorithm Knuth's definition: "*a finite, definite, effective procedure that takes some input and returns an output, with the output being the answer to the problem you want to solve.*" So, a finite sequence of steps, not only a finite numbers of operations but also the algorithms must **terminate**. `while (true) do ...` will go on forever, so it's not an algorithm. Definite means that the steps are definite in an unambiguous way. Effective means that every step is basic, atomic, something that we can execute in small or constant time, constant is not a very precise word (seconds? Milliseconds?) so we will accept the "small time" rough definition. Also, the mapping input $\rightarrow$ output must always be correct, which is the biggest difference with IA. An algorithm outputs the correct output for each input.

RAM Random Access Machine, CPU $\leftrightarrow$ M, classical computing model (Von Neumann machine), the memory can read any place in constant time.
We will make a more sophisticated step. But without presenting very complicated models. We need a good balance, not perfect but a better approximation than the RAM.

Algorithm A and find function $T_A(n)$ that describes the time complexity of A. n = input size, number of items that the algorithm has to process. Hours, seconds, milliseconds based on the machine, but we approximate the time taken with the number of steps. Also, the number of steps depends on the number of items but also on the items itselves. So we usually analyze the worst case scenario, or less often the average scenario. Analyzing the worst case scenario we can figure out the worst or "maximum" number of steps. **Asymptotic analysis**.
We want to exploit the characteristics of the various types of memory.

We will count not steps but the I/O ops, with a 2 level memory model: the first model is the fast (cache + RAM) and the second level is the mass memory. In small memory situations, the first level can be interpreted as cache and the second level as internal memory, other times the first level is the internal memory and the second level is unbound slow memory.

Spatial Locality: access near items

Temporal Locality, or small working set: far apart items used often so we can exploit their presence in the cache.

Poly vs Exp time complexity Let's say we have three algorithms: n, n^2, 2^n in time complexity respectively. Let's express the time complexity fixing t time and counting how many items we can process in t time. In the first case is linear, so $n = t$, the second is $n = \sqrt{t}$ and the third is $n = \log_2 t$. If we have a k times faster machine, we can imagine using the original machine for k more times. $n = kt, n = \sqrt{kt}, n = \log_s kt$. The linear algorithm has full advantage of the k times faster machine, the second algorithm has a small advantage of a multiplication factor $\sqrt{k}$ and the last a negligible advantage of a sum factor of $\log_2 k$, which is basically none.

Analysis $A[1, n]$ integer array of which we want to compute the sum. The number of steps is n.
First situation: first we load B items and process thems, then the next B items $\Rightarrow$ # I/O = n/B, which we will see often and is called the **scan cost**, because we need to see each element, in batches of B elements. So B is the size of the memory page.
But we can follow a different approach: we take the first item of each batch of B elements, then the second items and so on, which will take n steps, but this will be possibly slower because this method takes more I/Os ops, n I/O ops. The larger the jumps the more I/O ops we do. The model doesn't distinguish between local and random I/Os.

Binary Search Array of n elements. We pick the middle element, we go to left/right, middle element of the section and so on. The time complexity is $T(n) = O(\log_2 n)$, but we have a lot of I/Os and big jumps, so elements in different and far pages. We have $log_2 n/B$, because at a certain point the subarray where we search is smaller than a page, so we have $\log_2 n$ steps $- \log_2 B$ the last steps inside the page, and for the properties of the logarithms we have $\log_2 n/B$. So the larger the page the smaller the number of I/Os. One consideration: n is the number of items, B is in kilobytes.

So if we consider integers of 8 bytes, we have B/size items, and with $B = 32$ kb $= 2^{15}$ kb we have circa 4000 times, or $2^{15}/2^3 = 2^{12}$.
How can we improve the search? We can consider the B^+-trees. We split the array into the page size B and the array is sorted in ascendent order. The splits are called leaves, and each leaf has a key (one of the elements). We have a page with each key and the next element is the pointer to its page. Above one level, a page with a key of the first key list and a pointer to the key list, a key of the second and so on.
Fetch a page, binary search and follow the pointer. Number of I/Os is $\log_B n/B$ and the number of steps is a binary search for every page, so $(\log_B n/B)$ pages $\cdot \log_2 B$

Analysis $n = (1+\epsilon)M$ with $\epsilon > 0$ data outside of the memory. So M is totally full and ϵM is stored in the unbound memory.
The first point is we want to find P(accessing the disk), with a totally random algorithm, is $= \frac{\epsilon M}{n} = \frac{\epsilon \cancel{M}}{(1+\epsilon)\cancel{M}} = P(\epsilon)$

The second point is the average time of a step. $\sum_x P(X = x) \cdot x$ with X the variable of which we compute the average with that formula. X is time, in this case. The time is 1 in case of computing, and varies in case of accessing the memory. So we have the multiply the cost of access times the probability (of computing, going internal, going to the disk). Internal memory access costs 1, while on disk is larger and we say that the cost is c. So the average, with let's say a probability of memory access, $= (1-a) \cdot 1 + a(P(\epsilon) \cdot c + (1 - P(\epsilon)) \cdot 1)$ but $0 < a < 1$ and $1 - P(\epsilon)$ are very small, so we can rewrite as $= a \cdot P(\epsilon) \cdot c = a \cdot \frac{\epsilon}{1+\epsilon} \cdot c + O(1)$. If $a = 0$ then no memory access so the cost is constant. The larger is a the more memory access, the more the term is important, which is exactly what we want to capture. Usually, $a = 0.3 = 30\%$ and $c = 10^6$ the gap between accessing the disk and accessing the internal memory.
So $\frac{\epsilon}{1+\epsilon} \cdot 0.3 \cdot 10^6 = \frac{\epsilon}{1+\epsilon} \cdot 300000$. If $P(\epsilon) = 0.001$ the avg time of a step is $0.001 \cdot 300000 = 300$, so the disk has a lot of impact even with only a thousandth of memory access being on disk: the avg cost is 300 and not 1.

Sorting and permuting Given an array $S[1, m]$ and a permutation π, the permutation problem asks to permute S according to π. For example $S = [A, B, C, D], \pi = [3, 1, 2, 4]$ then π tells that that the first item goes to position $\pi[0] = 3$. So $S_\pi = [B, C, A, D]$.

```
for i = 1 to n:
        S[pi[i]] = S[i]
```

Which costs $\Theta(n)$

	PERM	SORT
RAM	n	$n \log n$
2-level memory		