1.1 Administratas

www.cs.tau.ac.il/~shpilka/courses/alg2017.html

redirect from nyu courses page

Office hours: Tue 5-6, Wed 7-8 CIWW 403

Grade: (tentative) 30% HW, 30% MIDTERM, 40% FINAL EXAM

HW: Weekly

Book: Introduction to Algorithms / CORMEN, LEISERSON, RIVEST, STEIN

MID-TERM: Tentative date October 25

TA: SIDDHARTH KRISHNA

This course is about the design and analysis of algorithms for important computational problems.

So what is an algorithm: very roughly, it is a procedure to solve some computational problem.

We will ~~focus on~~ study several important problems including:

- Sorting
- Fast multiplication
- Fast linear algebra
- Algorithms on graphs
- Some Data Structures

And we will see several techniques:

- Divide and Conquer
- Dynamic Programming
- Greedy

Finally, we will also speak about ~~limitation~~ limits of efficient algorithms and will talk about the class NP

Goals: Enjoy! Beautiful and important topic.

Rigorous thinking about algorithms, analysis

Some problem solving skills

And of course to learn the techniques, ideas, algorithms taught in this course.

Modelling, Power of reductions

So let us start by describing some of the main characters of our story

We shall design algorithms using different techniques for different goals.

Here are some problems we will look at:

Sorting: Given a sequence of integers $a_1, a_2, \ldots, a_n$ our goal is to find a permutation/reordering $a_{i_1}, \ldots, a_{i_n}$ s.t. $a_{i_1} \le a_{i_2} \le \ldots \le a_{i_n}$

E.G. given 13, 58, 19, 25, 81 we should output:

13, 19, 25, 58, 81

Shortest Path: Given a map with distance between pairs of adjacent intersections, find the shortest path from point A to point B

Integer multiplication: Given two n-digit numbers $a_1 \cdots a_n$, $b_1 \cdots b_n$ compute their product

We shall now see our first algorithm (for sorting) ~~we~~ it will demonstrate several important themes and concepts: pseudo code, proof of correctness, running time analysis, asymptotic behavior.

1.3 Sorting: Input: n integers $a_1, a_2, \dots, a_n$

Output: a reordering $a_{i} \le \dots \le a_{i}$

Imagine a ~~deck~~ set of cards ~~in your hand~~ ~~on the table. We pick one by one~~. How would you sort.
Naturally we can first sort the first two cards. Then put the 3rd in its place among them, then the 4th, 5th etc

To write this in a more formal way we shall use a pseudocode. It is not a code in any programming language yet it should be clear to you how to transform a pseudocode to a code in your favorite PL.

We now present the pseudocode for the procedure described above
We shall think of our numbers/input as an array $A[1..n]$ with elements $A[1], \dots, A[n]$. The alg. we described is called Insertion sort.

Insertion - Sort

```
For j = 2 to A.length                 (j is the number of the card we pick)
    key = A[j]
    i = j-1                           (we assume that we sorted A[1...j-1]
    While i>0 and A[i] > key           and find the place for A[j]
        A[i+1] = A[i]                 (so far we thought that key should
        i = i-1                        be in i+1 position, but it is too
    A[i+1] = key                       small)
```

It should be intuitively clear that the procedure does what it should.
How do we prove such a claim?

1.4

How do we argue about algorithms?

In our example we look for Loop Invariants. That is, a property that holds throughout the execution of the algorithm.

Our loop inv.: At the start of each iteration of the FOR loop, the subarray $A[1...j-1]$ consists of the original elements of $A[1...j-1]$, but in sorted order.

To prove that this is true we must show that it holds at the beginning, then show that if it was true ~~at the~~ before the iteration of the loop then it is true before the next one. ~~and, that it holds~~ ~~t~~

Initialization: Before the first loop iteration the inv. holds. Indeed, when $j=2$ (the moment before we start the iteration) $A[1..1]$ is $A[1]$ and it is the original $A[1]$ and it is sorted.

Maintainence. Notice that until we find the right spot for $A[j]$ we push $A[j-1]$, $A[j-2]$, .. to the right. Then insert $A[j]$ to the right location.

Formally we have to introduce a loop inv. for the while loop (e.g. $A[1...i]$ original sorted $key < A[i+2]$, $A[i+2 ... j]$ sorted and used to be in locations $A[i+1 .. j-1]$)

Termination: the loop ends ~~when~~ after $j=n$. That is we ran the loop for $j=n$ and stop ~~for j=~~ when $j=n+1$. At this point $A[1..j-1]$ is $A[1..n]$ and the inv. guarantees sortedness.

We argued about correctness. What about running time?

Let us denote $n = A.length$

For each $j = 2, 3, 4, \dots, n$ let t_j be the time the while loop required.

The cost, for each j, of lines 1..3 is constant

Thus the running time is $\sum_{j=2}^{n} c + t_j = c \cdot (n-1) + \sum t_j$

What is t_j? Well, if key > A[j-1] then we don't execute it.

But what if A[j] < A[1]? then we go through j-1 steps,

each taking a constant time. I.e. $c' \cdot (j-1)$

Thus running time is $c \cdot (n-1) + \sum_{j=2}^{n} c'(j-1) = c \cdot (n-1) + c' \cdot \frac{n(n+1)}{2} - 1 =$

$a \cdot n^2 + b \cdot n + c''$

in the worst case it is a quadratic function of n.

in the best case it is a linear function of n.

What about average case? if the numbers were randomly permuted.

informally, we expect that half the elements in A[1..j-1] will be larger than A[j] thus running time is still quadratic in this case.

This time around we more or less calculated running time exactly, but the important thing is the rate of growth. In our case it is quadratic. Lower order terms are relatively insignificant for large values of n

Final note: Insertion-Sort is an in-place alg. only a constant number of additional memory cells used for sorting.

1.6 As we said, we are mostly interested in understanding the rate of growth of the running time. We will now introduce some notation that will help us capture the relative performance of different algorithms.

We would like to have notations that will enable us to compare functions $f(n)$, $g(n)$ to each other and that will demonstrate that, roughly, an^2+bn+c is quadratic, i.e. similar to n^2, and that it is much larger than, say, $n\log n$, and much smaller than 2^n.

Def:

Given a function $g: \mathbb{N} \to \mathbb{R}$ we define the set

$$\Theta(g(n)) = \{ f: \mathbb{N} \to \mathbb{R} : \exists\, 0 < c_1, c_2, n_0 \text{ s.t. } \forall n \geq n_0 \quad c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \}$$

I.e. for large enough n $f(n)$ and $g(n)$ are equal up to a constant factor.

For example, for any $a>0, b, c$ $an^2+bn+c \in \Theta(n^2)$

We will also abuse notation and say that $f(n) = \Theta(g(n))$

It is not hard to see that $f(n) \in \Theta(g(n))$ iff $g(n) \in \Theta(f(n))$

$\Theta(\cdot)$ captures asymptotic equivalence. Next we define asymptotic upper and lower bounds.

$$O(g(n)) = \{ f : \exists\, 0<c, n_0 \text{ s.t. } \forall n \geq n_0 \ 0 \leq f(n) \leq c \cdot g(n) \}$$

$$\Omega(g(n)) = \{ f : \exists\, 0<c, n_0 \text{ s.t. } \forall n \geq n_0 \ f(n) \geq c \cdot g(n) \}$$

Claim: $f \in O(g)$ iff $g \in \Omega(f)$

1.7

The Θ notation tells us that $n = \Theta(n^2)$ but it is actually much smaller. This can be expressed by using $o(\cdot)$:

$o(g(n)) = \{ f : \forall c > 0 \ \exists n_c \text{ s.t } \forall n > n_c \ 0 \le f(n) \le c \cdot g(n) \}$

I.e. f grows smaller than any const. multiple of g.

Similarly
$\omega(g(n)) = \{ f : \forall c > 0 \ \exists n_c \text{ s.t } \forall n > n_c \ f(n) \ge c \cdot g(n) \}$

Claim: $f \in o(g)$ iff $g \in \omega(f)$

<u>Some easy properties:</u>
Transitivity: if $f \in \Theta(g)$, $g \in \Theta(h)$ then $f \in \Theta(h)$.
same for O, Ω, o, ω
Reflexivity: $f \in \Theta(f), O(f), \Omega(f)$
~~Symmetry~~
$f \in O(g)$ iff $g \in \Omega(f)$
$f \in o(g)$ iff $g \in \omega(f)$

Important classes of functions:
polynomials: $f(n) = \sum_{i=0}^{d} a_i \cdot n^i \quad , a_d \neq 0$
$f(n) \in \Theta(n^d)$
Exponentials: $f(n) = a^n$
Claim: $\forall a > 1, \forall d > 0 \quad n^d = o(a^n)$
logarithms: ~~log~~ $f(n) = \log(n)$
$\log(n) = o(n^a) \quad \forall a > 0$