2.0

Last class we discussed some administrative issues.

A few more clarifications:

- Bonus questions: Grade from this questions will be counted towards your final grade, but even if you do not answer them you can get 100% score for HW. Its main aim is challenge and exposure to new material.
- HW submission: Those who wish to submit HW online, please do so using NYU classes.

This class:

- Big Oh notation
- Merge sort, divide and conquer
- Solving recursion formulas, the master theorem

2.1 #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

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

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.

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

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

2.2

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

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

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

Similarly

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

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

Some easy properties:

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: $f \in \Theta(g)$ iff $g \in \Theta(f)$

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

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

Important classes of functions:

polynomials: $f(n) = \sum_{i=0}^{d} a_i \cdot n^i$, $a_d \neq 0$

$f(n) \in \Theta(n^d)$

Exponentials: $f(n) = a^n$

Claim: $\forall a > 1$, $\forall d > 0$ $n^d = o(a^n)$

Logarithms: $\log f(n) = \log g(n)$

$\log(n) = o(n^a)$ $\forall a > 0$

Thus, the running time of insertion sort is $\Theta(n^2)$ in the worst case (or even on average).

Let us now present a new algorithmic idea and see how to apply it to our sorting problem.

The idea is given a problem divide it to smaller subproblems (divide), solve each of them and then combine these solutions to obtain a solution to the original problem (conquer). The natural and self-defining name of this approach is divide and conquer and we shall see several examples of it.

Merge sort

So let us look at the sorting problem again. What could be a natural division to sub problems? First let us sort the first $n/2$ elements, then the last $n/2$ elements and at the end when we have the sorted lists, combine them to one.

Thus, the meta algorithm is:

- Merge-sort $A[1 \ldots n/2]$
- Merge-sort $A[\frac{n}{2}+1 \ldots n]$
- Merge $A[1 \ldots \frac{n}{2}]$ and $A[\frac{n}{2}+1 \ldots n]$

Let us start by something simple - how to merge two sorted lists. The idea is to start scanning them from the smallest elements and each step put the smaller of the two elements we see into the final list.

2.7

Merge: Let $A[1\ldots k]$ and $B[1\ldots m]$ be two arrays, not necessarily of the same length. For simplicity, similarly to what we did last time, let us set $A[k+1] = \infty$, $B[m+1] = \infty$ where ∞ can be understood as a symbol for a value larger than all other values.

```
Merge(A,k,B,m,C)
1.  i=1
2.  j=1
3.  For k=1 to k+m
4.      If A[i] ≤ B[j]
5.          C(k) = A[i]
6.          i = i+1
7.      else C[k] = B[j]
8.          j = j+1
```

To show that the alg. returns a merged union of A,B we shall use the following invariant:

At the start of each iteration of the FOR loop, the subarray $C[1\ldots k-1]$ contains the $k-1$ smallest elements of $A[1\ldots k]$ and $B[1\ldots m]$. Moreover, $A[i], B[j]$ are the smallest elements of their arrays that have not been copied to $C[1\ldots k-1]$

Proof: Initialization: when $k=1$ $C[1\ldots 0]$ is empty and $k-1=0$ so we are ok.

Maintainance: assume wlog that $A[i] \le B[j]$. Since $A[i]$ the smallest element not copied (yet) to $C[1\ldots k-1]$, after line 5, $C[1\ldots k]$ contains the k smallest elements. After line 6 the claim regarding $A[i], B[j]$ still holds.

Termination: At the end $C[1\ldots k+m]$ contains all the elements and is sorted.

What is the running time?

We have $k+m$ steps each taking $O(1)$ time. Thus, $O(k+m)$.

Let us now go back to our meta algorithm:

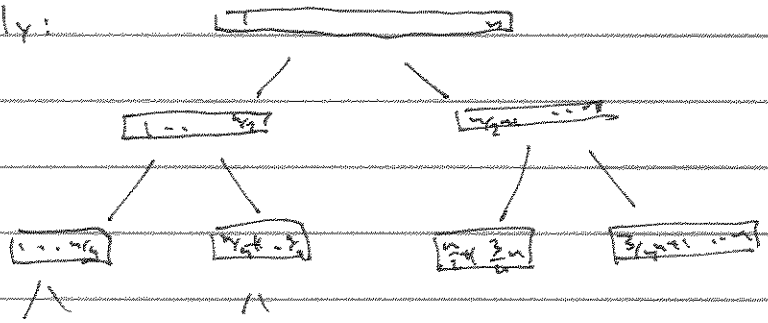
- merge-sort $A[1 \dots \frac{n}{2}]$
- merge-sort $A[\frac{n}{2}+1 \dots n]$
- merge $A[1 \dots \frac{n}{2}]$, $A[\frac{n}{2}+1 \dots n]$

We have seen that the last step can be done in time $O(n)$.

What about the first two steps?

Notice here is a recursion. Because When we execute $A[1 \dots \frac{n}{2}]$ we actually run merge-sort $A[1 \dots \frac{n}{2}]$, merge-sort $A[\frac{n}{2}+1 \dots n]$, merge( ) etc.

Pictorially:



To analyze the running time we will need to solve a recurrence equation.

Let us say that it takes $T(n)$ time to run merge-sort on n-elements array.

We get that

$$T(n) = 2T(\tfrac{n}{2}) + \Theta(n)$$

$2T(\frac{n}{2})$: each of the merge sorts

$\Theta(n)$: the merge

How do we solve such an equation?

We will later study a general theorem for solving recurrences.

For now let us try to get some intuition:

$T(n) = 2T(\frac{n}{2}) + \Theta(n)$

$\Rightarrow T(n) = 2\left(2T(\frac{n}{4}) + \Theta(\frac{n}{2})\right) + \Theta(n)$

$= 4T(\frac{n}{4}) + 2\Theta(\frac{n}{2}) + \Theta(n)$

(where the constants in $\Theta(\cdot)$ are the same. I.e. $\Theta(n) \leq c_1 n$ $\geq c_2 n$

$\leq 4T(\frac{n}{4}) + 2c_1 n$

$\geq 4T(\frac{n}{4}) + 2c_2 n$

$\Rightarrow T(n) = 4\left(2T(\frac{n}{8}) + \Theta(\frac{n}{4})\right) + 2\Theta(\frac{n}{2}) + \Theta(n)$

$\leq 8T(\frac{n}{8}) + 3c_1 n$

$\geq 8T(\frac{n}{8}) + 3c_2 n$

After $\log_2^k n$ steps

$T(n) = 2^k T(\frac{n}{2^k}) + 2^{k-1}\Theta(\frac{n}{2^{k-1}}) + 2^{k-2}\Theta(\frac{n}{2^{k-2}}) + \ldots + \Theta(n)$

$\leq 2^k T(\frac{n}{2^k}) + k \cdot c_1 n$

$\geq 2^k T(\frac{n}{2^k}) + k \cdot c_2 n$

After $\log n$ steps

$T(n) = nT(1) + \frac{n}{2}\Theta(2) + \ldots + \Theta(n)$

$\leq nT(1) + c_1 n \log n = O(n \log n)$

$\geq nT(1) + c_2 n \log n = \Omega(n \log n)$

$\Rightarrow T(n) = \Theta(n \log n)$

Master Theorem

More generally:

Theorem (Master Theorem):
Let $a \geq 1$, $b > 1$ be constants, $f(n)$ be a function, and $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n)$$

(where n/b is either $\lceil n/b \rceil$ or $\lfloor n/b \rfloor$) then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \varepsilon})$ for some $\varepsilon > 0$ then $T(n) = \Theta(n^{\log_b a})$
2. If $f(n) = \Theta(n^{\log_b a})$ then $T(n) = \Theta(n^{\log_b a} \cdot \lg n)$
3. If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some $\varepsilon > 0$, and if $af(n/b) \leq cf(n)$ for some const. $c < 1$ then $T(n) = \Theta(f(n))$

E.g. if $T(n) = 2T(\frac{n}{2}) + \Theta(n)$ then $a = b = 2$, $f(n) = \Theta(n) = \Theta(n^{\log_b a})$
and $T(n) = \Theta(n^{\log_2 2} \cdot \lg n) = \Theta(n \lg n)$

The intuition: ignore f for a moment. if $a > b$
we expect will get $T(n) = n^{\log_b a} \gg n$
if $a < b$ $T(n) = n^{\log_b a} \ll n$
Now add back f. If f is much smaller than these quantities (case 1) then it can be ignored.
If f is much larger (case 3) then it dominates the recursion.
If f is roughly $n^{\log_b a}$ then as in mergesort we have $\log_b n = \Theta(\lg n)$ recursion steps and we get $\log n \cdot n^{\log_b a}$.