Administration

Tom Plick: TA for on-campus students, Tuesdays 16:00-18:00
Prof. Ali Shkoufandeh: room 141, Tuesdays 17:00-18:00

- Assignments are posted from the 3rd edition of the book, submission online or in class.
- Both midterm and final exams are 3-4 hours.
- Midterm exam: closed-book, one page allowed. Will concise 30% of final grade.

Notes to self:

Cover skip-lists

Problem:
Language - collection of valid strings. For instance, the language of sorted sequences:
Given a sequence: $a_1, a_2, \dots, a_n$, it is considered **sorted** if it satisfies: $a_1 \leq a_2 \leq \cdots \leq a_n$
So the language of sorted sequences is $\{< a_1, \dots, a_n > \mid a_1 \leq \cdots \leq a_n\}$
To verify if a sequence is **sorted**, i.e. in the language, you design an algorithm that outputs either **yes or no** for an input sequence.
This is a *decision problem*: $s \stackrel{?}{\in} L$ – for it you design an algorithm: $A(s) = yes, s \in L \; OR \; no, o.w.$
Another problem would be an *optimization problem*.

Insertion Sort:

Given an initially not sorted sequence, the algorithm sorts the sequence in the following way:
The initial condition is: the sequence is unsorted. There is a sorted part in the array which is initially is empty.
The outer loop takes a misplaced element, pushes it as far as possible in the sorted part.
The invariance of the loop is: in every iteration the sorted part grows and the unsorted part shrinks. So this is a finite algorithm.
The finite state of the sequence is that the array is sorted.
What about complexity?
In the worst case, each element i is to be pushed $i-1$ steps to the left (with initial backwards sorted sequence). And so the **time** complexity is $T(n) = O(n^2)$
The **space** complexity of the algorithm is $S(n) = O(n)$
But, this is not the most efficient algorithm for sorting, as there are algorithms that merge in $O(n \log n)$.

Prime Factorization:

$z = a \times b, \; a, b \neq 1, z \; \Rightarrow z$ is not prime.
This is a *decision problem*: given x, is x prime?
Brute force method:
$for \; i = 1 \; to \; x - 1$:
    $if \; x \mid i \; then \; x \; is \; not \; prime$
Better would be to get to $\sqrt{x}$ instead of x, since if x has non-trivial factors, one of them is definitely $\leq \sqrt{x}$.
The complexity over the size of the input: $n = \lceil \log_2 x \rceil \Rightarrow x \cong 2^n$ and so if the complexity is $\sqrt{x}$ it is actually $\sqrt{2^n} = 2^{\frac{n}{2}}$, so the time complexity of the problem is **exponential**.

Another decision problem:

A decision problem: given $F = \{a_1, \dots, a_n\}, is\ x \in F$? That maps for instance to finding a file in a filesystem.
The time complexity of the problem, the *worst case* is n – you have to go over the entire set until you find x. The best case is constant, but it doesn't happen very often…
So the above is the **lower bound** of the **complexity of the problem** (not the algorithm). That is, no non-trivial algorithm exists that can solve this problem in less than n time.

Data Structure:

Data structures are designed to allow efficient performance for accessing data.

For the file search problem above, we'll define a permutation: $\sigma(F) =< \tilde{a}_1 \leq \cdots \leq \tilde{a}_n >$
This data structure allows **binary search**:
You look at the total array first, and compare x with $a_{\frac{n}{2}}$, and after only **one** operation you eliminated $\frac{n}{2}$ of the elements. The next step would recursively continue to the segment where x is at. The second step eliminates $\frac{n}{4}$ elements. In general, after the i comparison we eliminated $\frac{n}{2^i}$ elements. Therefore $\frac{n}{2^k} = 1$ when $n \cong 2^k \Rightarrow k = \log_2 n$ steps.
And so the data structure allowed us performing a task faster.

Running time analysis:

You want the worst case which is $\max T(n)$.
You can also look at the expectancy: $E[T(n)]$

Back to insertion sort:
The best behavior of the algorithm is for an already sorted sequence. In that case the complexity is n.
We got to the complexity: $T(n) = n + \underbrace{\big((n-1) + (n-1) + (n-1)\big)}_{initialization} + 3\sum_{j=2}^{n}(t_j - 1)$, where the last term is $\frac{n(n+1)}{2}$ as main term. Eventually, we only care of the main term asymptotically.
For instance: $f(n) = 5n^4 + 10n^2 - 3n + 2 = \Theta(n^4)$, i.e. as $n \to \infty$, $\Theta(n^2)$ and the others grow slower than $\Theta(n^4)$.
Mathematically:
$\lim_{n\to\infty} \frac{5n^4+10n^2-3n+2}{n^4} = 5$, that is asymptotically the numerator grows 5 times faster than the denominator.
Whereas: $\lim_{n\to\infty} \frac{5n^4\dots}{n^2} = \infty$

Θ, O *and* Ω notations:

$$T(n) \cong \sum_{j=2}^{n}(j) \cong \Theta\left(\sum_{j=2}^{n} j\right) \cong \Theta(n^2)$$

Θ means: the term on the left of the equation grows as fast as the term in the Θ.
$\Theta(1) = \Theta(n) = 100n^0$
$\Theta(1) + \Theta(1) = \Theta(1)$
But! $\underbrace{\Theta(1) + \Theta(1) + \cdots + \Theta(1)}_{n\ times} = \Theta(n)$

Definition: big O:

$f(n) = O(g(n)) \Leftrightarrow \exists constant\ c, n_0\ s.t. \forall n \geq n_0 : 0 \leq f(n) \leq c \cdot g(n)$

So there exists a constant (at least one) and a breakpoint such that after that breakpoint, f grows slower than $c \cdot g$.

For instance: $100n = O(n^2)$

The O notations says that $\lim_{n\to\infty} \frac{g(n)}{f(n)} > 0$

Big O is like " $\leq$ "

Definition: small o:

$f(n) = o(g(n)) \Leftrightarrow \forall constant\ c\ \exists n_0\ s.t. \forall n \geq n_0 : 0 \leq f(n) < c \cdot g(n)$

No matter what constant choose, there is a break point from which on $c \cdot g(n)$ is strictly bigger than $f(n)$, that is $g(n)$ grows "much" faster than $f(n)$.

Mathematically: $\lim_{n\to\infty} \frac{g(n)}{f(n)} = \infty$ - $g(n)$ grows much faster than $f(n)$, asymptotically.

For instance: $n^3 = o(n^7)$ since $\lim_{n\to\infty} \frac{n^7}{n^3} = \infty$

o is like " $<$ "

Definition: big Omega – Ω:

$f(n) = \Omega(g(n)) \Leftrightarrow \exists constant\ c, n_0\ s.t. \forall n \geq n_0 : 0 < c \cdot g(n) \leq f(n)$

This is the opposite of the O notation: $n^3 = O(n^5) \Leftrightarrow n^5 = \Omega(n^3)$

Ω is like " $\geq$ "

Definition: small omega: ω:

$f(n) = \omega(g(n)) \Leftrightarrow \forall constant\ c, \exists n_0\ s.t. \forall n \geq n_0 : 0 \leq c \cdot g(n) < f(n)$

So if $f(n) = \omega(g(n))$ then $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$

ω is like " $>$ "

Definition: big theta – Θ:

$f(n) = \Theta(g(n)) \Leftrightarrow \exists const.\ c_1, c_2, n_0\ s.t. \forall n \geq n_0 : 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$

If $f = O(g), f = \Omega(g)$ then $f \equiv g : f = \Theta(g)$

If $f = \Theta(g)$ then $\lim_{n\to\infty} \frac{g}{f} = constant$

For instance: $\lim_{n\to\infty} \frac{10n^5+5n}{n^5} = \lim_{n\to\infty} 10 + \frac{5}{n^4} = 10$

$f = \Theta(g) \Rightarrow f = O(g)$ and $f = \Omega(g)$

Having the possibility to compare functions is non-trivial always (such as sin or cos).

Problem (quiz):

Given a sequence $a_1, \dots, a_n$, find max and 2-max in $n + \log(n)$ time (infinite space):

- First compare each pair, totally comparing $\frac{n}{2}$ comparisons.
- Do the same one level up - $\frac{n}{4}$ more comparisons, and so on until you get to the top – which is the max. the total operations are $\frac{n}{2} + \frac{n}{4} + \cdots + \frac{n}{2^{\log n}}$
- As we go up in the structure, each one you bit to the top – remember in a list (infinite list). Since you go up $\log n$ steps, the max has 2-max in his list, which is of length $\log n$.

Therefore: $n + \log n$

If you also want the 3-max, you'll spend $n + 2 \cdot \log n + \log\log n$ since after the first $n + \log n$ where we find max and 2-max, we do the same for the $\log n$-list of max (which contains 2-max and 3-max), and that goes for another "$log[n + \log\ n]$, which is $\log n + \log\log n$.

Divide and Conquer:

Breaking a problem into sub problems, solve them, and merge the solutions.

For instance, Hanoy tower problem:

The simplest case contains 3 polls and 3 disks sorted by descending size on one poll, and you wish to move all disks to a different poll, keeping that every disk is placed only on bigger disk than it.

The solution is D&C:

- Solve the problem for the top 2
- Move the 3rd to the desired poll
- Solve the problem again for 2

In general, to solve a n-disks Hanoy tower, we will solve the problem for $n - 1$, and so on and so forth. So to solve one problem of size n you solve 2 problems of size $n - 1$ and make one move:

$T(n) = 2T(n-1) + 1 = 2[2T(n-2) + 1] + 1 = 2^2T(n-2) + 2^1 + 2^0 = 2^2[2T(n-3) + 1] + 2^1 + 2^0 =$

$2^3T(n-3) + 2^2 + 2^1 + 2^0 = \cdots = 2^kT(n-k) + \sum_{i=0}^{k-1} 2^i = \cdots$

And when $n - k = 1$ you stop, that is $k = n - 1$ and so: $T(n) = 2^{n-1}T(1) + \sum_{i=0}^{n-2} 2^i$ where $T(1) = 1$

$\Rightarrow T(n) = 2^{n-1} + 2^{n-2} + \cdots + 2^0 = 2^n - 1$ operations.

And so, the **D&C algorithm is exponential**.

The **binary search** is also a type of D&C. The recurrence:

$T(n) = T\left(\frac{n}{2}\right) + 1 = T\left(\frac{n}{4}\right) + 1 + 1 = \cdots = T\left(\frac{n}{2^k}\right) + k = \cdots \Rightarrow$

$\left[\frac{n}{2^k} = 1 \Rightarrow k = \log_2 n\right]$

$\Rightarrow T(n) = T(1) + \log_2 n$

The **insertion sort** is also recursive (not in the classical sense). The recurrence is:

$T(n) = T(n-1) + 1 = T(n-2) + 2 + 1 = T(n-3) + 3 + 2 + 1 = \cdots = T(n-k) + \sum_{i=1}^{k} i \Rightarrow$

$[n - k = 1 \Rightarrow k = n - 1]$

$\Rightarrow T(n) = (n-1) + (n-2) + \cdots + 2 + 1 = \frac{n(n-1)}{2}$

Merge Sort:

To sort a sequence of size n:

- Divide the sequence into 2 sequences of size $\frac{n}{2}$
- Apply the algorithm recursively on each of the two parts to sort them.
- After the 2 parts are sorted:
    - Prepare an empty array, and have 2 pointers to the beginnings of the sorted parts.
    - Each time take the minimum of the 2 pointers and advance that pointers

  Total: n for this part

Recurrence:

$T(n) = 2T\left(\frac{n}{2}\right) + n = 2 \cdot \left[2T\left(\frac{n}{4}\right) + \frac{n}{2}\right] + n = 2^2\left[2T\left(\frac{n}{8}\right) + \frac{n}{4}\right] + \frac{n}{2} + n = \cdots = 2^k T\left(\frac{n}{2^k}\right) + k \cdot n, k = \log_2 n \Rightarrow$

$T(n) = 2^{\log n} T(1) + n \cdot \log n = n + n \cdot \log n$

Summary:

- Hanoy: $T(n) = 2T(n-1) + 1 = \Theta(2^n)$
- Merge Sort: $T(n) = 2T\left(\frac{n}{2}\right) + n = \Theta(n \log n)$

The small difference above – how much is left for "future actions" – can have great implications over the complexity. Hanoy leaves almost all for the next iteration, Merge sort is not.