3.0

Last week we introduced $\Theta, O, \Omega, o, \omega$ and discussed the relations among them. Then we gave a divide and conquer algorithm for sorting, called Merge sort and ~~wrote~~ expressed its running time via a recurrence equation. We solved this equation via a technique called recursion tree (although we did not use this term). We then stated the Master Thm which helps solve recurrence equations of certain form.

This week:
More divide and conquer algorithms.
- Quick sort
- ~~Karatsuba~~ Binary search
- Karatsuba's integer multiplication alg.
- Strassen's fast Matrix multiplication alg.

Remark: please read part VIII of CLRS for the required math background!

Quick-sort.

Quick sort is another sorting algorithm. Its running time in the worst case is $\Omega(n^2)$.
So why introduce another slow sorting alg. when we already have Merge-sort?
The point is that Q.S. takes on average $O(n\lg n)$ steps.
OK, but we still have Merge-Sort so what's the idea?
Q.S. is a simple to implement in-place sorting alg. The pseudocode we gave for M.S. isn't in place. It can be made in-place with some effort but this complicates the code. ~~Last week there was~~

:

Note: Last week a suggestion was made to make M.S. in-place but that idea had a bug in it that I didn't notice when answering the question. If anyone is interested they can contact me and ask.

Anyway, Q.S. is simple in-place alg. and in practice it runs pretty well and is usually implemented instead of M.S.

So what's the idea:
The alg. picks an element that is called a pivot and moves all elements smaller than the pivot to its left and all elements larger then it to its right and then recursively sorts the left and right parts.

There are 3 parts for the alg:
1. Selecting the pivot
2. Partitioning the elements to those smaller than it and those larger than it
3. Recursion

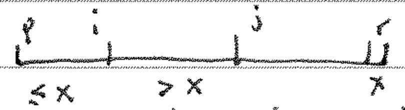
3.2

Lets start with the simplest selection - that of the last element.
The pseudocode for quick-sorting a subarray A[p...r] is given by:

```
Quick-sort(A, p, r)
1. If p<r
2.      q = Partition(A, p, r)      (here we partition the elements)
                                    q is the correct location of the pivot
3.      Quick-sort(A, p, q-1)
4.      Quick-sort(A, q+1, r)
```

To sort A we run Quick-sort(A, 1, A.length)
Next we describe the partition alg.

```
Partition(A, p, r)
1. x = A[r]                         x is the pivot
2. i = p-1                          i will be the last element
                                    smaller than x
3. For j = p to r-1
4.      If A[j] ≤ x
5.          i = i+1
6.          Swap A[i] with A[j]
7. Swap A[i+1] with A[r]            puts the pivot in place
8. Return i+1
```

Idea: at each step the picture is 

The loop invariant is that at the beginning of each iteration, for any index k
1. if $p \le k \le i$ then $A[k] \le x$
2. if $i+1 \le k \le j-1$ then $A[k] > x$
3. if $k = r$ then $A[k] = x$

This is indeed what the picture tells us and it can be verified quite easily.

Run-time analysis: (of partition)
~~[illegible]~~ Notice that each execution of partition takes linear time. We simply run over all elements and at each step perform a constant number of simple operations.

Back to Q.S. What can we say about its running time?
(Correctness is simple to verify given that partition is correct).

Worst case: Imagine that the array is already sorted! Then, each ~~step~~ execution of partition will return an array smaller by 1 and an empty array. As Partition takes linear time, overall we have $c \sum_{i=1}^{n} i = \Theta(n^2)$

What happens in the best case? In this case Partition returns two subarrays of roughly equal sizes and we get the recurrence
$T(n) \le 2T(\frac{n}{2}) + \Theta(n)$
which gives $T(n) \le \Theta(n \lg n)$ as we saw.

What is the typical case? If $A[n]$ is random then we expect to get a balanced partition. Notice that even an unbalanced, but still somewhat balanced partition to $0.9n$ and $0.1n$ gives a recurrence
$T(n) = $ ~~$T(0.9n)$~~ $T(9n/10) + T(n/10) + \Theta(n)$.
Solving this still gives $T(n) = \Theta(n \lg n)$
(there are $\le \log_{10/9} n$ levels of recursion steps, and cost of each step is $\Theta(n)$)

3.4

What about the average case?

For a random input we expect the partition to be roughly balanced. Notice that even if half the steps give an unbalanced partition (which is actually unlikely) then the running time is still $O(n\lg n)$ as the recursion depth grows by a factor of 2.

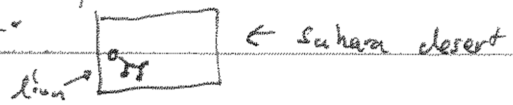
The alg. we showed has the drawback that for some inputs it performs poorly (e.g. sorted arrays).
A way to overcome this is by selecting a random index to be the pivot.

Notice that the probability that the chosen pivot is either smaller than ~~almost~~ 9/10 of the elements or larger than 9/10 is at most $\frac{2}{10} = 20\%$. Thus with good probability each step is at least (9/10, 1/10) balanced. It can also be proved that with high probability at least 1/2 of the steps are "good".

35

More examples of divide and conquer.

The simplest example is that of binary search. It is also sometimes called "how to catch a lion in the desert"



The alg. divide the desert to two parts. Then consider the half in which the lion is and catch the lion in that half of the desert.

What is the binary search problem:

Input: A sorted Array $A[1 \dots n]$ and an element e.

Goal: find i s.t. $A[i] = e$.

Idea: compare e to $A[n/2]$. If e is smaller then search $A[1 \dots \frac{n}{2}]$ o.w. search $A[\frac{n}{2}+1 \dots n]$.

Alg:

```
Binary-Search(A, e, n)
1. If A empty return Not Found
2. [illegible]
2. Else  $m = \lfloor n/2 \rfloor$
3. If  $e = A[m]$
4.      return m.
5. else If  $e < A[m]$
6.      return Binary-search($A[1 \dots m-1]$, e, $m-1$)
7.      else return Binary-Search($A[m+1, \dots, n]$, e, $n-m$)
```

Claim: Binary-Search(A, e, n) solves binary search...

Pf: By induction on n.

3.b More sophisticated examples of divide and conquer.

Karatsuba's alg. for integer multiplication.

Input: two n-digit binary numbers $a = a_1 \dots a_n$ $b = b_1 \dots b_n$

Goal: compute the binary rep. of $a \cdot b$

School alg. runs in time $O(n^2)$. Why?

Karatsuba's Idea: Notice $a = (a_{\frac{n}{2}+1} \dots a_n) + 2^{n/2} \cdot (a_1 \dots a_{n/2}) = a_\ell + 2^{n/2} a_h$

$b = (b_{\frac{n}{2}+1} \dots b_n) + 2^{n/2} \cdot (b_1 \dots b_{n/2}) = b_\ell + 2^{n/2} b_h$

~~Now~~ where a_ℓ, a_h, b_ℓ, b_h are $\frac{n}{2}$-digit numbers.

Now, $a \cdot b = (a_\ell + 2^{n/2} a_h) \cdot (b_\ell + 2^{n/2} b_h) = a_\ell \cdot b_\ell + 2^{n/2}(a_\ell b_h + a_h b_\ell) + 2^n a_h b_h$

If we naively compute $a_\ell b_\ell, a_\ell b_h, a_h b_\ell, a_h b_h$ ~~then we get~~ then we get $T(n) = 4T(\frac{n}{2}) + \overset{\uparrow}{O(n)}$

adding n-dig. numbers

Solving (e.g. Master thm) we get $T(n) = O(n^2)$, like school alg. ...

If we could save even one multiplication we would get a much faster alg.

Notice, paying an $O(n)$ cost for each step is ok as it doesn't change ~~instead compute the following products~~ the asymptotic of the running time.

Let us compute

$(a_\ell + a_h)(b_\ell + b_h) = a_\ell b_\ell + a_h b_h + (a_\ell b_h + a_h b_\ell)$ I

$(a_\ell - a_h)(b_\ell - b_h) = a_\ell b_\ell + a_h b_h - (a_\ell b_h + a_h b_\ell)$ II

$a_\ell b_\ell$ III

~~N~~Now, $a_\ell b_\ell$ is given, $a_\ell b_h + a_h b_\ell$ is obtained via $\frac{1}{2}(\mathrm{I} - \mathrm{II})$

and $a_h b_h$ via $\frac{1}{2}(\mathrm{I} + \mathrm{II}) - \mathrm{III}$

Thus, using 3 multiplications and 7 addition/subtractions we can compute $a_L b_L$, $a_H b_H$, $a_L b_H + a_H b_L$

In $O(n)$ additional time we get $2^n a_H b_H$, $2^{n/2}(a_L b_H + a_H b_L)$ and two more additions give $a \cdot b$.

Hence, $T(n) = 3T(\frac{n}{2}) + O(n)$

$\Longrightarrow \quad T(n) = O(n^{3/2})$

Fact: There is a faster alg. that we may see running in time $\sim O(n \lg n)$

Another amazing alg. is Strassen's matrix multiplication algorithm.

Recall, given $A_{n\times n}$ $B_{n\times n}$ $A=(a_{ij})$ $B=(b_{ij})$

$$(A\cdot B)_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

The simple alg. computes each (i,j) entry using ~~n~~ n mult. and $n-1$ additions.

Thus, overall n^3 mult. $O(n^3)$ additions. $(n^2 \cdot (n-1))$

For 2×2 matrices 8 mult and 4 additions.

Strassen's brilliant alg. solve 2×2 using 7 mult. and 18 additions

Actually sounds much worse. It is, for 2×2 matrices...

The brilliant part is how it helps multiplying ~~[illegible]~~ $n\times n$ matrices.

Idea: $A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$ $B = \begin{pmatrix} B_{11} & \cdot \\ \cdot & \end{pmatrix}$ ~~A_{12}~~

Now, ~~solve~~ compute $A \cdot B$ using 7 mult. of $\frac{n}{2}\times\frac{n}{2}$ matrices and 18 additions of $\frac{n}{2}\times\frac{n}{2}$ matrices.

$T(n) = 7T(\frac{n}{2}) + O(n^2)$

$\Rightarrow \quad T(n) = O(n^{\lg 7}) = O(n^{2.81..})$

3.8

We must now deliver. No need to remember the details but rather the general idea.

1. $C_1 = (A_{11}+A_{22}) \cdot (B_{11}+B_{22})$
2. $C_2 = (A_{21}+A_{22}) \cdot B_{11}$
3. $C_3 = A_{11} \cdot (B_{12}-B_{22})$
4. $C_4 = A_{22} \cdot (B_{21}-B_{11})$
5. $C_5 = (A_{11}+A_{12}) \cdot B_{22}$
6. $C_6 = (A_{21}-A_{11}) \cdot (B_{11}+B_{12})$
7. $C_7 = (A_{12}-A_{22}) \cdot (B_{21}+B_{22})$
8. $(A \cdot B)_{11} = C_1 + C_4 - C_5 + C_7$
9. $(A \cdot B)_{12} = C_3 + C_5$
10. $(A \cdot B)_{21} = C_2 + C_4$
11. $(A \cdot B)_{22} = C_1 + C_3 - C_2 + C_6$

Over all 7 mult. 18 add.