Announcements: quiz Friday, equatio, Plickers

Course Organization
- Algorithmic Paradigms
  - Divide & Conquer
  - Dynamic Programming
  - Greedy
- Applications to graph Problems
- What problems can we not solve efficiently

Recall goal:
Appreciate beauty, creativity, surprising cleverness of algorithm design. Next algorithm hopefully touches on some of these...

example:

**Multiplication** - basic operation used frequently in many applications

**Input**: 2 n-digit integers $a, b$. $a = a_{n-1} \ldots a_0 \quad (a = \sum_{i=0}^{n-1} a_i \cdot 10^i)$

$b = b_{n-1} \ldots b_0$

**Output**: Integer $c = a \times b$
Suppose multiplying/adding 2 one-digit numbers takes $O(1)$ time. Using "grade school" algorithm, what is runtime of multiplying $a$ and $b$?

A) $O(n)$  B) $O(n^2)$  C) $O(\log(n))$  D) Not enough information

Use this example to remind ourselves about runtime and big-O notation

Ex:

\[
\begin{array}{c}
3 & 2 & 1
\
6 & 2 & 9
\end{array}
\]

Step 1
Produce these numbers.
Each digit requires 1 multiplication & $\leq 1$ addition
$n^2$ times $\Rightarrow 2n^2$ operations

Step 2
Produce this number (adding n digit numbers uses $n$ time, keep a running sum) $\Rightarrow \frac{3}{2}(n^2+n)$ operations

Total: $\frac{7}{2}n^2 + \frac{3}{2}n$ $\Rightarrow O(n^2)$ runtime

Teaser: We'll be able to do in $O(n^{1.58})$ using Karatsuba multiplication!
Example brings up 2 guiding principles:

**Principles of Algorithm Analysis**

1. Use worst-case analysis. e.g. Counted 1 addition operation for carryover for every multiplication, even though might not actually use.
   - upper bounds hold
   - for all instances
   - don't need extra info about problem
   - easier

2. Use asymptotic analysis (**big-O notation**)  

Q. Why do we use big-O notation?

- Only care about big problems (otherwise do by hand or using brute force algorithm)
- and for big problems, lower order terms don't matter
- e.g. Insertion Sort: \( \frac{1}{2} n^2 \) Merge Sort: \( 6n \log n + 1 \)

- Independent of architecture

- Easier

- Predicts how time will scale as problem size increases
Divide & Conquer ( & Combine)

Split big problem into smaller versions of same problem.

Solve smaller problems via recursion

Combine solutions to get solution to big problem

Already Seen!

Merge Sort

Input: Array A of integers of size n
Output: Sorted array

\[
\text{MergeSort}(A) \quad \text{if } \text{length}(A) = 1 : \text{return } A \quad \text{3 base case}
\]

\[
A_1 = \text{MergeSort}(A[1; \lfloor n/2 \rfloor])
\]

\[
A_2 = \text{MergeSort}(A[\lceil n/2 \rceil + 1; n])
\]

\[
p_1 = p_2 = 1
\]

for \(i = 1\) to \(n\)

if \(A_1[p_1] < A_2[p_2]\)

\[
A[i] = A_1[p_1]
\]

\(p_1++\)

else

\[
A[i] = A_2[p_2]
\]

\(p_2++\)

return A
Q: Create a Divide & Conquer Algorithm for Multiplication
(a, b n digit numbers, n is a power of 2)

Base Case: If 1 digit numbers, can do in $O(1)$ time

Divide: $a = (a_{n-1} \ldots a_0) \cdot 10^{n/2} + (a_{n-2} \ldots a_0)$ & $a', a'', b', b''$
$b = (b_{n-1} \ldots b_0) \cdot 10^{n/2} + (b_{n-2} \ldots b_0)$ each $\frac{n}{2}$ digits.

E.g.: $5284 = 5 \times 10^3 + 2 \times 10^2 + 8 \times 10 + 4$
      $= 52 \times 10^2 + 84$

$a \cdot b = (a' \cdot 10^{n/2} + a'') \cdot (b' \cdot 10^{n/2} + b'')$
      $= a' \cdot b' \cdot 10^n + (a'' + a' b') 10^{n/2} + a'' b''$

Conquer: Solve smaller problems: $a' b', a'' b', a' b', a'' b$

Combine: Add $a' b' \cdot 10^n + (a'' b' + a' b'') 10^{n/2} + a'' b''$
Q: How many \( 1 \text{ digit} \times 1 \text{ digit} \) multiplication problems will this algorithm require? \((n \text{ digit} \times n \text{ digit})\)

A) \(\log_2 n\)  
B) \(n\)  
C) \(n^2\)  
D) \(2^n\)

Each level increases by factor of 4:

\[4 \cdot 4 \cdot 4 \cdots 4 \cdot \underbrace{4 \log_2 n}_{\text{log}_2 n} = (2^4)^{\log_2 n} = 2^{2^{\log_2 n}} = (2^{\log_2 n})^2 = n^2\]

This is bad! There are at least \(n^2\) basic operations, so time complexity is \(\Omega(n^2)\). No better than regular!