1 Integer Multiplication

Input: 2 n-digit numbers x and y

Output: product $x \times y$

Primitive operation: add or multiply 2 single-digit numbers. n^2 number of operations n operations per row up to a constant

1.1 can we do better?

recursion, split the numbers in 2

write $x = 10^{\frac{n}{2}}a + b$ and $y = 10^{\frac{n}{2}}c + d$, where a, b, c, d are $\frac{n}{2}$ digit numbers

Example:

$a = 56$, $b = 78$, $c = 12$, $d = 34$

Then:

$x \times y = (10^{\frac{n}{2}}a + b)(10^{\frac{n}{2}}c + d) = 10^n ac + 10^{\frac{n}{2}}(ad + bc) + bd$

Idea:

recursively compute ac, ad, bc, bd, then compute (*) in the obvious way.

1.2 Karatsuba Multiplication

$x \times y = 10^n \mathbf{ac} + 10^{\frac{n}{2}}(\mathbf{ad} + \mathbf{bc}) + \mathbf{bd}$

There are really just 3 quantities that we care about

- Recursively compute ac
- Recursively compute bd
- recursively compute $(a + b)(c + d) = ac + bd + ad + bc$
  - Gauss's trick: The result of the third recursive call minus first and second $= ad + bc$
    - Upshot: only need 3 recursive multiplications (and some additions)

1.3 Merge Sort: Motivation and Example

1.3.1 Why Study Merge Sort?

- Good introduction to divide & conquer
  - Improves over Selection, Insertion, Bubble sort (n^2)
- Calibrate your preparation
- Motivates guiding principles for algorithm analysis (worst-case and asymptotic analysis)
- Analysis generalizes to "Master Method"

1.3.2 The Sorting Problem

Input: an array of n numbers, unsorted, assume distinct

Output: same numbes, sorted in increasing order

Merge sort is recursive, it will spawn calls to itself to smaller arrays.

E.g. [8:30]

1.4 Merge Sort: Pseudocode

- Recursively sort 1st half of input array
- Recursively sort 2nd half of input array
- Merge two sorted sublists into one

[ignores base cases, odd number of elements, details of recursion implementation]

1.4.1 Pseudocode for Merge

C = output array [length=n] A = 1st sorted array [n/2] B = 2nd sorted array [n/2] i = counter of A elements j = counter of B elements i = j = 1

for k = 1 to n

if A(i) < B(j) C(k) = A(i) i++ else [B(j)<A(i)] C(k)=B(j) j++ end (ignores end cases)

1.4.2 Merge Sort Running Time

Key Question: running tome of Merge Sort on array of n numbers?

How many will get executed on a **single merge**?

Initialization step - 2 operations

for loop executes n times

3 operations per iteration: test and assignment, increment

Upshot: running time of Merge on array of m numbers is $\leq 4m + 2 \approx 6m\,(since\, m \geq 1)$ for the sake of simplicity.

On successive recursive calls, our input is smaller by half.

Claim: MergeSort requires

$$\leq 6n \log_2 n + 6n$$

operations to sort n numbers.

Logarithm review: # of times you divide by 2 until you get down to 1.

1.5 Merge Sort: Analysis

Claim: For every input array of n numbers, Merge Sort produces a sorted output array and uses at most $6n \log_2 n + 6n$ operations.

Proof of claim (assuming n = power of 2)

Recursion tree method [2:00]

The recursion tree has roughly $log_2 n$ levels, n = length of input array. The reason is that essentially the input size is being decreased by a factor of 2. $log_2 n + 1$ to be exact.

This allows us to count the work level by level.

At each level $j = 0, 1, 2, ..., \log_2 n$, there are 2^j subproblems, each of size $\frac{n}{2^j}$.

MergeSort calls itself twice, hence 2^j.

Proof of claim (assuming n = power of 2)

Total number of operations at level j:

[Each $j = 0, 1, 2, ..., \log_2 n$]

$\leq \left[2^j\right]_1 + \left[6\left[\left(\frac{n}{2^j}\right)\right]_2\right]_3 = [6n]_4$

1: all of level j subproblems

2: subproblem size at level j

3: work per level j subproblem

4: independent of j

The number of subproblems is doubling and the amount of work we do per sub problem is halving.

We care about the total work but if we have the amount of work the algorithm does per level, we just take the and multiply by number of levels

$\leq [6n]_1 [(\log_2 n + 1)]_2$

1: work per level

2: # of levels

1.6 Guiding Principles for Analysis of Algorithms

Guiding Principle #1

Worst case analysis:

- Our running time bout holds for every input of length n.
- particularly appropriate for "general-purpose" routines

Guiding Principle #2

As opposed to:

1. average-case analysis: analyze average running time of an algorithm under some assumption about relative frequency of the inputs.
2. bench marks: one agrees up front about some set, say ten or twenty, benchmark inputs, which are thought to represent practical or typical inputs for the algorithm.

Won't pay much attention to constant factors, lower order terms.

- Justifications:
  - Way easier mathematically
  - constants depend on architecture/compiler/programmer
  - lose very little predictive power

Guiding Principle #3

Asyptotic Analysis: Focus on running time for **large** input sizes.

E.g. $6n \log_2 n + 6n$ "better than" $\frac{1}{2}n^2$

Merge Sort better than Insertion Sort

Only big problems are interesting

To decide principle, you need to have domain knowledge of the problem.

Fast algorithm $\approx$ worst-case running time grows slowly with input size.

Holy grail: linear running time or close to it. Number of instructions grows proportional to the input size.