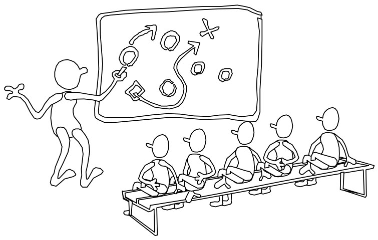
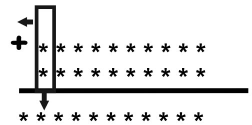
Great Theoretical Ideas In Computer Science		
Anupam Gupta Danny Sleator		CS 15-251 Fall 2010
Lecture 22	Nov 4, 2010	Carnegie Mellon University

Grade School Revisited:
How To Multiply Two Numbers



Time complexity of grade school addition

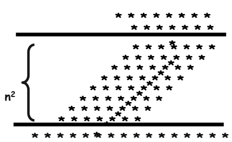


T(n) = amount of time grade school addition uses to add two n-bit numbers

T(n) is linear:

$$T(n) = c_1 n$$

Time complexity of grade school multiplication

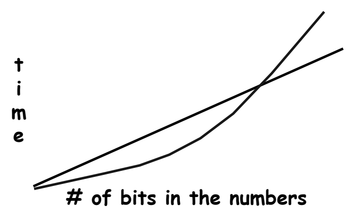


T(n) = The amount of time grade school multiplication uses to multiply two n-bit numbers

T(n) is quadratic:

$$T(n) = c_2 n^2$$

Grade School Addition: Linear time
Grade School Multiplication: Quadratic time



No matter how dramatic the difference in the constants, the quadratic curve will eventually dominate the linear curve

<tangent on asymptotic notation>

Our Goal

We want to define "time" in a way that transcends implementation details and allows us to make assertions about grade school addition in a very general yet useful way.

Roadblock ???

A given algorithm will take different amounts of time on the same inputs depending on such factors as:

- Processor speed
- Instruction set
- Disk speed
- Brand of compiler

On any reasonable computer, adding 3 bits and writing down the two bit answer can be done in constant time

Pick any particular computer M and define c to be the time it takes to perform on that computer.

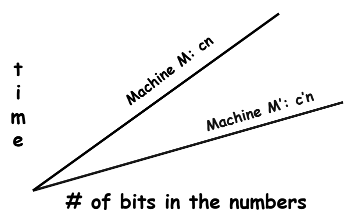
Total time to add two n-bit numbers using grade school addition:

cn [i.e., c time for each of n columns]

On another computer M', the time to perform may be c'.

Total time to add two n-bit numbers using grade school addition:

c'n [c' time for each of n columns]



The fact that we get a line is invariant under changes of implementations. Different machines result in different slopes, but the time taken grows linearly as input size increases.

Thus we arrive at an implementation-independent insight:

Grade School Addition is a linear time algorithm

This process of abstracting away details and determining the rate of resource usage in terms of the problem size n is one of the fundamental ideas in computer science.

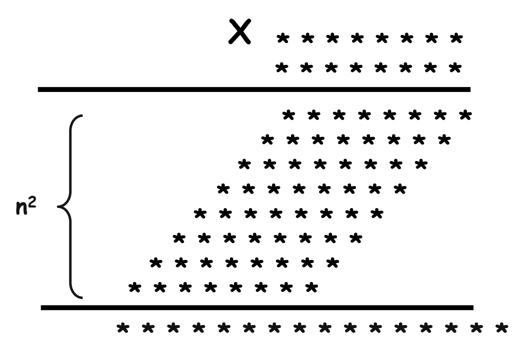
Time vs Input Size

For any algorithm, define
Input Size = # of bits to specify its inputs.

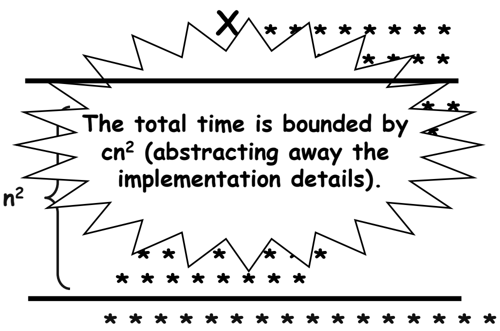
Define
$TIME_n$ = the worst-case amount of time used by the algorithm on inputs of size n

We often ask:
What is the growth rate of $Time_n$?

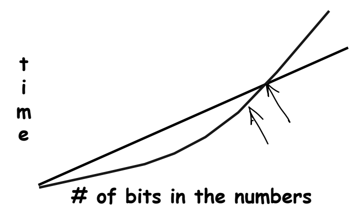
How to multiply 2 n-bit numbers.



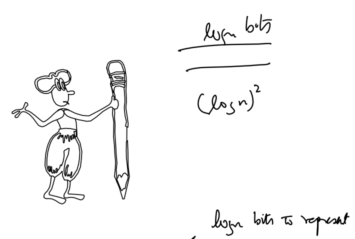
How to multiply 2 n-bit numbers.



Grade School Addition: Linear time
Grade School Multiplication: Quadratic time

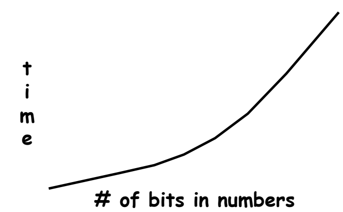


No matter how dramatic the difference in the constants, the quadratic curve will eventually dominate the linear curve



How much time does it take to square the number n using grade school multiplication?

Grade School Multiplication: Quadratic time



$c(\log n)^2$ time to square the number n
Input size is measured in bits, unless we say otherwise.

Worst Case Time

Worst Case Time T(n) for algorithm A:

$$T(n) = \text{Max}_{[\text{all permissible inputs X of size n}]} \text{Runtime}(A,X)$$

Runtime(A,X) = Running time of algorithm A on input X.

If T(n) is not polynomial, the algorithm is not efficient: the run time scales too poorly with the input size.

This will be the yardstick with which we will measure "efficiency".

Multiplication is efficient, what about "reverse multiplication"?

Let's define FACTORING(N) to be any method to produce a non-trivial factor of N, or to assert that N is prime.

Factoring The Number N By Trial Division

Trial division up to √N

for k = 2 to √N do
if k | N then
return "N has a non-trivial factor k"
return "N is prime"

c √N (logN)^2 time if division is c (logN)^2 time

Is this efficient?

No! The input length n = log N.
Hence we're using c $2^{n/2}$ n^2 time.

Can we do better?

We know of methods for FACTORING that are sub-exponential (about $2^{n^{1/3}}$ time) but nothing efficient.

$2^{n^{1/3}}$

Notation to Discuss Growth Rates

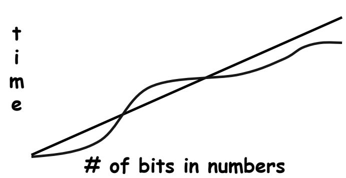
For any monotonic function f from the positive integers to the positive integers, we say

"f = O(n)" or "f is O(n)"

If some constant times n eventually dominates f

[Formally: there exists a constant c such that for all sufficiently large n: f(n) ≤ cn]

f = O(n) means that there is a line that can be drawn that stays above f from some point on



Other Useful Notation: Ω

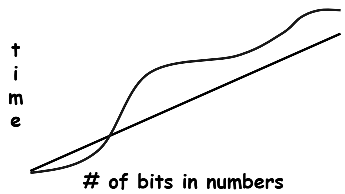
For any monotonic function f from the positive integers to the positive integers, we say

"f = Ω(n)" or "f is Ω(n)"

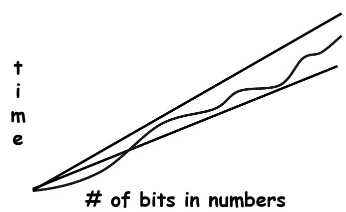
If f eventually dominates some constant times n

[Formally: there exists a constant c such that for all sufficiently large n: f(n) ≥ cn]

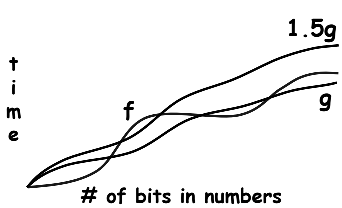
f = Ω(n) means that there is a line that can be drawn that stays below f from some point on



f = Θ(n) means that f can be sandwiched between two lines from some point on.



f = O(g) means that there is some constant c such that c g(n) stays above f(n) from some point on.



Yet More Useful Notation: Θ

For any monotonic function f from the positive integers to the positive integers, we say

"f = Θ(n)" or "f is Θ(n)"

if: f = O(n) and f = Ω(n)

Notation to Discuss Growth Rates

For any two monotonic functions f and g from the positive integers to the positive integers, we say

"f = O(g)" or "f is O(g)"

If some constant times g eventually dominates f

[Formally: there exists a constant c such that for all sufficiently large n: $f(n) \le c\, g(n)$]

Other Useful Notation: Ω

For any two monotonic functions f and g from the positive integers to the positive integers, we say

"f = Ω(g)" or "f is Ω(g)"

If f eventually dominates some constant times g

[Formally: there exists a constant c such that for all sufficiently large n: $f(n) \ge c\, g(n)$]

Yet More Useful Notation: Θ

For any two monotonic functions f and g from the positive integers to the positive integers, we say

"$f = \Theta(g)$" or "f is $\Theta(g)$"

If: $f = O(g)$ and $f = \Omega(g)$

</tangent on asymptotic notation>

Can we even break the quadratic time barrier?

In other words, can we do something very different than grade school multiplication?

Divide And Conquer

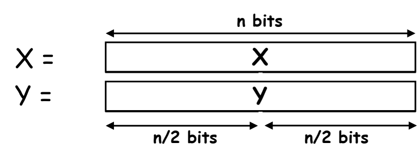
An approach to faster algorithms:

DIVIDE a problem into smaller subproblems

CONQUER them recursively

GLUE the answers together so as to obtain the answer to the larger problem

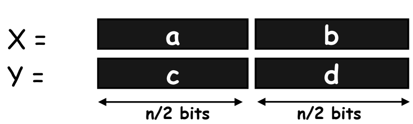
Multiplication of 2 n-bit numbers



$$X = a\ 2^{n/2} + b \qquad Y = c\ 2^{n/2} + d$$

$$X \times Y = ac\ 2^n + (ad + bc)\ 2^{n/2} + bd$$

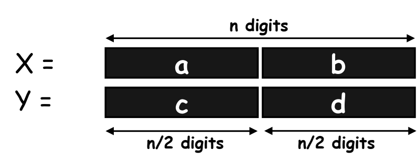
Multiplication of 2 n-bit numbers



$$X \times Y = ac\ 2^n + (ad + bc)\ 2^{n/2} + bd$$

```
MULT(X,Y):
If |X| = |Y| = 1 then return XY
else    break X into a;b and Y into c;d
  return MULT(a,c) 2^n + (MULT(a,d)
      + MULT(b,c)) 2^{n/2} + MULT(b,d)
```

Same thing for numbers in decimal



$$X = a\ 10^{n/2} + b \qquad Y = c\ 10^{n/2} + d$$

$$X \times Y = ac\ 10^n + (ad + bc)\ 10^{n/2} + bd$$

Multiplying (Divide & Conquer style)

12345678 * 21394276

1234*2139   1234*4276   5678*2139   5678*4276

12*21   12*39   34*21   34*39

1*2   1*1   2*2   2*1

2   1   4   2

Hence: 12*21 = $2*10^2 + (1 + 4)10^1 + 2 = 252$

a | b
c | d

Multiplying (Divide & Conquer style)

12345678 * 21394276

1234*2139   1234*4276   5678*2139   5678*4276

252 $*10^4$ + 468 $*10^2$ + 714 $*10^2$ + 1326 $*1$ = 2639526

a | b
c | d

Multiplying (Divide & Conquer style)

12345678 * 21394276

2639526 $*10^8$ + 5276584 $*10^4$ + 12145242 $*10^4$ + 24279128

= 264126842539128

a | b
c | d

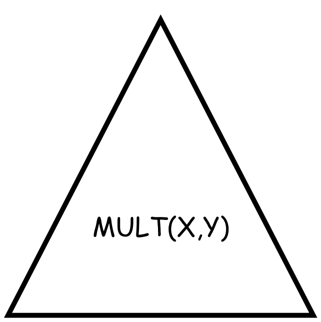
Multiplying (Divide & Conquer style)

12345678 * 21394276

= 264126842539128

a | b
c | d

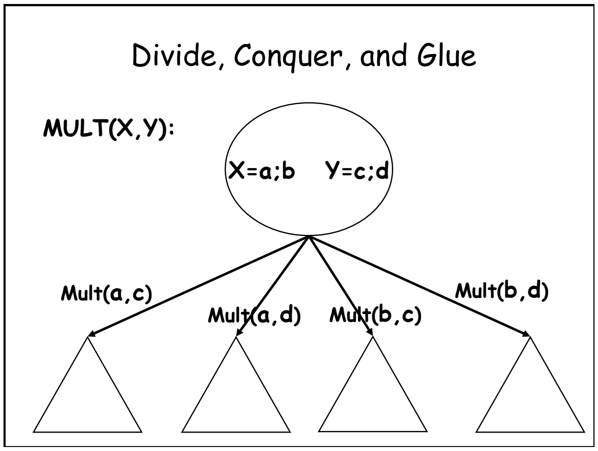
Divide, Conquer, and Glue

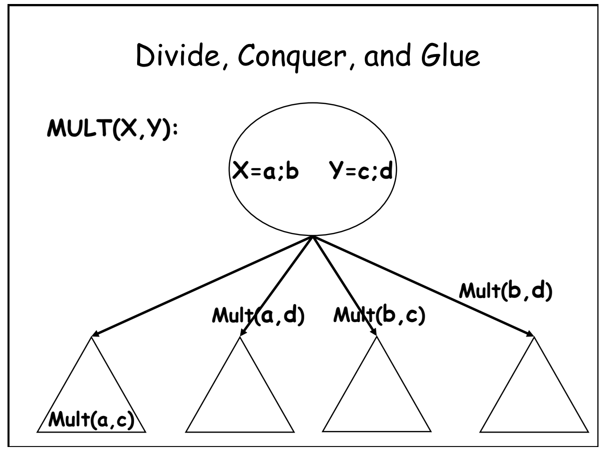


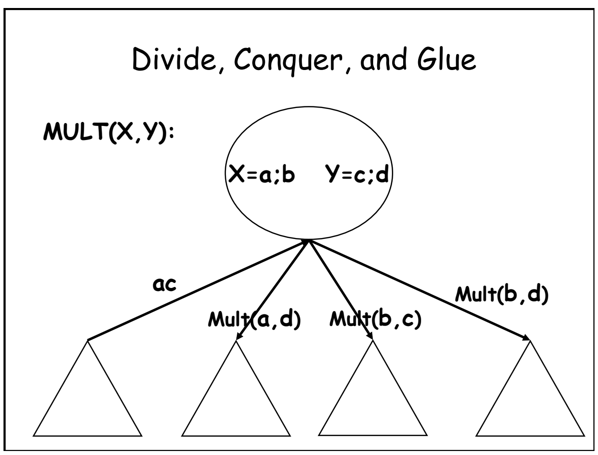
Divide, Conquer, and Glue

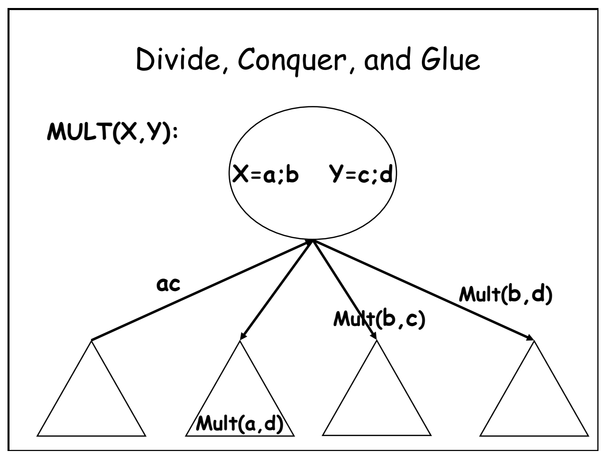
MULT(X,Y):

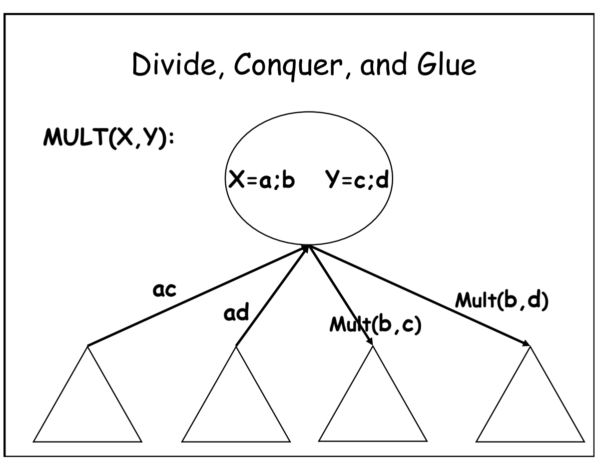
**if |X| = |Y| = 1
then return XY,
else…**

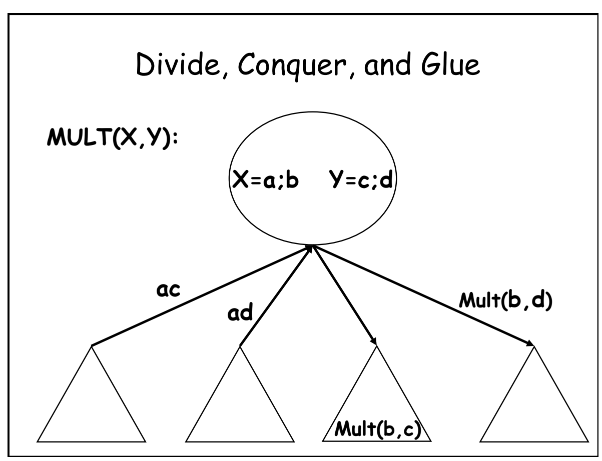

Divide, Conquer, and Glue
MULT(X,Y):
X=a;b Y=c;d
Mult(a,c)
Mult(a,d)
Mult(b,c)
Mult(b,d)


Divide, Conquer, and Glue
MULT(X,Y):
X=a;b Y=c;d
Mult(a,d)
Mult(b,c)
Mult(b,d)
Mult(a,c)


Divide, Conquer, and Glue
MULT(X,Y):
X=a;b Y=c;d
ac
Mult(a,d)
Mult(b,c)
Mult(b,d)

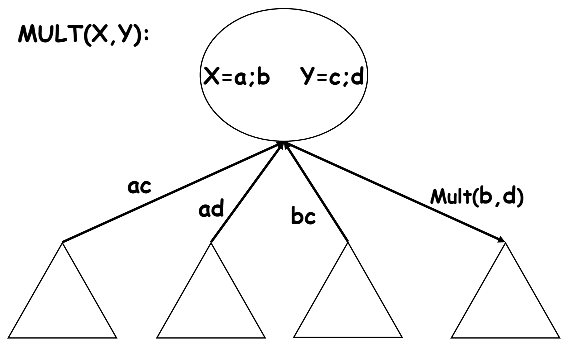

Divide, Conquer, and Glue
MULT(X,Y):
X=a;b Y=c;d
ac
Mult(b,c)
Mult(b,d)
Mult(a,d)


Divide, Conquer, and Glue
MULT(X,Y):
X=a;b Y=c;d
ac
ad
Mult(b,c)
Mult(b,d)


Divide, Conquer, and Glue
MULT(X,Y):
X=a;b Y=c;d
ac
ad
Mult(b,d)
Mult(b,c)

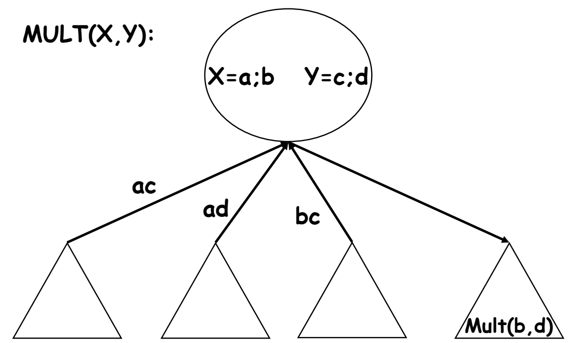
Divide, Conquer, and Glue

MULT(X,Y):



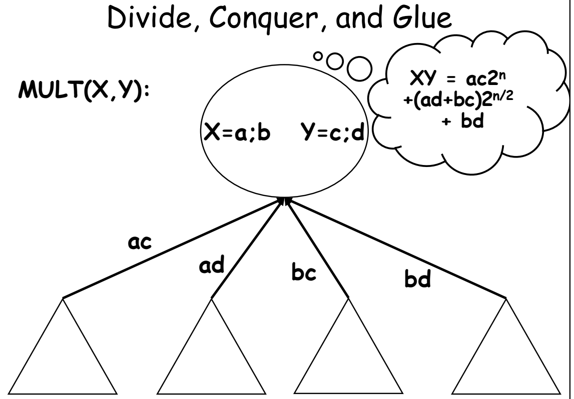
Divide, Conquer, and Glue

MULT(X,Y):



Divide, Conquer, and Glue

MULT(X,Y):



Time required by MULT

T(n) = time taken by MULT on two n-bit numbers

What is T(n)? What is its growth rate?

Big Question: Is it $\Theta(n^2)$?

$T(n) = 4\ T(n/2) + O(n)$

conquering time → $4\ T(n/2)$

divide and glue → $O(n)$

Recurrence Relation

$T(1) = 1$

$T(n) = 4\ T(n/2) + O(n)$

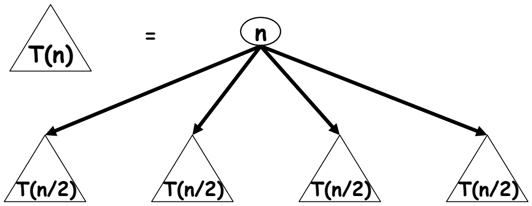
Simplified Recurrence Relation

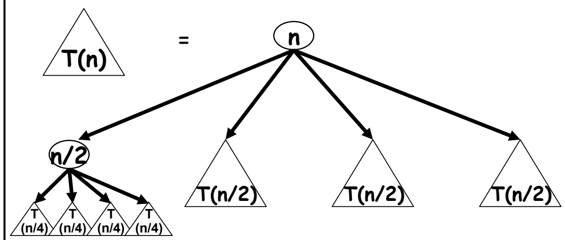
$T(1) = 1$

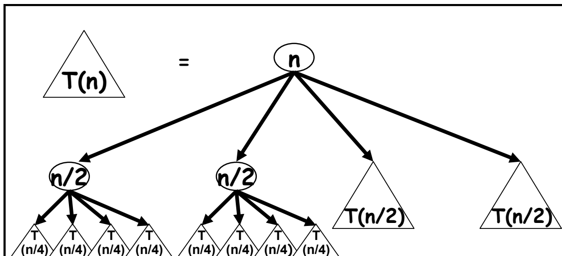
$T(n) = 4\ T(n/2) + n$

conquering time → $4\ T(n/2)$

divide and glue → n







0	n
1	n/2 + n/2 + n/2 + n/2
2	
i	**Level i is the sum of 4^i copies of $n/2^i$**
	. . . .
	1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1

1n =	n
2n =	n/2 + n/2 + n/2 + n/2
4n =	
$2^i n$ =	Level i is the sum of 4^i copies of $n/2^i$
	. . . .
(n)n =	1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1

$n(1+2+4+8+ \; . \; . \; . \; +n) = n(2n-1) = 2n^2-n$

Divide and Conquer MULT: $\Theta(n^2)$ time
Grade School Multiplication: $\Theta(n^2)$ time

Bummer!

MULT calls itself 4 times. Can you see a way to reduce the number of calls?

Gauss' Complex Puzzle

Remember how to multiply two complex numbers a + bi and c + di?

(a+bi)(c+di) = [ac – bd] + [ad + bc] i

Input: a,b,c,d
Output: ac-bd, ad+bc

If multiplying two real numbers costs $1 and adding them costs a penny, what is the cheapest way to obtain the output from the input?

Can you do better than $4.03?

Gauss' $3.05 Method

Input: a,b,c,d
Output: ac-bd, ad+bc

¢ $X_1 = a + b$
¢ $X_2 = c + d$
$ $X_3 = X_1 X_2$ $= ac + ad + bc + bd$
$ $X_4 = ac$
$ $X_5 = bd$
¢ $X_6 = X_4 – X_5$ $= ac - bd$
¢¢ $X_7 = X_3 – X_4 – X_5$ $= bc + ad$

The Gauss optimization saves one multiplication out of four.

It requires 25% less work.

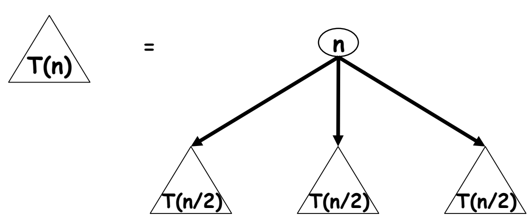
Karatsuba, Anatolii Alexeevich (1937-2008)

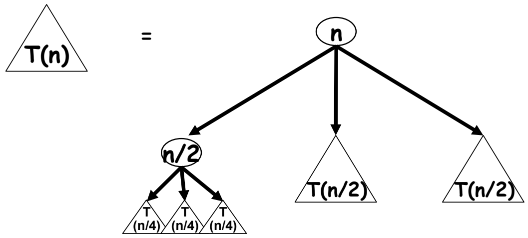
In 1962 Karatsuba had formulated the first mult. algorithm to break the n^2 barrier!

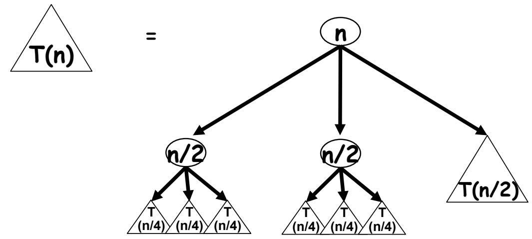
Gaussified MULT (Karatsuba 1962)

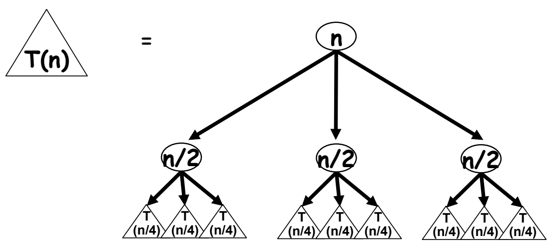
```
MULT(X,Y):
If |X| = |Y| = 1 then return XY
else    break X into a;b and Y into c;d
        e : = MULT(a,c)
        f  := MULT(b,d)
   return
  e 2^n + (MULT(a+b,c+d) – e – f) 2^(n/2) + f
```

$T(n) = 3\ T(n/2) + n$









0	n
1	n/2 + n/2 + n/2
2	
i	**Level i is the sum of 3^i copies of $n/2^i$**
	. . . .
$\log_2(n)$	1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1

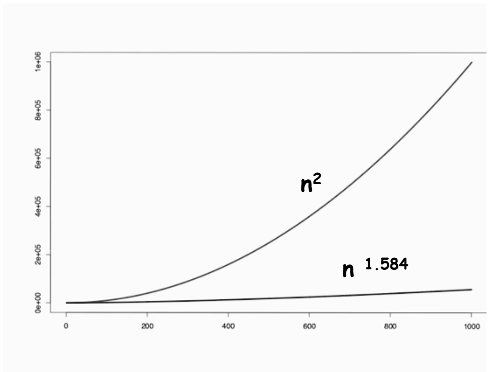
1n =	n
3/2n =	n/2 + n/2 + n/2
9/4n =	
$(3/2)^i n$ =	**Level i is the sum of 3^i copies of $n/2^i$**
	. . . .
$(3/2)^{\log n} n$ =	1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1

$n(1+3/2+(3/2)^2+ \ . \ . \ . \ + (3/2)^{\log_2 n}) = 3n^{1.58\ldots} - 2n$

Dramatic Improvement for Large n

$$T(n) = 3n^{\log_2 3} - 2n$$
$$= \Theta(n^{\log_2 3})$$
$$= \Theta(n^{1.58\ldots})$$

A huge savings over $\Theta(n^2)$ when n gets large.



3-Way Multiplication

The key idea of the algorithm is to divide a large integer into 3 parts (rather than 2) of size approximately n/3 and then multiply those parts.

$154517766 = 154 * 10^6 + 517 * 10^3 + 766$

3-Way Multiplication

Let

$X = x_2\ 10^{2p} + x_1\ 10^p + x_0$

$Y = y_2\ 10^{2p} + y_1\ 10^p + y_0$

Then

$X*Y = 10^{4p}\ x_2y_2 + 10^{3p}\ (x_2y_1+x_1y_2) + 10^{2p}\ (x_2y_0+x_1y_1+x_0y_2) + 10^p\ (x_1y_0+x_0y_1) + x_0y_0$

$T(n) = 9\ T(n/3) + \Theta(n)$

$T(n) = \Theta(n^2)$

3-Way Multiplication

Consider the equation in general form p > 3

$T(n) = p\ T(n/3) + O(n)$

Its solution is given by

$T(n) = O(n^{\log_3 p})$

Thus, this is faster if p = 5 or less

$T(n) = O(n^{\log_3 5}) = O(n^{1.46\ldots})$

Is it possible to reduce the number of multiplications to 5?

Here is the system of new variables:

$(x_0\ y_0)$	$= Z_0$
$12\ (x_1y_0+x_0y_1)$	$= 8\ Z_1 - Z_2 - 8\ Z_3 + Z_4$
$24\ (x_2y_0+x_1y_1+x_0y_2)$	$= -30\ Z_0 + 16\ Z_1 - Z_2 + 16\ Z_3 - Z_4$
$12\ (x_2y_1+x_1y_2)$	$= -2\ Z_1 + Z_2 + 2\ Z_3 - Z_4$
$24\ (x_2y_2)$	$= 6\ Z_0 - 4\ Z_1 + Z_2 - 4\ Z_3 + Z_4$

5 Multiplications Suffice

Here are the values of Z_k which make this work:

$Z_0 = x_0\ y_0$

$Z_1 = (x_0+x_1+x_2)\ (y_0+y_1+y_2)$

$Z_2 = (x_0+2\ x_1+4\ x_2)\ (y_0+2\ y_1+4\ y_2)$

$Z_3 = (x_0-x_1+x_2)\ (y_0-y_1+y_2)$

$Z_4 = (x_0-2\ x_1+4\ x_2)\ (y_0-2\ y_1+4\ y_2)$

We leave checking this to the reader. Note that multiplying and dividing by small constants (eg:2,4,12,24) are $O(n)$ time and absorbed by the constant term in the recurrence.

Further Generalizations

It is possible to develop a faster algorithm by increasing the number of splits.

A 4-way splitting:

$$T(n) = 7\,T(n/4) + O(n)$$

$$T(n) = O(n^{1.403\ldots})$$

Further Generalizations

In similar fashion, the k-way split requires 2k-1 multiplications. (We do not show that here. See http://en.wikipedia.org/wiki/Toom-Cook_multiplication)

A k-way splitting:

$$T(n) = (2k-1)\,T(n/k) + O(n)$$

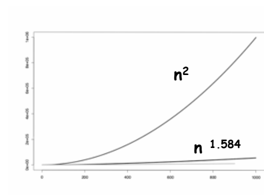
$$T(n) = O(n^{\log_k (2k-1)})$$

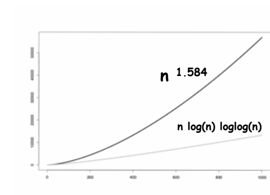
$n^{1.58}$, $n^{1.46}$, $n^{1.40}$, $n^{1.36}$, $n^{1.33}$, …

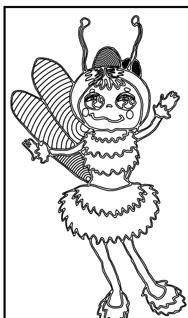
Note, we will never get a linear performance

Multiplication Algorithms

Grade School	$O(n^2)$
Karatsuba	$O(n^{1.58\ldots})$
3-way split	$O(n^{1.46\ldots})$
K-way split	$O(n^{\log_k (2k-1)})$
Fast Fourier Transform	O(n logn loglogn)







Study Bee

- Asymptotic notation
- Divide and Conquer
- Karatsuba Multiplication
- Solving Recurrences