CS 473: Algorithms

Chandra Chekuri
chekuri@cs.illinois.edu
3228 Siebel Center

University of Illinois, Urbana-Champaign

Fall 2009

Part I

Divide and Conquer

Divide and Conquer Paradigm

Divide and Conquer is a common and useful type of recursion

Approach

Divide and Conquer Paradigm

Divide and Conquer is a common and useful type of recursion

Approach

- Break problem instance into smaller instances - divide step

Divide and Conquer Paradigm

Divide and Conquer is a common and useful type of recursion

Approach

- Break problem instance into smaller instances - divide step
- Recursively solve problem on smaller instances

Divide and Conquer Paradigm

Divide and Conquer is a common and useful type of recursion

Approach

- Break problem instance into smaller instances - divide step
- Recursively solve problem on smaller instances
- Combine solutions to smaller isntances to obtain a solution to the original instance - conquer step

Divide and Conquer Paradigm

Divide and Conquer is a common and useful type of recursion

Approach

- Break problem instance into smaller instances - divide step
- Recursively solve problem on smaller instances
- Combine solutions to smaller isntances to obtain a solution to the original instance - conquer step

Divide and Conquer Paradigm

Divide and Conquer is a common and useful type of recursion

Approach

- Break problem instance into smaller instances - divide step
- Recursively solve problem on smaller instances
- Combine solutions to smaller isntances to obtain a solution to the original instance - conquer step

Question: Why is this not plain recursion?

Divide and Conquer Paradigm

Divide and Conquer is a common and useful type of recursion

Approach

- Break problem instance into smaller instances - divide step
- Recursively solve problem on smaller instances
- Combine solutions to smaller isntances to obtain a solution to the original instance - conquer step

Question: Why is this not plain recursion?

- In divide and conquer, each smaller instance is typically at least a constant factor smaller than the original instance which leads to efficienct running times

Divide and Conquer Paradigm

Divide and Conquer is a common and useful type of recursion

Approach

- Break problem instance into smaller instances - divide step
- Recursively solve problem on smaller instances
- Combine solutions to smaller isntances to obtain a solution to the original instance - conquer step

Question: Why is this not plain recursion?

- In divide and conquer, each smaller instance is typically at least a constant factor smaller than the original instance which leads to efficienct running times
- There are many examples of this particular type of recursion that it deserves its own treatment

Sorting

Input Given an array of n elements

Goal Rearrange them in ascending order

Merge Sort [von Neumann]

1. Input: Array $A[1 \ldots n]$

$$A\ L\ G\ O\ R\ I\ T\ H\ M\ S$$

Merge Sort [von Neumann]

1. Input: Array $A[1 \ldots n]$

   $A\ L\ G\ O\ R\ I\ T\ H\ M\ S$

2. Divide into subarrays $A[1 \ldots m]$ and $A[m+1 \ldots n]$, where $m = \lfloor n/2 \rfloor$

   $A\ L\ G\ O\ R \qquad I\ T\ H\ M\ S$

Merge Sort [von Neumann]

1. Input: Array $A[1 \ldots n]$

   $$A\ L\ G\ O\ R\ I\ T\ H\ M\ S$$

2. Divide into subarrays $A[1 \ldots m]$ and $A[m+1 \ldots n]$, where $m = \lfloor n/2 \rfloor$

   $$A\ L\ G\ O\ R \qquad I\ T\ H\ M\ S$$

3. Recursively mergesort $A[1 \ldots m]$ and $A[m+1 \ldots n]$

   $$A\ G\ L\ O\ R \qquad H\ I\ M\ S\ T$$

Merge Sort [von Neumann]

1. Input: Array $A[1 \ldots n]$

   $$A\ L\ G\ O\ R\ I\ T\ H\ M\ S$$

2. Divide into subarrays $A[1 \ldots m]$ and $A[m+1 \ldots n]$, where $m = \lfloor n/2 \rfloor$

   $$A\ L\ G\ O\ R \qquad I\ T\ H\ M\ S$$

3. Recursively mergesort $A[1 \ldots m]$ and $A[m+1 \ldots n]$

   $$A\ G\ L\ O\ R \qquad H\ I\ M\ S\ T$$

4. Merge the sorted arrays

   $$A\ G\ H\ I\ L\ M\ O\ R\ S\ T$$

Merge Sort [von Neumann]

1. Input: Array $A[1 \ldots n]$

   $A\ L\ G\ O\ R\ I\ T\ H\ M\ S$

2. Divide into subarrays $A[1 \ldots m]$ and $A[m+1 \ldots n]$, where $m = \lfloor n/2 \rfloor$

   $A\ L\ G\ O\ R \qquad I\ T\ H\ M\ S$

3. Recursively mergesort $A[1 \ldots m]$ and $A[m+1 \ldots n]$

   $A\ G\ L\ O\ R \qquad H\ I\ M\ S\ T$

4. Merge the sorted arrays

   $A\ G\ H\ I\ L\ M\ O\ R\ S\ T$

Merging Sorted Arrays

- Use a new array C to store the merged array
- Scan A and B from left-to-right, storing elements in C in order

$$A\ G\ L\ O\ R \qquad H\ I\ M\ S\ T$$
$$A$$

Merging Sorted Arrays

- Use a new array C to store the merged array
- Scan A and B from left-to-right, storing elements in C in order

$$A\ G\ L\ O\ R \qquad\qquad H\ I\ M\ S\ T$$
$$A\ G$$

Merging Sorted Arrays

- Use a new array C to store the merged array
- Scan A and B from left-to-right, storing elements in C in order

$$A\ G\ L\ O\ R \qquad H\ I\ M\ S\ T$$
$$A\ G\ H$$

Merging Sorted Arrays

- Use a new array C to store the merged array
- Scan A and B from left-to-right, storing elements in C in order

$$A\ G\ L\ O\ R \qquad H\ I\ M\ S\ T$$
$$A\ G\ H\ I$$

Merging Sorted Arrays

- Use a new array C to store the merged array
- Scan A and B from left-to-right, storing elements in C in order

$$A\ G\ L\ O\ R \qquad H\ I\ M\ S\ T$$
$$A\ G\ H\ I\ L\ M\ O\ R\ S\ T$$

Merging Sorted Arrays

- Use a new array C to store the merged array
- Scan A and B from left-to-right, storing elements in C in order

$$A\ G\ L\ O\ R \qquad H\ I\ M\ S\ T$$
$$A\ G\ H\ I\ L\ M\ O\ R\ S\ T$$

- Merge two arrays using only constantly more extra space (exercise).

Running Time

$T(n)$: time for merge sort to sort an n element array

Running Time

$T(n)$: time for merge sort to sort an n element array

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn$$

Running Time

$T(n)$: time for merge sort to sort an n element array

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn$$

What do we want as a solution to the recurrence?

Almost always only an *asymptotically* tight bound. That is we want to know $f(n)$ such that $T(n) = \Theta(f(n))$.

- $T(n) = O(f(n))$ - upper bound
- $T(n) = \Omega(f(n))$ - lower bound

Solving Recurrences: Some Techniques

- Know some basic math: geometric series, logarithms, exponentials, elementary calculus
- Expand the recurrence and spot a pattern and use simple math
- Recursion tree method — imagine the computation as a tree
- Guess and verify — useful for proving upper and lower bounds even if not tight bounds

Solving Recurrences: Some Techniques

- Know some basic math: geometric series, logarithms, exponentials, elementary calculus
- Expand the recurrence and spot a pattern and use simple math
- Recursion tree method — imagine the computation as a tree
- Guess and verify — useful for proving upper and lower bounds even if not tight bounds

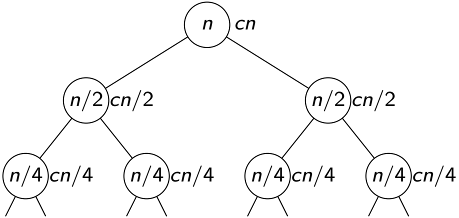
Albert Einstein: "Everything should be made as simple as possible, but not simpler."

Know where to be loose in analysis and where to be tight. Comes with practice, practice, practice!

Recursion Trees

MergeSort: n is a power of 2

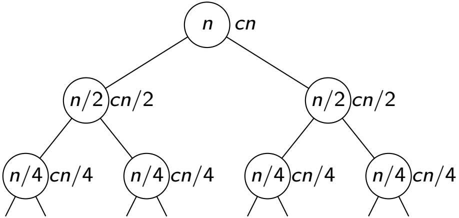
1. Unroll the recurrence. $T(n) = 2T(n/2) + cn$



Recursion Trees

MergeSort: n is a power of 2

1. Unroll the recurrence. $T(n) = 2T(n/2) + cn$

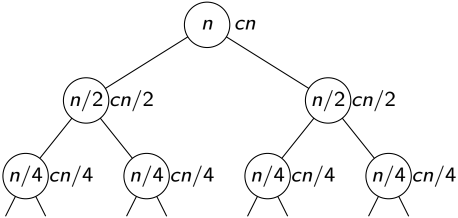


2. Identify a pattern.

Recursion Trees

MergeSort: n is a power of 2

1. Unroll the recurrence. $T(n) = 2T(n/2) + cn$

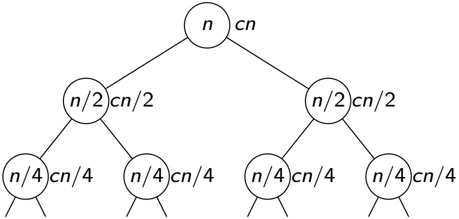


2. Identify a pattern. At the ith level total work is cn

Recursion Trees

MergeSort: n is a power of 2

1. Unroll the recurrence. $T(n) = 2T(n/2) + cn$

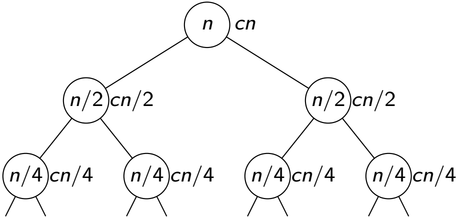


2. Identify a pattern. At the ith level total work is cn
3. Sum over all levels.

Recursion Trees

MergeSort: n is a power of 2

1. Unroll the recurrence. $T(n) = 2T(n/2) + cn$



2. Identify a pattern. At the ith level total work is cn
3. Sum over all levels. The number of levels is $\log n$. So total is $cn \log n = O(n \log n)$

Mergesort Analysis

When n is not a power of 2

- When n is not a power of 2, the running time of mergesort is expressed as

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn$$

Mergesort Analysis

When n is not a power of 2

- When n is not a power of 2, the running time of mergesort is expressed as

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn$$

- $n_1 = 2^{k-1} < n \leq 2^k = n_2$ (n_1, n_2 powers of 2)

Mergesort Analysis

When n is not a power of 2

- When n is not a power of 2, the running time of mergesort is expressed as

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn$$

- $n_1 = 2^{k-1} < n \leq 2^k = n_2$ (n_1, n_2 powers of 2)
- $T(n_1) < T(n) \leq T(n_2)$ (Why?)

Mergesort Analysis

When n is not a power of 2

- When n is not a power of 2, the running time of mergesort is expressed as

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn$$

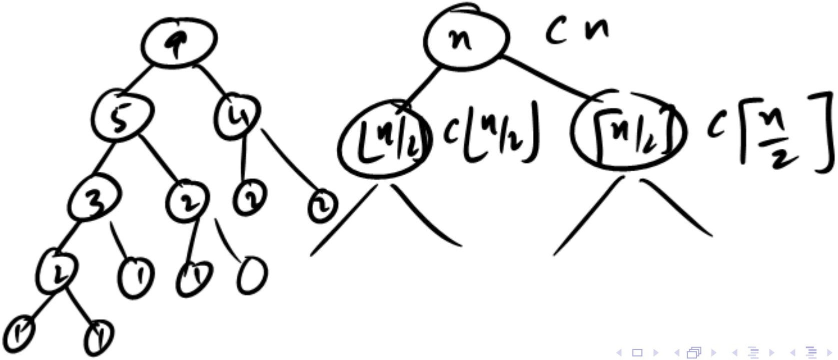
- $n_1 = 2^{k-1} < n \leq 2^k = n_2$ (n_1, n_2 powers of 2)
- $T(n_1) < T(n) \leq T(n_2)$ (Why?)
- $T(n) = \Theta(n \log n)$ since $n/2 \leq n_1 < n \leq n_2 \leq 2n$.

Recursion Trees

MergeSort: n is not a power of 2

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn$$

Observation: For any number x, $\lfloor x/2 \rfloor + \lceil x/2 \rceil = x$.



$\lceil n/2 \rceil \leq \frac{2}{3} n \quad \forall n \geq 2$

$\left(\frac{2}{3}\right)^i \cdot n \approx 2$

$i \approx \log_{3/2} n \approx \frac{\log_2 n}{\log_2 3/2}$

$\approx 2 \log_2 n$

When n is not a power of 2: Guess and Verify

If n is power of 2 we saw that $T(n) = \Theta(n \log n)$.
Can *guess* that $T(n) = \Theta(n \log n)$ for all n.
Verify?

When n is not a power of 2: Guess and Verify

If n is power of 2 we saw that $T(n) = \Theta(n \log n)$.
Can *guess* that $T(n) = \Theta(n \log n)$ for all n.
Verify? proof by induction!

Induction Hypothesis: $T(n) \leq 2cn \log n$ for all $n \geq 1$
Base Case: $n = 1$. $T(1) = 0$ since no need to do any work and $2cn \log n = 0$ for $n = 1$.
Induction Step Assume $T(k) \leq 2ck \log k$ for all $k < n$ and prove it for $k = n$.

Induction Step

We have

$$\begin{aligned}
T(n) &= T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn \\
&\leq 2c\lfloor n/2 \rfloor \log \lfloor n/2 \rfloor + 2c\lceil n/2 \rceil \log \lceil n/2 \rceil + cn \quad \text{(by induction)} \\
&\leq 2c\lfloor n/2 \rfloor \log \lceil n/2 \rceil + 2c\lceil n/2 \rceil \log \lceil n/2 \rceil + cn \\
&\leq 2c(\lfloor n/2 \rfloor + \lceil n/2 \rceil) \log \lceil n/2 \rceil + cn \\
&\leq 2cn \log \lceil n/2 \rceil + cn \\
&\leq 2cn \log(2n/3) + cn \quad (\text{since } \lceil n/2 \rceil \leq 2n/3 \text{ for all } n \geq 2) \\
&\leq 2cn \log n + cn(1 - 2\log 3/2) \\
&\leq 2cn \log n + cn(\log 2 - \log 9/4) \\
&\leq 2cn \log n
\end{aligned}$$

Guess and Verify

The math worked out like magic!
Why was $2cn \log n$ chosen instead of say $4cn \log n$?

Guess and Verify

The math worked out like magic!
Why was $2cn \log n$ chosen instead of say $4cn \log n$?

Typically we don't know upfront what constant to choose. Instead we assume that $T(n) \leq \alpha cn \log n$ for some constant α that will be fixed later. All we need to prove that there is some sufficiently large constant α that will make the algebra go through.

We need to choose α such that $\alpha \log 3/2 > 1$.

Typically you do the algebra with α and then show at the end that α can be chosen to be sufficiently large constant.

Guess and Verify: When is a guess incorrect?

Suppose we guessed that the soln to the mergesort recurrent is $T(n) = O(n)$. We try to prove by induction that $T(n) \leq \alpha cn$ for some constant α.

Induction Step: attempt

$$\begin{aligned} T(n) &= T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn \\ &\leq \alpha c \lfloor n/2 \rfloor + \alpha c \lceil n/2 \rceil + cn \\ &\leq \alpha cn + cn \\ &\leq (\alpha + 1)cn \end{aligned}$$

But we want to show that $T(n) \leq \alpha cn$! So guess does not work for *any* constant α. Suggests that our guess is incorrect.

Selection Sort vs Merge Sort

- Selection Sort spends $O(n)$ work to reduce problem from n to $n-1$ leading to $O(n^2)$ running time.
- Merge Sort spends $O(n)$ time *after* reducing problem to two instances of size $n/2$ each. Running time is $O(n \log n)$

Selection Sort vs Merge Sort

- Selection Sort spends $O(n)$ work to reduce problem from n to $n-1$ leading to $O(n^2)$ running time.
- Merge Sort spends $O(n)$ time *after* reducing problem to two instances of size $n/2$ each. Running time is $O(n \log n)$

Question: Merge Sort splits into 2 (roughly) equal sized arrays. Can we do better by splitting into more than 2 arrays? Say k arrays of size n/k each?

Quick Sort

Quick Sort[Hoare]

1. Pick a pivot element from array
2. Split array into 3 subarrays: those smaller than pivot, those larger than pivot, and the pivot itself.
3. Recursively sort the subarrays, and concatenate them.

Quick Sort

Quick Sort[Hoare]

1. Pick a pivot element from array
2. Split array into 3 subarrays: those smaller than pivot, those larger than pivot, and the pivot itself.
3. Recursively sort the subarrays, and concatenate them.

Example:

- array: 16, 12, 14, 20, 5, 3, 18, 19, 1
- pivot: 16
- split into 12, 14, 5, 3, 1 and 20, 19, 18 and recursively sort
- put them together with pivot in middle

Quick Sort

Quick Sort[Hoare]

1. Pick a pivot element from array
2. Split array into 3 subarrays: those smaller than pivot, those larger than pivot, and the pivot itself. Linear scan of array does it. Time is $O(n)$
3. Recursively sort the subarrays, and concatenate them.

Example:

- array: 16, 12, 14, 20, 5, 3, 18, 19, 1
- pivot: 16
- split into 12, 14, 5, 3, 1 and 20, 19, 18 and recursively sort
- put them together with pivot in middle

Quick Sort

Quick Sort[Hoare]

1. Pick a pivot element from array
2. Split array into 3 subarrays: those smaller than pivot, those larger than pivot, and the pivot itself. Linear scan of array does it. Time is $O(n)$
3. Recursively sort the subarrays, and concatenate them.

Example:

- array: 16, 12, 14, 20, 5, 3, 18, 19, 1
- pivot: 16
- split into 12, 14, 5, 3, 1 and 20, 19, 18 and recursively sort
- put them together with pivot in middle

Time Analysis

- Let k be the rank of the chosen pivot. Then, $T(n) = T(k-1) + T(n-k) + O(n)$

Time Analysis

- Let k be the rank of the chosen pivot. Then,
  $T(n) = T(k-1) + T(n-k) + O(n)$
- If $k = \lceil n/2 \rceil$ then
  $T(n) = T(\lceil n/2 \rceil - 1) + T(\lfloor n/2 \rfloor) + O(n) \leq 2T(n/2) + O(n)$.
  Then, $T(n) = O(n \log n)$.

Time Analysis

- Let k be the rank of the chosen pivot. Then,
  $T(n) = T(k-1) + T(n-k) + O(n)$
- If $k = \lceil n/2 \rceil$ then
  $T(n) = T(\lceil n/2 \rceil - 1) + T(\lfloor n/2 \rfloor) + O(n) \leq 2T(n/2) + O(n)$.
  Then, $T(n) = O(n \log n)$.
    - Theoretically, median can be found in linear time.

Time Analysis

- Let k be the rank of the chosen pivot. Then, $T(n) = T(k-1) + T(n-k) + O(n)$
- If $k = \lceil n/2 \rceil$ then $T(n) = T(\lceil n/2 \rceil - 1) + T(\lfloor n/2 \rfloor) + O(n) \leq 2T(n/2) + O(n)$. Then, $T(n) = O(n \log n)$.
  - Theoretically, median can be found in linear time.
- Typically, pivot is the first or last element of array. Then,

$$T(n) = \max_{1 \leq k \leq n} (T(k-1) + T(n-k) + O(n))$$

In the worst case $T(n) = T(n-1) + O(n)$, which means $T(n) = O(n^2)$. Happens if array is already sorted and pivot is always first element.

Part II

Fast Multiplication

Multiplying Numbers

Problem Given two n-digit numbers x and y, compute their product.

Grade School Multiplication

Compute "partial product" by multiplying each digit of y with x and adding the partial products.

$$
\begin{array}{r}
3141 \\
\times 2718 \\
\hline
25128 \\
3141 \\
21987 \\
6282 \\
\hline
8537238
\end{array}
$$

Time Analysis of Grade School Multiplication

- Each partial product: $\Theta(n)$
- Number of partial products: $\Theta(n)$
- Addition of partial products: $\Theta(n^2)$
- Total time: $\Theta(n^2)$

A Trick of Gauss

Carl Fridrich Gauss: 1777–1855 "Prince of Mathematicians"

Observation: Multiply two complex numbers: $(a + bi)$ and $(c + di)$

$$(a + bi)(c + di) = ac - bd + (ad + bc)i$$

A Trick of Gauss

Carl Fridrich Gauss: 1777–1855 "Prince of Mathematicians"

Observation: Multiply two complex numbers: $(a + bi)$ and $(c + di)$

$$(a + bi)(c + di) = ac - bd + (ad + bc)i$$

How many multiplications do we need?

A Trick of Gauss

Carl Fridrich Gauss: 1777–1855 "Prince of Mathematicians"

Observation: Multiply two complex numbers: $(a + bi)$ and $(c + di)$

$$(a + bi)(c + di) = ac - bd + (ad + bc)i$$

How many multiplications do we need?

Only 3! If we do extra additions and subtractions.
Compute $ac, bd, (a + b)(c + d)$. Then
$(ad + bc) = (a + b)(c + d) - ac - bd$

Divide and Conquer

Assume n is a power of 2 for simplicity and numbers are in decimal.

- $x = x_{n-1}x_{n-2}\ldots x_0$ and $y = y_{n-1}y_{n-2}\ldots y_0$
- $x = 10^{n/2}x_L + x_R$ where $x_L = x_{n-1}\ldots x_{n/2}$ and $x_R = x_{n/2-1}\ldots x_0$
- $y = 10^{n/2}y_L + y_R$ where $y_L = y_{n-1}\ldots y_{n/2}$ and $y_R = y_{n/2-1}\ldots y_0$

Therefore

$$xy = (10^{n/2}x_L + x_R)(10^{n/2}y_L + y_R) = 10^n x_L y_L + 10^{n/2}(x_L y_R + x_R y_L) + x_R y_R$$

Example

$$
\begin{aligned}
1234 \times 5678 &= (100 \times 12 + 34) \times (100 \times 56 + 78) \\
&= 10000 \times 12 \times 56 + 100 \times (12 \times 78 + 34 \times 56) \\
&\quad +34 \times 78
\end{aligned}
$$

Time Analysis

$$xy = (10^{n/2}x_L + x_R)(10^{n/2}y_L + y_R) = 10^n x_L y_L + 10^{n/2}(x_L y_R + x_R y_L) + x_R y_R$$

4 recursive multiplications of number of size $n/2$ each plus 4 additions and left shifts (adding enough 0's to the right)

Time Analysis

$$xy = (10^{n/2}x_L + x_R)(10^{n/2}y_L + y_R) = 10^n x_L y_L + 10^{n/2}(x_L y_R + x_R y_L) + x_R y_R$$

4 recursive multiplications of number of size $n/2$ each plus 4 additions and left shifts (adding enough 0's to the right)

$$T(n) = 4T(n/2) + O(n) \qquad T(1) = O(1)$$

Time Analysis

$$xy = (10^{n/2}x_L + x_R)(10^{n/2}y_L + y_R) = 10^n x_L y_L + 10^{n/2}(x_L y_R + x_R y_L) + x_R y_R$$

4 recursive multiplications of number of size $n/2$ each plus 4 additions and left shifts (adding enough 0's to the right)

$$T(n) = 4T(n/2) + O(n) \qquad T(1) = O(1)$$

$T(n) = \Theta(n^2)$. No better than grade school multiplication!

Time Analysis

$$xy = (10^{n/2}x_L + x_R)(10^{n/2}y_L + y_R) = 10^n x_L y_L + 10^{n/2}(x_L y_R + x_R y_L) + x_R y_R$$

4 recursive multiplications of number of size $n/2$ each plus 4 additions and left shifts (adding enough 0's to the right)

$$T(n) = 4T(n/2) + O(n) \qquad T(1) = O(1)$$

$T(n) = \Theta(n^2)$. No better than grade school multiplication!

Can we invoke Gauss's trick here?

Improving the Running Time

$$xy = (10^{n/2}x_L + x_R)(10^{n/2}y_L + y_R) = 10^n x_L y_L + 10^{n/2}(x_L y_R + x_R y_L) + x_R y_R$$

Gauss trick: $x_L y_R + x_R y_L = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R$

Improving the Running Time

$$xy = (10^{n/2}x_L + x_R)(10^{n/2}y_L + y_R) = 10^n x_L y_L + 10^{n/2}(x_L y_R + x_R y_L) + x_R y_R$$

Gauss trick: $x_L y_R + x_R y_L = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R$

Recursively compute only $x_L y_L, x_R y_R, (x_L + x_R)(y_L + y_R)$.

Improving the Running Time

$$xy = (10^{n/2}x_L + x_R)(10^{n/2}y_L + y_R) = 10^n x_L y_L + 10^{n/2}(x_L y_R + x_R y_L) + x_R y_R$$

Gauss trick: $x_L y_R + x_R y_L = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R$

Recursively compute only $x_L y_L, x_R y_R, (x_L + x_R)(y_L + y_R)$.

Time Analysis

Running time is given by

$$T(n) = 3T(n/2) + O(n) \qquad T(1) = O(1)$$

which means

Improving the Running Time

$$xy = (10^{n/2}x_L + x_R)(10^{n/2}y_L + y_R) = 10^n x_L y_L + 10^{n/2}(x_L y_R + x_R y_L) + x_R y_R$$

Gauss trick: $x_L y_R + x_R y_L = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R$

Recursively compute only $x_L y_L, x_R y_R, (x_L + x_R)(y_L + y_R)$.

Time Analysis

Running time is given by

$$T(n) = 3T(n/2) + O(n) \qquad T(1) = O(1)$$

which means $T(n) = O(n^{\log_2 3}) = O(n^{1.585})$

State of the Art

Schönhage-Strassen 1971: $O(n \log n \log \log n)$ time using Fast-Fourier-Transform (FFT)

Martin Fürer 2007: $O(n \log n 2^{O(\log^* n)})$ time

Conjecture: there is an $O(n \log n)$ time algorithm

Analyzing the Recurrences

- Basic divide and conquer: $T(n) = 4T(n/2) + O(n)$, $T(1) = 1$. **Claim:** $T(n) = \Theta(n^2)$.
- Saving a multiplication: $T(n) = 3T(n/2) + O(n)$, $T(1) = 1$. **Claim:** $T(n) = \Theta(n^{1+\log 1.5})$

Analyzing the Recurrences

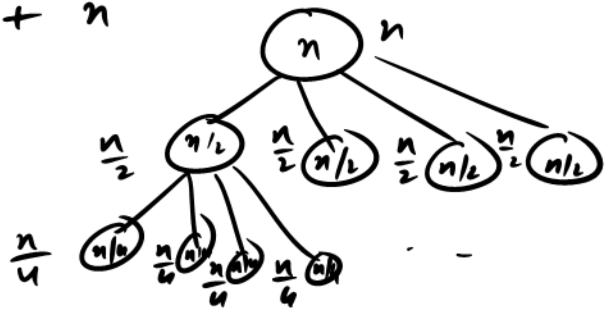
- Basic divide and conquer: $T(n) = 4T(n/2) + O(n)$, $T(1) = 1$. **Claim:** $T(n) = \Theta(n^2)$.
- Saving a multiplication: $T(n) = 3T(n/2) + O(n)$, $T(1) = 1$. **Claim:** $T(n) = \Theta(n^{1+\log 1.5})$

Use recursion tree method:

- In both cases, depth of recursion $L = \log n$.
- Work at depth i is $4^i n/2^i$ and $3^i n/2^i$ respectively: number of children at depth i times the work at each child
- Total work is therefore $n\sum_{i=0}^{L} 2^i$ and $n\sum_{i=0}^{L}(3/2)^i$ respectively.

Recursion tree analysis

$$T(n) = 4T\left(\frac{n}{2}\right) + n$$



$$n + 4 \cdot \frac{n}{2} + 4^2 \cdot \frac{n}{4} + \cdots + 4^i \cdot \frac{n}{2^i} + \cdots,$$

$$= n + 2 \cdot n + 4 \cdot n + \cdots + 2^i \cdot n + \cdots$$

$$n\left(1+2+2^2+\cdots+2^L\right)$$

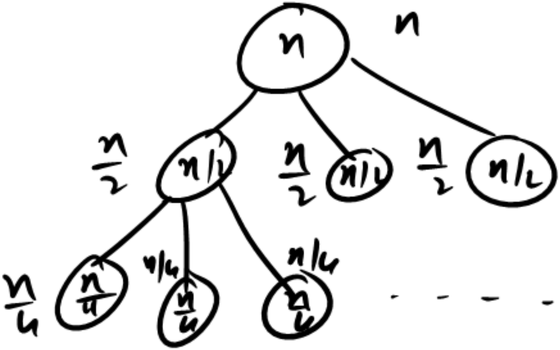
$$L = \log n$$

$$= n \cdot \frac{2^{L+1}-1}{2-1} = n \cdot \left(2^{\log_2 n+1}-1\right)$$

$$= n\left(n \cdot 2 - 1\right)$$

$$= 2n^2 - n$$

$$T(n) = 3T\left(\frac{n}{2}\right) + n$$



$$n + 3 \cdot \frac{n}{2} + 3^2 \cdot \frac{n}{2^2} + \cdots + 3^i \frac{n}{2^i} + \cdots + \frac{3^L}{2^L} \cdot 1$$

$$T(n) = n\left(1 + \frac{3}{2} + \left(\frac{3}{2}\right)^2 + \cdots + \left(\frac{3}{2}\right)^i + \cdots + \left(\frac{3}{2}\right)^L\right)$$

$$= n \cdot \frac{\left(\frac{3}{2}\right)^{L+1} - 1}{\frac{3}{2} - 1}$$

$$= 2n \left(\frac{3}{2}\right)^{\log_2 n + 1}$$

$$= 2n \cdot \left(\frac{3}{2}\right)^{\frac{\log_{\frac{3}{2}} n}{\log_{3/2} 2}}$$

$$= 2n \cdot (n)^{\frac{1}{\log_{3/2} 2}} = 2n \cdot n^{\log_2 3/2}$$