EE360C: Algorithms

Divide and Conquer

Pedro Santacruz

Department of Electrical and Computer Engineering
University of Texas at Austin

Fall 2014

APPROACHES TO ALGORITHM DESIGN

- **incremental design** – given a sorted subarray, insertion sort inserts a single element into its proper place, generating a longer sorted subarray
- **greedy choice** – frame the problem as a choice; make the *greedy* (obvious, naïve) choice
- **divide and conquer design** – break the problem into several subproblems that are similar to but smaller than the original
  - solve the subproblems recursively
  - combine solutions to subproblems to get a solution to the original problem

The Divide and Conquer Approach

Divide: the problem into a number of subproblems

Conquer: the subproblems by solving them recursively. If the subproblems are small enough, just solve the subproblems directly.

Combine: the solutions to the subproblems into the solution for the original problem.

Have you used a divide and conquer algorithm before?

Exponentiation

Given a number a and a positive integer n, consider the problem of calculating the expression a^n. How would you write a program?

SlowPower(a, n)

```
x ← a
for i ← 2 to n
        do x ← x × a
return x
```

What is the running time of this algorithm?
It does $n-1$ multiplications, so assuming multiplication is $O(1)$, it's $\Theta(n)$.
We can actually do this for anything that can be multiplied, but it would have a different running time if multiplication was more expensive.

Can we do better?

Divide and Conquer Exponentiation

Consider the following equality: $a^n = a^{\lfloor n/2 \rfloor} \times a^{\lceil n/2 \rceil}$. This is the constant time addition of the solutions to two smaller subproblems. On top of that, the smaller subproblems are really close to the same thing. How would you write this program?

```
FastPower(a, n)
if n = 1
    then return a
    else
        x ← FastPower(a, ⌊n/2⌋)
        if n is even
           then return x × x
           else
               return x × x × a
```

What's the running time of this algorithm? How many problem instances does it make if it divides the problem in half every recursive call? What is the running time of each of those instances?
It makes $\lg n$ instances, each of which takes $O(1)$ time to run, resulting in an overall running time of $\Theta(\lg n)$.

Divide and Conquer Sorting

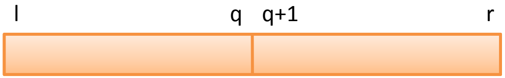
Merge-Sort

Divide: divide the n-element sequence to be sorted into two subsequences of $n/2$ elements each.

Conquer: sort the two subsequences recursively using merge-sort.

Combine: Merge the two sorted subsequences to produce the sorted answer.

The recursion ends when the sequence to be sorted has length 1 and is therefore already sorted.

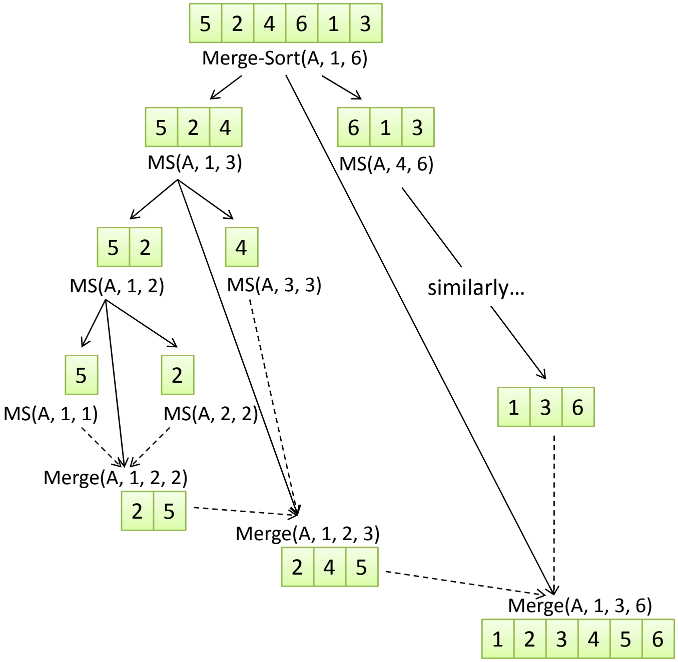


$$q = \left\lfloor \frac{(l+r)}{2} \right\rfloor$$

MERGE-SORT

- Assume the availability of a subroutine MERGE(A, p, q, r), where A is an array, and p, q, and r are indices numbering elements of the array such that $p \leq q \leq r$.
- MERGE assumes that the subarrays $A[p \mathinner{..} q]$ and $A[q+1 \mathinner{..} r]$ are in sorted order.
- It merges them to form a single sorted subarray that replaces the current subarray $A[p \mathinner{..} r]$.

```
MERGE-SORT(A, p, r)
if p < r
   then q ← ⌊(p + r)/2⌋
        MERGESORT(A, p, q)
        MERGESORT(A, q + 1, r)
        MERGE(A, p, q, r)
```


5 2 4 6 1 3
Merge-Sort(A, 1, 6)
5 2 4
MS(A, 1, 3)
6 1 3
MS(A, 4, 6)
5 2
MS(A, 1, 2)
4
MS(A, 3, 3)
similarly...
5
MS(A, 1, 1)
2
MS(A, 2, 2)
1 3 6
Merge(A, 1, 2, 2)
2 5
Merge(A, 1, 2, 3)
2 4 5
Merge(A, 1, 3, 6)
1 2 3 4 5 6

MERGE-SORT

MERGE(A, p, q, r)

1 $n_1 \leftarrow q - p + 1$
2 $n_2 \leftarrow r - q$
3 create arrays $L[1 \mathinner{..} n_1 + 1]$ and $R[1 \mathinner{..} n_2 + 1]$
4 **for** $i \leftarrow 1$ **to** n_1
5   **do** $L[i] \leftarrow A[p + i - 1]$
6 **for** $j \leftarrow 1$ **to** n_2
7   **do** $R[j] \leftarrow A[q + j]$
8 $L[n_1 + 1] \leftarrow \infty$
9 $R[n_2 + 1] \leftarrow \infty$
10 $i \leftarrow 1$
11 $j \leftarrow 1$
12 **for** $k \leftarrow p$ **to** r
13   **do if** $L[i] \leq R[j]$
14   **then** $A[k] \leftarrow L[i]$
15             $i \leftarrow i + 1$
16   **else** $A[k] \leftarrow R[j]$
17             $j \leftarrow j + 1$

Line 1 computes the length of the subarray $A[p \mathinner{..} q]$ Line 2 computes the length of the subarray $A[q + 1 \mathinner{..} r]$ Line 3 creates the arrays L and R Lines 4–5 copy the subarray $A[p \mathinner{..} q]$ into L Lines 6–7 copy the subarray $A[q + 1 \mathinner{..} r]$ into R Lines 8 and 9 put sentinels at the ends of L and R Lines 10–17 are the meat of the merging

Merge Details

```
Start
for k ← p to r
     do if L[i] ≤ R[j]
           then A[k] ← L[i]
                i ← i + 1
           else A[k] ← R[j]
                j ← j + 1
```

What is the loop invariant?

- At the start of each iteration of the **for** loop, the subarray $A[p\,..\,k-1]$ contains the $k-p$ smallest elements, in sorted order
- $L[i]$ and $R[i]$ are the smallest elements of their arrays that have not been copied back into A.

The Merge Loop Invariant Intuition

What is the loop invariant?

- At the start of each iteration of the **for** loop, the subarray $A[p\,..\,k-1]$ contains the $k-p$ smallest elements, in sorted order
- $L[i]$ and $R[i]$ are the smallest elements of their arrays that have not been copied back into A.

	8	9	10	11	12	13	14	15	16	17
A		2	4	5	7	1	2	3	6	
		k								

	1	2	3	4	5
L	2	4	5	7	∞
	i				

	1	2	3	4	5
R	1	2	3	6	∞
	j				

	8	9	10	11	12	13	14	15	16	17
A		1	4	5	7	1	2	3	6	
			k							

Merge Loop Invariant Correctness

What is the loop invariant?

- At the start of each iteration of the **for** loop, the subarray $A[p \mathinner{..} k-1]$ contains the $k-p$ smallest elements, in sorted order
- $L[i]$ and $R[i]$ are the smallest elements of their arrays that have not been copied back into A.

Initialization: *the loop invariant is true initially*. Prior to the first iteration of the loop, $k = p$, so the subarray $A[p \mathinner{..} k-1]$ is empty.

Maintenance: *the loop invariant remains true after each iteration of the loop*. Suppose $L[i] \leq R[j]$. Because $A[p \mathinner{..} k-1]$ already contains the $k-p$ smallest elements, after $L[i]$ is copied into $A[k]$, $A[p \mathinner{..} k]$ will contain the $k-p+1$ smallest elements. Incrementing k reestablishes the loop invariant.

ANALYZING MERGE

What is the *running time* of MERGE?

- The assignments (Lines 1–3 and 8–11) all take constant time.
- Copying into L and R from A (Lines 4–7) takes $n_1 + n_2 = n$ where n is the number of elements in A.
- The **for** loop (Lines 12–17) takes constant time and is executed n times (once for every element).

$\therefore$ the running time of MERGE is $\Theta(n)$.

What is the running time of MERGE-SORT?

Analyzing Divide and Conquer Algorithms

- We often express the running time of a recursive algorithm using a **recurrence equation** or just **recurrence**.
- A recurrence for a divide and conquer algorithm is based on the three steps:
  - divide
  - conquer
  - combine

Generic Divide and Conquer Recurrence

- Let $T(n)$ be the running time of a problem of size n.
- If the problem size is small enough (i.e., $n \le c$ for some constant c), then the straightforward solution takes constant time, or $\Theta(1)$.
- Suppose the **divide** step generates a subproblems, each of which are a fraction $1/b$ of the original problem size.
- Assume that the divide step takes $D(n)$ time and the combine step takes $C(n)$ time.

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le c, \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

MERGE-SORT ANALYSIS

Divide: this step simply computes the middle of the subarray: $\Theta(1)$

Conquer: we define 2 subproblems, each of size $n/2$, which contributes $2T(n/2)$ to the running time.

Combine: the MERGE procedure takes $\Theta(n)$ times

Therefore, the *worst case running time* of MERGE-SORT is:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

- This is a final running time of $\Theta(n\, lg\, n)$.
- This is intuitive in this case. Why?
- We'll see how to solve these generically using the *master method*.

The Master Method

Consider:

$$T(n) = aT(n/b) + f(n)$$

where $a \geq 1$ and $b > 1$ and $f(n)$ is an asymptotically positive function.

These often result from divide and conquer approaches.

The **master method** is a cookbook method for solving these common recurrences

The Master Method (cont.)

The Master Theorem

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the non-negative integers by the recurrence:

$$T(n) = aT(n/b) + f(n)$$

where we interpret n/b to be either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ can be bounded asymptotically as follows:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$ and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

The ϵ factor is really n^{ϵ}; this is required because the function $f(n)$ must be polynomially different from $n^{\log_b a}$.

A Quick Master Method Example

- What is the running time of:

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

More Master Method Examples

- What is the running time of:

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

More Master Method Examples

- What is the running time of:

$$T(n) = T(2n/3) + 1$$

More Master Method Examples

- What is the running time of:

$$T(n) = 3T(n/4) + n \lg n$$

More Master Method Examples

- What is the running time of:

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

The Master Method Revisited

The master method can also be written as:

Solve any recurrence of the form:

$$\begin{aligned} T(n) &= aT(n/b) + \Theta(n^l\,(\lg n)^k) \\ T(c) &= \Theta(1) \text{ for some constant } c \end{aligned}$$

where $a \geq 1$, $b > 1$, $l \geq 0$, and $k \geq 0$.

The goal is to compare l and $\log_b a$. The intuition is that $n^{\log_b a}$ is the number of times the termination condition ($T(c)$) is reached.

The Master Method Revisited (cont.)

Then we can restate the three cases as:

- Case 1: $l < \log_b a$. Then $T(n) = \Theta(n^{\log_b a})$.
- Case 2: $l = \log_b a$. Then $T(n) = \Theta(f(n) \lg n)$. Which is equivalent to $T(n) = \Theta(n^l (\lg n)^{k+1})$. Which is ultimately equivalent to (a more generic statement of) what we had before: $T(n) = \Theta(n^{\log_b a} (\lg n)^{k+1})$.
- Case 3: $l > \log_b a$. Then $T(n) = \Theta(f(n)) = \Theta(n^l (\lg n)^k)$.

Multiplication

Consider the traditional iterative "ripple carry" algorithm for addition. What is its running time for adding two n-digit numbers?
Assuming one-digit addition is a constant time operation, $O(n)$.

What is our common algorithm for multiplying two n-digit numbers?
We use n one-digit multiplications and n n-digit additions. What is the running time of this algorithm?
It's fairly straightforward to write as a pair of nested for loops, each done $O(n)$ times, so it's $O(n^2)$ overall.

Can we do better?

Recursive Multiplication I

We start recursive multiplication by observing:

$$(10^m a + b)(10^m c + d) = 10^{2m} ac + 10^m (bc + ad) + bd$$

```
MULTIPLY(x, y, n)
  if n = 1
  return x × y
     else
  m ← ⌈n/2⌉
  a ← ⌊x/10^m⌋; b ← x mod 10^m
  d ← ⌊y/10^m⌋; c ← y mod 10^m
  e ← MULTIPLY(a, c, m)
  f ← MULTIPLY(b, d, m)
  g ← MULTIPLY(b, c, m)
  h ← MULTIPLY(a, d, m)
  return 10^{2m} e + 10^m (g + h) + f
```

Can you write the recurrence?

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

Can you solve the recurrence? Use the master method (revisited) where $a = 4$, $b = 2$, $l = 1$, and $k = 0$. $\log_b a = \log_2 4 = 2$. So $l < \log_b a$, so use Case 1. Then $T(n) = \Theta(n^{\log_b a}) = \Theta(n^2)$

Recursive Multiplication II

We can replace two multiplications with two we're already doing anyway and one additional one.

$$ac + bd - (a - b)(c - d) = bc + ad$$

$\textsc{FastMultiply}(x, y, n)$

if $n = 1$
return $x \times y$
    else
$m \leftarrow \lceil n/2 \rceil$
$a \leftarrow \lfloor x/10^m \rfloor$; $b \leftarrow x \bmod 10^m$
$d \leftarrow \lfloor y/10^m \rfloor$; $c \leftarrow y \bmod 10^m$
$e \leftarrow \textsc{Multiply}(a, c, m)$
$f \leftarrow \textsc{Multiply}(b, d, m)$
$g \leftarrow \textsc{Multiply}(a - b, c - d, m)$
return $10^{2m}e + 10^m(e + f - g) + f$

What's the recurrence for this one?

$$T(n) = 3T(\lceil n/2 \rceil) + O(n)$$

Which solves to? $\Theta(n^{\lg 3}) = \Theta(n^{1.585})$.

Matrix Multiplication

Matrix Multiplication

Given two n-by-n matrices, A and B, compute $C = AB$

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

$$\begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix}$$

Brute Force

$\Theta(n^3)$ arithmetic operations

Fundamental Question

Can we improve upon brute force?

Matrix Multiplication Warmup

Divide and Conquer

- Divide: partition A and B into $\frac{1}{2}n$ by $\frac{1}{2}n$ blocks
- Conquer: multiply 8 $\frac{1}{2}n$ by $\frac{1}{2}n$ recursively
- Combine: add appropriate products using 4 matrix additions

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$\begin{aligned} C_{11} &= (A_{11} \times B_{11}) + (A_{12} \times B_{21}) \\ C_{12} &= (A_{11} \times B_{12}) + (A_{12} \times B_{22}) \\ C_{21} &= (A_{21} \times B_{11}) + (A_{22} \times B_{21}) \\ C_{22} &= (A_{21} \times B_{12}) + (A_{22} \times B_{22}) \end{aligned}$$

$$\mathrm{T}(n) = \underbrace{8T(n/2)}_{\text{recursive calls}} + \underbrace{\Theta(n^2)}_{\text{add, form submatrices}} \Rightarrow \mathrm{T}(n) = \Theta(n^3)$$

Matrix Multiplication: Key Idea

Key Idea

Multiply 2-by-2 block matrices with only 7 multiplications (7 multiplications and 18 additions/subtractions)

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$\begin{aligned} C_{11} &= P_5 + P_4 - P_2 + P_6 \\ C_{12} &= P_1 + P_2 \\ C_{21} &= P_3 + P_4 \\ C_{22} &= P_5 + P_1 - P_3 - P_7 \end{aligned}$$

$$\begin{aligned} P_1 &= A_{11} \times (B_{12} - B_{22}) \\ P_2 &= (A_{11} + A_{12}) \times B_{22} \\ P_3 &= (A_{21} + A_{22}) \times B_{11} \\ P_4 &= A_{22} \times (B_{21} - B_{11}) \\ P_5 &= (A_{11} + A_{22}) \times (B_{11} + B_{22}) \\ P_6 &= (A_{12} - A_{22}) \times (B_{21} + B_{22}) \\ P_7 &= (A_{11} - A_{21}) \times (B_{11} + B_{12}) \end{aligned}$$

Fast Matrix Multiplication

Fast matrix multiplication (Strassen 1969)

- Divide: partition A and B into $\frac{1}{2}n$ by $\frac{1}{2}n$ blocks
- Compute: 14 $\frac{1}{2}n$ by $\frac{1}{2}n$ matrices via 10 matrix additions
- Conquer: multiply 7 $\frac{1}{2}n$ by $\frac{1}{2}n$ matrices recursively
- Combine: 7 products into 4 terms using 8 matrix additions

Analysis

- Assume n is a power of 2
- $T(n)$ is the number of arithmetic operations

$$\mathrm{T}(n) = \underbrace{7T(n/2)}_{\text{recursive calls}} + \underbrace{\Theta(n^2)}_{\text{add, subtract}} \Rightarrow \mathrm{T}(n) = \Theta(n^{\log_2 7}) = O(n^{2.81})$$

So, Another Example

- Can $T(n) = 2T(n/2) + \Theta(n/\lg n)$ be solved using the master method?
- $a = 2$, $b = 2$, $l = 1$, and $k = -1$. Whoops!

As an exercise, use the original master method formulation and show that this formula doesn't fit there either. (It doesn't.)

So how do we solve generic recurrences?

Recursion Trees

In a **recursion tree**, each node represents the cost of a single subproblem somewhere in the set of recursive function invocations.

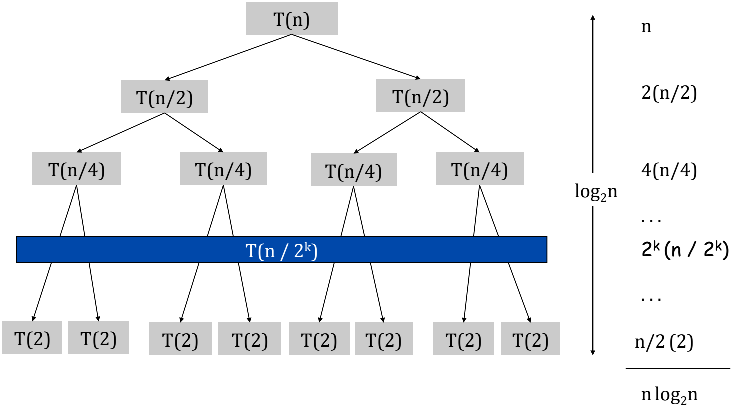
- Sum the nodes in each level to get a per-level cost
- Sum all of the levels to get a total cost

Recursion trees are particularly useful for divide and conquer problems.

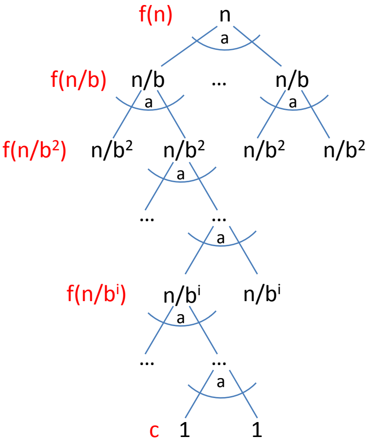
Recursion Trees (cont.)

A simple example:

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$



Recursion Trees (cont.)



level	num. of nodes	time per node
0	1	$f(n)$
1	a	$f(n/b)$
2	a^2	$f(n/b^2)$
i	a^i	$f(n/b^i)$
$\log_b n$	$a^{\log_b n}$	c

Recursion Tree Summation

Before we get to the full summation, we need the following fact:

$$a^{\log_b n} = n^{\log_b a}$$

You can verify this by taking the $\log_b$ of both sides.

So we can get the running time of the entire recurrence by summing up all of the levels:

$$T(n) = \left[\sum_{i=0}^{(\log_b n)-1} a^i \times f(n/b^i) \right] + n^{\log_b a} \times c$$

Recursion Tree Summation (cont.)

$$T(n) = \left[\sum_{i=0}^{(\log_b n)-1} a^i \times f(n/b^i) \right] + n^{\log_b a} \times c$$

The term $f(n/b^i)$ represents the running time of a single subproblem at level i of the recursion tree. This is the second term of our general recurrence statement ($T(n) = aT(n/b) + f(n)$). From the master method, we know that it is useful to write $f(n)$ in terms of l and k:

$$f(n) = \Theta(n^l (\lg n)^k)$$

Substituting n/b^i for n, our sum becomes:

$$T(n) = \left[\sum_{i=0}^{(\log_b n)-1} a^i \times \Theta((n/b^i)^l (\lg (n/b^i))^k) \right] + n^{\log_b a} \times c$$

Recursion Tree Exercise

Use this summation to formulate (and solve) the merge sort recurrence:

$$T(n) = 2T(\lfloor n/2 \rfloor) + \Theta(n)$$

Here's that summation again:

$$T(n) = \left[\sum_{i=0}^{(\log_b n)-1} a^i \times \Theta((n/b^i)^l \, (\lg \, (n/b^i))^k) \right] + n^{\log_b a} \times c$$

RECURSION TREE EXAMPLE

Let's use a recursion tree to solve:

$$T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$$

Substituting into the summation, we have:

$$T(n) = \left[\sum_{i=0}^{(\log_4 n)-1} 3^i \times f(n/4^i) \right] + \Theta(n^{\log_4 3})$$

Restating using the Θ expression from master method:

$$T(n) = \left[\sum_{i=0}^{(\log_4 n)-1} 3^i \times \Theta((n/4^i)^l \, (\lg\,(n/4^i))^k) \right] + \Theta(n^{\log_4 3})$$

Since $l = 2$ and $k = 0$, this reduces to:

$$T(n) = \left[\sum_{i=0}^{(\log_4 n)-1} 3^i \times \Theta(n^2/16^i) \right] + \Theta(n^{\log_4 3})$$

Recursion Tree Example (cont.)

So how do we know what the running time is, given:

$$T(n) = \left[\sum_{i=0}^{(\log_4 n)-1} 3^i \times \Theta((n/4^i)^2) \right] + \Theta(n^{\log_4 3})$$

We have to solve the sum. First let's rewrite it a little bit:

$$T(n) = cn^2 \left[\sum_{i=0}^{(\log_4 n)-1} (3/16)^i \right] + \Theta(n^{\log_4 3})$$

Then we can apply the geometric series rule (Appendix A, A.5) and get:

$$cn^2 \left[\frac{(3/16)^{\log_4 n} - 1}{3/16 - 1} \right] + \Theta(n^{\log_4 3})$$

Recursion Tree Example (cont.)

Wow. That's not helpful. What does this mean?

$$cn^2\left[\frac{(3/16)^{\log_4 n}-1}{3/16-1}\right]+\Theta(n^{\log_4 3})$$

But we can take a step back and say that this finite geometric series must be less than the geometric series that is the same other than being infinite. That is:

$$T(n) < cn^2\left[\sum_{i=0}^{\infty}(3/16)^i\right]+\Theta(n^{\log_4 3})$$

Which is a simpler summation to solve:

$$\begin{aligned} T(n) &< \frac{16}{13}cn^2+\Theta(n^{\log_4 3}) \\ &= O(n^2) \end{aligned}$$

Recall the original recurrence ($T(n)=3T(\lfloor n/4\rfloor)+\Theta(n^2)$). The first recursive step takes $\Theta(n^2)$ time, so, given that the entire recurrence is $O(n^2)$, it must also be $\Theta(n^2)$.

Recurrences and Induction

Let's say we wanted to verify that this was the correct answer. How would we do it? Induction!

Our original recurrence was: $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$, and our guess is $O(n^2)$.

Our base case is taken care of by the fact that we assume $T(n) = \Theta(1)$ when $n = 1$.

Inductive step: we need to prove that $T(n) \leq cn^2$ for an appropriate choce of c. We assume that the claim holds for $\lfloor n/4 \rfloor$, i.e., $T(\lfloor n/4 \rfloor) \leq c(\lfloor n/4 \rfloor)^2$. Then

$$\begin{aligned} T(n) &\leq 3(c(\lfloor n/4 \rfloor)^2) + \Theta(n^2) \\ &\leq 3cn^2/16 + dn^2 \\ &\leq cn^2 \end{aligned}$$

which is true for values $c \geq (16/3)d$

Recurrences and Induction Exercise

Solve the merge sort recurrence by induction:

$$T(n) = 2T(\lfloor n/2 \rfloor) + \Theta(n)$$

Summations

Solving recurrences like this requires copious use of summations like the one above. Look these up.

The most common:

Arithmetic Series

$$\sum_{k=1}^{n} k = \frac{1}{2}n(n+1)$$

Geometric Series

$$\sum_{k=0}^{n} x^k = \frac{x^{n+1}-1}{x-1}$$

Infinite Geometric Series

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$$

Another Example: Closest Pairs

Closest Pair

Given n points in the plane, find a pair with smallest Euclidean distance between them.

A Fundmental geometric primitive

Used in raphics, computer vision, geographic information systems molecular modeling, air traffic control.

Brute Force

Check all pairs of points p and q with $\Theta(n^2)$ comparisons.

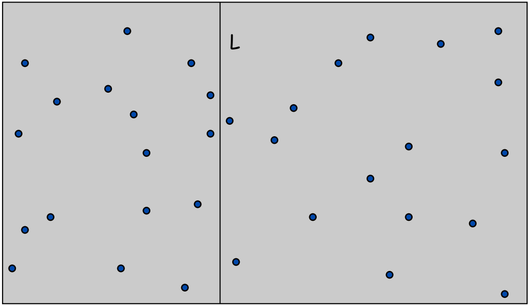
Can we do better?

- To make the presentation clearer, let's assume no two points have the same x coordinate.

Closest Pair of Points

Algorithm: Divide

Draw a vertical line L so that roughly $1/2n$ points are on each side of L.



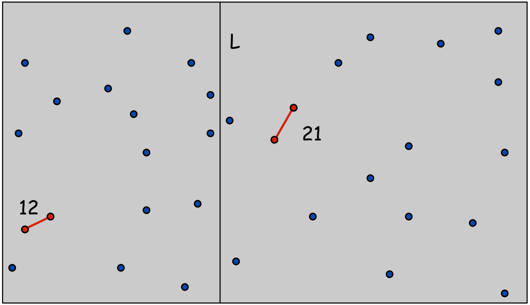
Closest Pair of Points

Algorithm: Divide

Draw a vertical line L so that roughly $1/2n$ points are on each side of L.

Algorithm: Conquer

Find the closest pair in each side recursively.



Closest Pair of Points

Divide

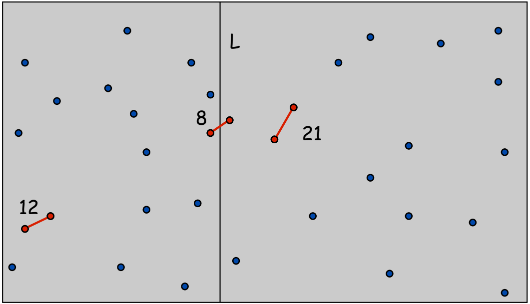
Draw a vertical line L so that roughly $1/2n$ points are on each side of L.

Conquer

Find the closest pair in each side recursively.

Combine

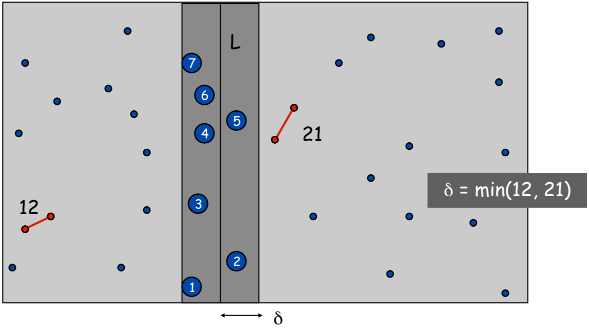
Finde the closest pair with one point in each side. Return the best of the three solutions.



Closest Pair: Combine Step

Find the closest pair with one point on each side, assuming that *distance* $< \delta$ (where δ is the minimum of the closest pair distance in the two halves.

- We only need to consider points within δ of L
- Sort points in 2δ strip by their y coordinate
- Only check distances of those within 11 positions in the sorted list



Closest Pair: Combine Step (II)

Definition

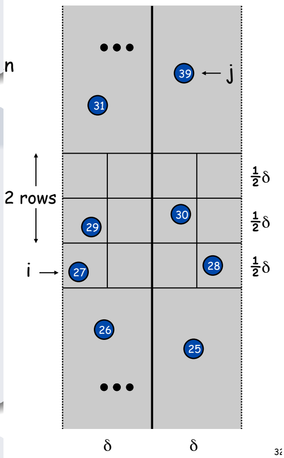
Let s_i be the point in the 2δ-strip with the i^{th} smallest y coordinate.

Claim

If $|i - j| \geq 12$ then the distance between s_i and s_j is at least δ.

- No two points lie in the same $\frac{1}{2}\delta$ by $\frac{1}{2}\delta$ box.
- Two points at least 2 rows apart have distance $\geq 2(\frac{1}{2}\delta)$.

This is actually also true if you replace 12 with 7.



Closest Pair Algorithm

ClosestPair($p_1, p_2, \ldots, p_n$)

Compute L s.t. half the points are on each side of L
$\delta_1 = \text{ClosestPair}(p_1, \ldots, p_L)$
$\delta_2 = \text{ClosestPair}(p_{L+1}, \ldots, p_n)$
$\delta = \min \delta_1, \delta_2$
Delete all points further than δ from L
Sort remaining points by y-coordinate
Scan points in y order and compare distance between each point and next 11 neighbors. If any of these distances is less than δ, update δ.
return δ

Closest Pair Analysis

- Time to sort original points: $O(n \log n)$
- Divide: $O(n)$
- Conquer: 2 subproblems of size $n/2$
- Combine: $O(n \log n)$

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

Which solves, by the Master Method, to $O(n \log^2 n)$

Which we can reduce to $O(n \log n)$ by pre-sorting the y coordinates before we start.

QUESTIONS

?