

Recurrences, and Linear Time Selection

Lecture 12
March 3, 2015

Part I

Recurrences

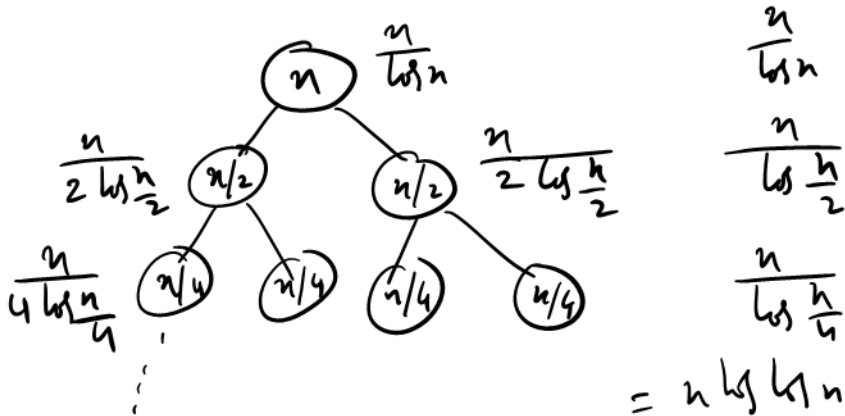
Solving Recurrences

Two general methods:

- 1 Recursion tree method: need to do sums
 - 1 elementary methods, geometric series
 - 2 integration
- 2 Guess and Verify
 - 1 guessing involves intuition, experience and trial & error
 - 2 verification is via induction

Recurrence: Example I

- 1 Consider $T(n) = 2T(n/2) + n/\log n$. $T(2) = 1$



$$\frac{n}{\log n} + \frac{n}{\log n - 1} + \frac{n}{\log n - 2} + \dots + \frac{n}{1} = n \left(\frac{1}{\log n} + \frac{1}{\log n - 1} + \dots + \frac{1}{1} \right)$$

Recurrence: Example I

- 1 Consider $T(n) = 2T(n/2) + n/\log n$.
- 2 Construct recursion tree, and observe pattern.

Recurrence: Example I

- 1 Consider $T(n) = 2T(n/2) + n / \log n$.
- 2 Construct recursion tree, and observe pattern. i th level has 2^i nodes, and problem size at each node is $n/2^i$ and hence work at each node is $\frac{n}{2^i} / \log \frac{n}{2^i}$.

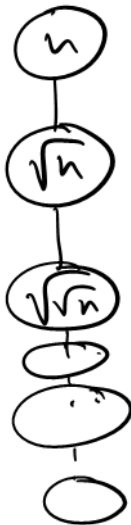
Recurrence: Example I

- 1 Consider $T(n) = 2T(n/2) + n/\log n$.
- 2 Construct recursion tree, and observe pattern. i th level has 2^i nodes, and problem size at each node is $n/2^i$ and hence work at each node is $\frac{n}{2^i} / \log \frac{n}{2^i}$.
- 3 Summing over all levels

$$\begin{aligned} T(n) &= \sum_{i=0}^{\log n - 1} 2^i \left[\frac{(n/2^i)}{\log(n/2^i)} \right] \\ &= \sum_{i=0}^{\log n - 1} \frac{n}{\log n - i} \\ &= n \sum_{j=1}^{\log n} \frac{1}{j} = nH_{\log n} = \Theta(n \log \log n) \end{aligned}$$

Recurrence: Example II

- 1 Consider $T(n) = T(\sqrt{n}) + 1$



$$n^1$$

$$n^{\frac{1}{2}}$$

$$n^{\frac{1}{4}}$$

$$n^{\frac{1}{8}}$$

$$n^{\frac{1}{16}}$$

$$n^{\frac{1}{32}}$$

$$n^{\frac{1}{2^k}} = 2$$

Recurrence: Example II

① Consider $T(n) = T(\sqrt{n}) + 1$

② What is the depth of recursion?

$\sqrt{n}, \sqrt{\sqrt{n}}, \sqrt{\sqrt{\sqrt{n}}}, \dots, O(1).$

Recurrence: Example II

① Consider $T(n) = T(\sqrt{n}) + 1$

② What is the depth of recursion?

$\sqrt{n}, \sqrt{\sqrt{n}}, \sqrt{\sqrt{\sqrt{n}}}, \dots, O(1).$

③ Number of levels: $n^{2^{-L}} = 2$ means $L = \log \log n.$

Recurrence: Example II

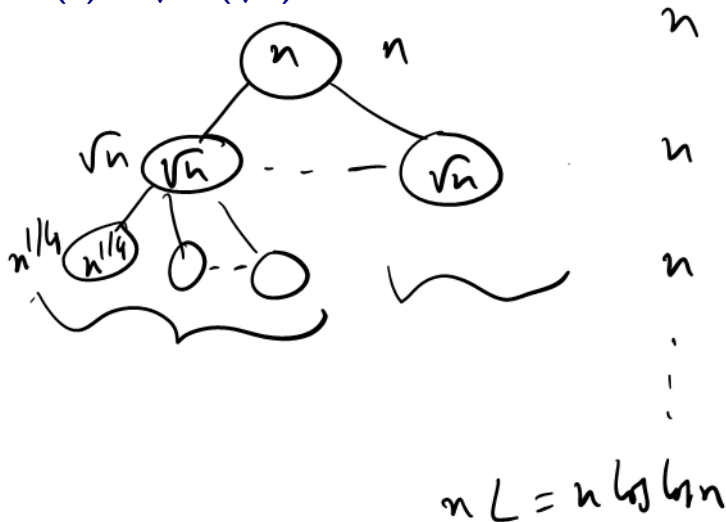
- 1 Consider $T(n) = T(\sqrt{n}) + 1$
- 2 What is the depth of recursion?
 $\sqrt{n}, \sqrt{\sqrt{n}}, \sqrt{\sqrt{\sqrt{n}}}, \dots, O(1)$.
- 3 Number of levels: $n^{2^{-L}} = 2$ means $L = \log \log n$.
- 4 Number of children at each level is **1**, work at each node is **1**

Recurrence: Example II

- 1 Consider $T(n) = T(\sqrt{n}) + 1$
- 2 What is the depth of recursion?
 $\sqrt{n}, \sqrt{\sqrt{n}}, \sqrt{\sqrt{\sqrt{n}}}, \dots, O(1)$.
- 3 Number of levels: $n^{2^{-L}} = 2$ means $L = \log \log n$.
- 4 Number of children at each level is 1 , work at each node is 1
- 5 Thus, $T(n) = \sum_{i=0}^L 1 = \Theta(L) = \Theta(\log \log n)$.

Recurrence: Example III

- 1 Consider $T(n) = \sqrt{n}T(\sqrt{n}) + n$.



Recurrence: Example III

- 1 Consider $T(n) = \sqrt{n}T(\sqrt{n}) + n$.
- 2 Using recursion trees: number of levels $L = \log \log n$

Recurrence: Example III

- 1 Consider $T(n) = \sqrt{n}T(\sqrt{n}) + n$.
- 2 Using recursion trees: number of levels $L = \log \log n$
- 3 Work at each level? Root is n , next level is $\sqrt{n} \times \sqrt{n} = n$.
Can check that each level is n .

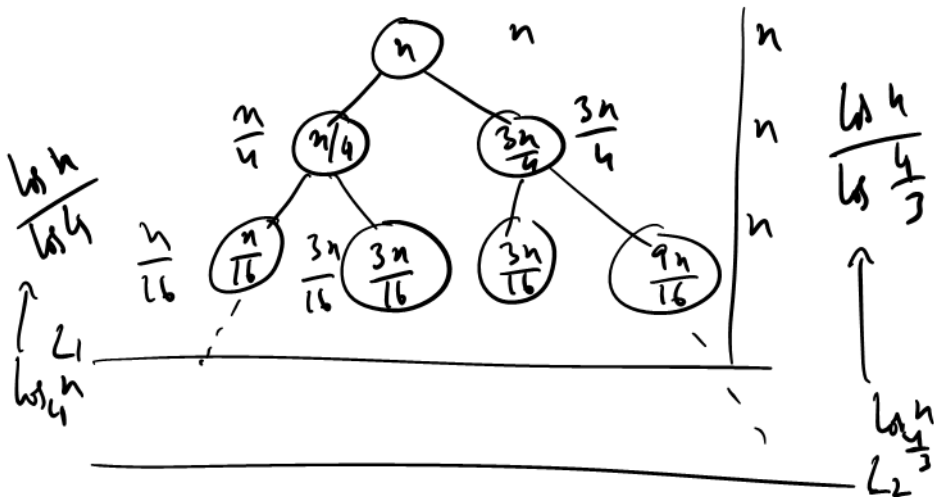
Recurrence: Example III

- 1 Consider $T(n) = \sqrt{n}T(\sqrt{n}) + n$.
- 2 Using recursion trees: number of levels $L = \log \log n$
- 3 Work at each level? Root is n , next level is $\sqrt{n} \times \sqrt{n} = n$.
Can check that each level is n .
- 4 Thus, $T(n) = \Theta(n \log \log n)$

Recurrence: Example IV

- 1 Consider $T(n) = T(n/4) + T(3n/4) + n$.

$$T(1) = 1$$



Recurrence: Example IV

- 1 Consider $T(n) = T(n/4) + T(3n/4) + n$.
- 2 Using recursion tree, we observe the tree has leaves at different levels (a *lop-sided* tree).

Recurrence: Example IV

- 1 Consider $T(n) = T(n/4) + T(3n/4) + n$.
- 2 Using recursion tree, we observe the tree has leaves at different levels (a *lop-sided* tree).
- 3 Total work in any level is at most n . Total work in any level without leaves is exactly n .

Recurrence: Example IV

- 1 Consider $T(n) = T(n/4) + T(3n/4) + n$.
- 2 Using recursion tree, we observe the tree has leaves at different levels (a *lop-sided* tree).
- 3 Total work in any level is at most n . Total work in any level without leaves is exactly n .
- 4 Highest leaf is at level $\log_4 n$ and lowest leaf is at level $\log_{4/3} n$

Recurrence: Example IV

- 1 Consider $T(n) = T(n/4) + T(3n/4) + n$.
- 2 Using recursion tree, we observe the tree has leaves at different levels (a *lop-sided* tree).
- 3 Total work in any level is at most n . Total work in any level without leaves is exactly n .
- 4 Highest leaf is at level $\log_4 n$ and lowest leaf is at level $\log_{4/3} n$
- 5 Thus, $n \log_4 n \leq T(n) \leq n \log_{4/3} n$, which means $T(n) = \Theta(n \log n)$

Part II

Selecting in Unsorted Lists

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:

- 1 array: 16, 12, 14, 20, 5, 3, 18, 19, 1
- 2 pivot: 16
- 3 split into 12, 14, 5, 3, 1 and 20, 19, 18 and recursively sort
- 4 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:

- 1 array: 16, 12, 14, 20, 5, 3, 18, 19, 1
- 2 pivot: 16
- 3 split into 12, 14, 5, 3, 1 and 20, 19, 18 and recursively sort
- 4 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:

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

Time Analysis

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

Time Analysis

- 1 Let k be the rank of the chosen pivot. Then,
$$T(n) = T(k - 1) + T(n - k) + O(n)$$
- 2 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

- 1 Let k be the rank of the chosen pivot. Then,
$$T(n) = T(k - 1) + T(n - k) + O(n)$$
- 2 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, $T(n) = O(n \log n)$.
 - 1 Theoretically, median can be found in linear time.

Time Analysis

- 1 Let k be the rank of the chosen pivot. Then,
 $T(n) = T(k - 1) + T(n - k) + O(n)$
- 2 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, $T(n) = O(n \log n)$.
 - 1 Theoretically, median can be found in linear time.
- 3 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.

Prune and search

Consider an algorithm **alg** that is given an input of size **n**. In **$O(n)$** time, it either solves the problem, or solve it by calling recursively on an input of size, say, $\leq (15/16)n$. The running time of **alg** is

- (A) **$O(n^2)$**
- (B) **$O(n \log n)$**
- (C) **$O(n)$**
- (D) There are not enough details.

Problem - Selection

Input Unsorted array **A** of **n** integers

Goal Find the **j**th smallest number in **A** (*rank j* number)

Example

A = {4, 6, 2, 1, 5, 8, 7} and **j** = 4. The **j**th smallest element is **5**.

Median: $j = \lfloor (n + 1)/2 \rfloor$

Problem - Selection

Input Unsorted array **A** of **n** integers

Goal Find the **j**th smallest number in **A** (*rank j* number)

Example

A = {4, 6, 2, 1, 5, 8, 7} and **j** = 4. The **j**th smallest element is **5**.

Median: $j = \lfloor (n + 1)/2 \rfloor$

Simplifying assumption for sake of notation: elements of **A** are distinct



Algorithm I

- 1 Sort the elements in **A**
- 2 Pick **j**th element in sorted order

Time taken = **$O(n \log n)$**

Algorithm 1

- 1 Sort the elements in **A**
- 2 Pick **j**th element in sorted order

Time taken = **$O(n \log n)$**

Do we need to sort? Is there an **$O(n)$** time algorithm?

Algorithm II

If j is small or $n - j$ is small then

- 1 Find j smallest/largest elements in A in $O(jn)$ time. (How?)
- 2 Time to find median is $O(n^2)$.

Divide and Conquer Approach

- 1 Pick a pivot element \mathbf{a} from \mathbf{A}
- 2 Partition \mathbf{A} based on \mathbf{a} .
 $\mathbf{A}_{\text{less}} = \{x \in \mathbf{A} \mid x \leq \mathbf{a}\}$ and $\mathbf{A}_{\text{greater}} = \{x \in \mathbf{A} \mid x > \mathbf{a}\}$
- 3 $|\mathbf{A}_{\text{less}}| = \mathbf{j}$: return \mathbf{a}
- 4 $|\mathbf{A}_{\text{less}}| > \mathbf{j}$: recursively find \mathbf{j} th smallest element in \mathbf{A}_{less}
- 5 $|\mathbf{A}_{\text{less}}| < \mathbf{j}$: recursively find \mathbf{k} th smallest element in $\mathbf{A}_{\text{greater}}$
where $\mathbf{k} = \mathbf{j} - |\mathbf{A}_{\text{less}}|$.

Time Analysis

- 1 Partitioning step: $O(n)$ time to scan **A**
- 2 How do we choose pivot? Recursive running time?

Time Analysis

- 1 Partitioning step: $O(n)$ time to scan A
- 2 How do we choose pivot? Recursive running time?

Suppose we always choose pivot to be $A[1]$.

Time Analysis

- 1 Partitioning step: $O(n)$ time to scan A
- 2 How do we choose pivot? Recursive running time?

Suppose we always choose pivot to be $A[1]$.

Say A is sorted in increasing order and $j = n$.

Exercise: show that algorithm takes $\Omega(n^2)$ time

A Better Pivot

Suppose pivot is the ℓ th smallest element where $n/4 \leq \ell \leq 3n/4$.

That is pivot is *approximately* in the middle of \mathbf{A}

Then $n/4 \leq |\mathbf{A}_{\text{less}}| \leq 3n/4$ and $n/4 \leq |\mathbf{A}_{\text{greater}}| \leq 3n/4$. If we apply recursion,

A Better Pivot

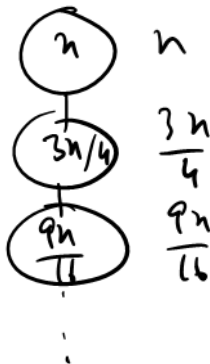
Suppose pivot is the ℓ th smallest element where $n/4 \leq \ell \leq 3n/4$.

That is pivot is *approximately* in the middle of \mathbf{A}

Then $n/4 \leq |\mathbf{A}_{\text{less}}| \leq 3n/4$ and $n/4 \leq |\mathbf{A}_{\text{greater}}| \leq 3n/4$. If we apply recursion,

$$T(n) \leq T(3n/4) + O(n)$$

Implies $T(n) = O(n)$!



A Better Pivot

Suppose pivot is the ℓ th smallest element where $n/4 \leq \ell \leq 3n/4$.

That is pivot is *approximately* in the middle of \mathbf{A}

Then $n/4 \leq |\mathbf{A}_{\text{less}}| \leq 3n/4$ and $n/4 \leq |\mathbf{A}_{\text{greater}}| \leq 3n/4$. If we apply recursion,

$$T(n) \leq T(3n/4) + O(n)$$

Implies $T(n) = O(n)$!

How do we find such a pivot?

A Better Pivot

Suppose pivot is the ℓ th smallest element where $n/4 \leq \ell \leq 3n/4$.

That is pivot is *approximately* in the middle of \mathbf{A}

Then $n/4 \leq |\mathbf{A}_{\text{less}}| \leq 3n/4$ and $n/4 \leq |\mathbf{A}_{\text{greater}}| \leq 3n/4$. If we apply recursion,

$$T(n) \leq T(3n/4) + O(n)$$

Implies $T(n) = O(n)$!

How do we find such a pivot? Randomly?

A Better Pivot

Suppose pivot is the ℓ th smallest element where $n/4 \leq \ell \leq 3n/4$.

That is pivot is *approximately* in the middle of \mathbf{A}

Then $n/4 \leq |\mathbf{A}_{\text{less}}| \leq 3n/4$ and $n/4 \leq |\mathbf{A}_{\text{greater}}| \leq 3n/4$. If we apply recursion,

$$T(n) \leq T(3n/4) + O(n)$$

Implies $T(n) = O(n)$!

How do we find such a pivot? Randomly? In fact works!

Analysis a little bit later.

A Better Pivot

Suppose pivot is the ℓ th smallest element where $n/4 \leq \ell \leq 3n/4$.

That is pivot is *approximately* in the middle of \mathbf{A}

Then $n/4 \leq |\mathbf{A}_{\text{less}}| \leq 3n/4$ and $n/4 \leq |\mathbf{A}_{\text{greater}}| \leq 3n/4$. If we apply recursion,

$$T(n) \leq T(3n/4) + O(n)$$

Implies $T(n) = O(n)$!

How do we find such a pivot? Randomly? In fact works!
Analysis a little bit later.

Can we choose pivot deterministically?

Divide and Conquer Approach

A game of medians

Idea

- 1 Break input \mathbf{A} into many subarrays: $\mathbf{L}_1, \dots, \mathbf{L}_k$.
- 2 Find median \mathbf{m}_i in each subarray \mathbf{L}_i .
- 3 Find the median \mathbf{x} of the medians $\mathbf{m}_1, \dots, \mathbf{m}_k$.
- 4 Intuition: The median \mathbf{x} should be close to being a good median of all the numbers in \mathbf{A} .
- 5 Use \mathbf{x} as pivot in previous algorithm.

But we have to be...

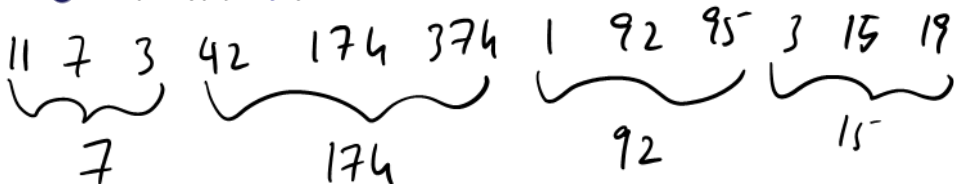
More specific...

- 1 Size of each group?
- 2 How to find median of medians?

Choosing the pivot

A clash of medians

- 1 Partition array **A** into $\lceil n/5 \rceil$ lists of **5** items each.
 $L_1 = \{A[1], A[2], \dots, A[5]\}$, $L_2 = \{A[6], \dots, A[10]\}$, \dots ,
 $L_i = \{A[5i + 1], \dots, A[5i + 5]\}$, \dots ,
 $L_{\lceil n/5 \rceil} = \{A[5\lceil n/5 \rceil - 4], \dots, A[n]\}$.
- 2 For each **i** find median **b_i** of **L_i** using brute-force in **O(1)** time.
Total **O(n)** time
- 3 Let **B** = $\{b_1, b_2, \dots, b_{\lceil n/5 \rceil}\}$
- 4 Find median **b** of **B**



Choosing the pivot

A clash of medians

- 1 Partition array \mathbf{A} into $\lceil n/5 \rceil$ lists of 5 items each.
 $L_1 = \{A[1], A[2], \dots, A[5]\}$, $L_2 = \{A[6], \dots, A[10]\}$, \dots ,
 $L_i = \{A[5i + 1], \dots, A[5i + 5]\}$, \dots ,
 $L_{\lceil n/5 \rceil} = \{A[5\lceil n/5 \rceil - 4], \dots, A[n]\}$.
- 2 For each i find median b_i of L_i using brute-force in $O(1)$ time.
Total $O(n)$ time
- 3 Let $\mathbf{B} = \{b_1, b_2, \dots, b_{\lceil n/5 \rceil}\}$
- 4 Find median b of \mathbf{B}

Lemma

Median of \mathbf{B} is an approximate median of \mathbf{A} . That is, if b is used a pivot to partition \mathbf{A} , then $|A_{less}| \leq 7n/10 + 6$ and $|A_{greater}| \leq 7n/10 + 6$.

Algorithm for Selection

A storm of medians

select(**A**, **j**):

Form lists $L_1, L_2, \dots, L_{\lceil n/5 \rceil}$ where $L_i = \{A[5i - 4], \dots, A[5i]\}$

Find median b_i of each L_i using brute-force

Find median **b** of $B = \{b_1, b_2, \dots, b_{\lceil n/5 \rceil}\}$

Partition **A** into A_{less} and A_{greater} using **b** as pivot

if ($|A_{\text{less}}| = j$) **return b**

else if ($|A_{\text{less}}| > j$)

return select(A_{less} , **j**)

else

return select(A_{greater} , $j - |A_{\text{less}}|$)

Algorithm for Selection

A storm of medians

select(A , j):

Form lists $L_1, L_2, \dots, L_{\lceil n/5 \rceil}$ where $L_i = \{A[5i - 4], \dots, A[5i]\}$

Find median b_i of each L_i using brute-force

Find median b of $B = \{b_1, b_2, \dots, b_{\lceil n/5 \rceil}\}$

Partition A into A_{less} and A_{greater} using b as pivot

if ($|A_{\text{less}}| = j$) **return** b

else if ($|A_{\text{less}}| > j$)

return **select**(A_{less} , j)

else

return **select**(A_{greater} , $j - |A_{\text{less}}|$)

How do we find median of B ?

Algorithm for Selection

A storm of medians

select(A , j):

Form lists $L_1, L_2, \dots, L_{\lceil n/5 \rceil}$ where $L_i = \{A[5i - 4], \dots, A[5i]\}$

Find median b_i of each L_i using brute-force

Find median b of $B = \{b_1, b_2, \dots, b_{\lceil n/5 \rceil}\}$

Partition A into A_{less} and A_{greater} using b as pivot

if ($|A_{\text{less}}| = j$) **return** b

else if ($|A_{\text{less}}| > j$)

return **select**(A_{less} , j)

else

return **select**(A_{greater} , $j - |A_{\text{less}}|$)

How do we find median of B ? Recursively!

Algorithm for Selection

A storm of medians

select(**A**, **j**):

Form lists $L_1, L_2, \dots, L_{\lceil n/5 \rceil}$ where $L_i = \{A[5i - 4], \dots, A[5i]\}$

Find median b_i of each L_i using brute-force

$B = [b_1, b_2, \dots, b_{\lceil n/5 \rceil}]$

$b = \text{select}(B, \lceil n/10 \rceil)$

Partition **A** into A_{less} and A_{greater} using **b** as pivot

if ($|A_{\text{less}}| = j$) **return** **b**

else if ($|A_{\text{less}}| > j$)

return **select**(A_{less} , **j**)

else

return **select**(A_{greater} , $j - |A_{\text{less}}|$)

Running time of deterministic median selection

A dance with recurrences

$$T(n) = T(\lceil n/5 \rceil) + \max\{T(|A_{\text{less}}|), T(|A_{\text{greater}}|)\} + O(n)$$

Running time of deterministic median selection

A dance with recurrences

$$T(n) = T(\lceil n/5 \rceil) + \max\{T(|A_{\text{less}}|), T(|A_{\text{greater}}|)\} + O(n)$$

From Lemma,

$$T(n) \leq T(\lceil n/5 \rceil) + T(\lfloor 7n/10 + 6 \rfloor) + O(n)$$

and

$$T(n) = O(1) \quad n < 10$$

Running time of deterministic median selection

A dance with recurrences

$$T(n) = T(\lceil n/5 \rceil) + \max\{T(|A_{\text{less}}|), T(|A_{\text{greater}}|)\} + O(n)$$

From Lemma,

$$T(n) \leq T(\lceil n/5 \rceil) + T(\lfloor 7n/10 + 6 \rfloor) + O(n)$$

and

$$T(n) = O(1) \quad n < 10$$

Exercise: show that $T(n) = O(n)$

Median of Medians: Proof of Lemma

Proposition

There are at least $3n/10 - 6$ elements greater than the median of medians b .

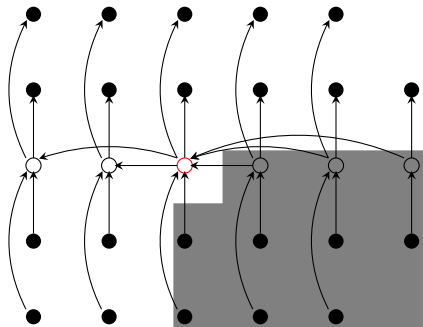


Figure : Shaded elements are all greater than b

Median of Medians: Proof of Lemma

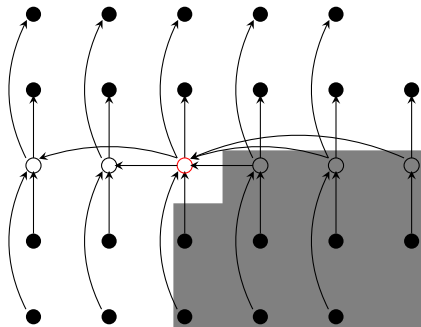


Figure : Shaded elements are all greater than b

Proposition

There are at least $3n/10 - 6$ elements greater than the median of medians b .

Proof.

At least half of the $\lceil n/5 \rceil$ groups have at least 3 elements larger than b , except for last group and the group containing b . Hence number of elements greater than b is:

$$3(\lceil (1/2)\lceil n/5 \rceil \rceil - 2) \geq 3n/10 - 6$$

□

Median of Medians: Proof of Lemma

Proposition

There are at least $3n/10 - 6$ elements greater than the median of medians b .

Corollary

$$|A_{\text{less}}| \leq 7n/10 + 6.$$

Via symmetric argument,

Corollary

$$|A_{\text{greater}}| \leq 7n/10 + 6.$$

Questions to ponder

- 1 Why did we choose lists of size **5**? Will lists of size **3** work?
- 2 Write a recurrence to analyze the algorithm's running time if we choose a list of size **k**.

Median of Medians Algorithm

Due to:

M. Blum, R. Floyd, D. Knuth, V. Pratt, R. Rivest, and R. Tarjan.

“Time bounds for selection”.

Journal of Computer System Sciences (JCSS), 1973.

Median of Medians Algorithm

Due to:

M. Blum, R. Floyd, D. Knuth, V. Pratt, R. Rivest, and R. Tarjan.

“Time bounds for selection”.

Journal of Computer System Sciences (JCSS), 1973.

How many Turing Award winners in the author list?

Median of Medians Algorithm

Due to:

M. Blum, R. Floyd, D. Knuth, V. Pratt, R. Rivest, and R. Tarjan.

“Time bounds for selection”.

Journal of Computer System Sciences (JCSS), 1973.

How many Turing Award winners in the author list?

All except Vaughn Pratt!

Part III

Exponentiation, Binary Search

Exponentiation

Input Two numbers: **a** and integer **n** ≥ 0

Goal Compute **aⁿ**

Exponentiation

Input Two numbers: **a** and integer **n** ≥ 0

Goal Compute **aⁿ**

Obvious algorithm:

```
SlowPow(a,n):  
  x = 1;  
  for i = 1 to n do  
    x = x*a  
  Output x
```

O(n) multiplications.

How many bits...

Let $a > 1$ and $n > 1$ be two integer numbers. Representing a^n in base 2 requires

- (A) $O(\log a + \log n)$ bits.
- (B) $O(n \log a)$ bits.
- (C) $O(a \log n)$ bits.
- (D) $O(\log a \log n)$ bits.
- (E) $O((\log a)^{\log n})$ bits.

Fast Exponentiation

Observation: $a^n = a^{\lfloor n/2 \rfloor} a^{\lceil n/2 \rceil} = a^{\lfloor n/2 \rfloor} a^{\lfloor n/2 \rfloor} a^{\lceil n/2 \rceil - \lfloor n/2 \rfloor}$.

Fast Exponentiation

Observation: $a^n = a^{\lfloor n/2 \rfloor} a^{\lceil n/2 \rceil} = a^{\lfloor n/2 \rfloor} a^{\lfloor n/2 \rfloor} a^{\lceil n/2 \rceil - \lfloor n/2 \rfloor}$.

```
FastPow(a, n):  
  if (n = 0) return 1  
  x = FastPow(a,  $\lfloor n/2 \rfloor$ )  
  x = x * x  
  if (n is odd) then  
    x = x * a  
  return x
```

Fast Exponentiation

Observation: $a^n = a^{\lfloor n/2 \rfloor} a^{\lceil n/2 \rceil} = a^{\lfloor n/2 \rfloor} a^{\lfloor n/2 \rfloor} a^{\lceil n/2 \rceil - \lfloor n/2 \rfloor}$.

```
FastPow(a, n):  
  if (n = 0) return 1  
  x = FastPow(a,  $\lfloor n/2 \rfloor$ )  
  x = x * x  
  if (n is odd) then  
    x = x * a  
  return x
```

T(n): number of multiplications for **n**

Fast Exponentiation

Observation: $a^n = a^{\lfloor n/2 \rfloor} a^{\lceil n/2 \rceil} = a^{\lfloor n/2 \rfloor} a^{\lfloor n/2 \rfloor} a^{\lceil n/2 \rceil - \lfloor n/2 \rfloor}$.

```
FastPow(a, n):  
  if (n = 0) return 1  
  x = FastPow(a,  $\lfloor n/2 \rfloor$ )  
  x = x * x  
  if (n is odd) then  
    x = x * a  
  return x
```

T(n): number of multiplications for **n**

$$T(n) \leq T(\lfloor n/2 \rfloor) + 2$$

T(n) =

Fast Exponentiation

Observation: $a^n = a^{\lfloor n/2 \rfloor} a^{\lceil n/2 \rceil} = a^{\lfloor n/2 \rfloor} a^{\lfloor n/2 \rfloor} a^{\lceil n/2 \rceil - \lfloor n/2 \rfloor}$.

```
FastPow(a, n):  
  if (n = 0) return 1  
  x = FastPow(a,  $\lfloor n/2 \rfloor$ )  
  x = x * x  
  if (n is odd) then  
    x = x * a  
  return x
```

T(n): number of multiplications for **n**

$$T(n) \leq T(\lfloor n/2 \rfloor) + 2$$

$$T(n) = \Theta(\log n)$$

Complexity of Exponentiation

Question: Is **SlowPow()** a polynomial time algorithm? **FastPow**?

Complexity of Exponentiation

Question: Is **SlowPow()** a polynomial time algorithm? **FastPow?**

Input size: **$O(\log a + \log n)$**

Complexity of Exponentiation

Question: Is **SlowPow()** a polynomial time algorithm? **FastPow?**

Input size: **$O(\log a + \log n)$**

Output size:

Complexity of Exponentiation

Question: Is **SlowPow**() a polynomial time algorithm? **FastPow**?

Input size: **$O(\log a + \log n)$**

Output size: **$O(n \log a)$** .

Not necessarily polynomial in input size!

Both **SlowPow** and **FastPow** are polynomial in output size.

26 mod 7 is?

- (A) 0
- (B) 1
- (C) 3
- (D) 5
- (E) 7

Exponentiation modulo a given number

Exponentiation in applications:

Input Three integers: \mathbf{a} , $\mathbf{n} \geq 0$, $\mathbf{p} \geq 2$ (typically a prime)

Goal Compute $\mathbf{a^n \bmod p}$

Exponentiation modulo a given number

Exponentiation in applications:

Input Three integers: a , $n \geq 0$, $p \geq 2$ (typically a prime)

Goal Compute $a^n \bmod p$

Input size: $\Theta(\log a + \log n + \log p)$

Output size: $O(\log p)$ and hence polynomial in input size.

Exponentiation modulo a given number

Exponentiation in applications:

Input Three integers: a , $n \geq 0$, $p \geq 2$ (typically a prime)

Goal Compute $a^n \bmod p$

Input size: $\Theta(\log a + \log n + \log p)$

Output size: $O(\log p)$ and hence polynomial in input size.

Observation: $xy \bmod p = ((x \bmod p)(y \bmod p)) \bmod p$

Exponentiation modulo a given number

Input Three integers: a , $n \geq 0$, $p \geq 2$ (typically a prime)

Goal Compute $a^n \bmod p$

FastPowMod(a, n, p):

if ($n = 0$) **return** 1

$x = \text{FastPowMod}(a, \lfloor n/2 \rfloor, p)$

$x = x * x \bmod p$

if (n is odd)

$x = x * a \bmod p$

return x

Exponentiation modulo a given number

Input Three integers: a , $n \geq 0$, $p \geq 2$ (typically a prime)

Goal Compute $a^n \bmod p$

FastPowMod(a, n, p):

```
if (n = 0) return 1
x = FastPowMod(a, [n/2], p)
x = x * x mod p
if (n is odd)
    x = x * a mod p
return x
```

FastPowMod is a polynomial time algorithm. **SlowPowMod** is not (why?).

Binary Search in Sorted Arrays

Input Sorted array **A** of **n** numbers and number **x**

Goal Is **x** in **A**?

Binary Search in Sorted Arrays

Input Sorted array **A** of **n** numbers and number **x**

Goal Is **x** in **A**?

BinarySearch(**A**[**a..b**], **x**):

if (**b** - **a** < **0**) **return** NO

mid = **A**[**[(a + b)/2]**]

if (**x** = **mid**) **return** YES

if (**x** < **mid**)

return BinarySearch(**A**[**a..[(a + b)/2] - 1**], **x**)

else

return BinarySearch(**A**[**[(a + b)/2] + 1..b**], **x**)

Binary Search in Sorted Arrays

Input Sorted array **A** of **n** numbers and number **x**

Goal Is **x** in **A**?

BinarySearch(**A**[**a..b**], **x**):

if (**b** - **a** < 0) **return** NO

mid = **A**[$\lfloor(\mathbf{a} + \mathbf{b})/2\rfloor$]

if (**x** = **mid**) **return** YES

if (**x** < **mid**)

return **BinarySearch**(**A**[**a..** $\lfloor(\mathbf{a} + \mathbf{b})/2\rfloor - 1$], **x**)

else

return **BinarySearch**(**A**[$\lfloor(\mathbf{a} + \mathbf{b})/2\rfloor + 1$..**b**], **x**)

Analysis: $T(n) = T(\lfloor n/2 \rfloor) + O(1)$. $T(n) = O(\log n)$.

Observation: After **k** steps, size of array left is $n/2^k$

Another common use of binary search

- 1 **Optimization version:** find solution of best (say minimum) value
- 2 **Decision version:** is there a solution of value at most a given value v ?

Another common use of binary search

- 1 **Optimization version:** find solution of best (say minimum) value
- 2 **Decision version:** is there a solution of value at most a given value v ?

Reduce optimization to decision (may be easier to think about):

- 1 Given instance I compute upper bound $U(I)$ on best value
- 2 Compute lower bound $L(I)$ on best value
- 3 Do binary search on interval $[L(I), U(I)]$ using decision version as black box
- 4 $O(\log(U(I) - L(I)))$ calls to decision version if $U(I), L(I)$ are integers

Example

- 1 **Problem:** shortest paths in a graph.
- 2 **Decision version:** given G with non-negative integer edge lengths, nodes s, t and bound B , is there an $s-t$ path in G of length at most B ?
- 3 **Optimization version:** find the length of a shortest path between s and t in G .

Question: given a black box algorithm for the decision version, can we obtain an algorithm for the optimization version?

Example continued

Question: given a black box algorithm for the decision version, can we obtain an algorithm for the optimization version?

- 1 Let U be maximum edge length in G .
- 2 Minimum edge length is L .
- 3 s - t shortest path length is at most $(n - 1)U$ and at least L .
- 4 Apply binary search on the interval $[L, (n - 1)U]$ via the algorithm for the decision problem.
- 5 $O(\log((n - 1)U - L))$ calls to the decision problem algorithm sufficient. Polynomial in input size.

Question

$\mathbf{G} = (\mathbf{V}, \mathbf{E})$ is a directed graph with non-negative edge lengths; $\ell(\mathbf{e})$ length of edge \mathbf{e} . Want to find cycle \mathbf{C} to minimize $\ell(\mathbf{C})/|\mathbf{C}|$, that is, the average length of the cycle.

Recall discussion question: given λ can reduce checking whether \mathbf{G} has cycle of average length $\leq \lambda$ to negative cycle detection.

Question: Suppose we do binary search using the preceding algorithm to find the minimize the average length of a cycle? What is the search range? How many times do we need to call the algorithm for negative cycle detection?

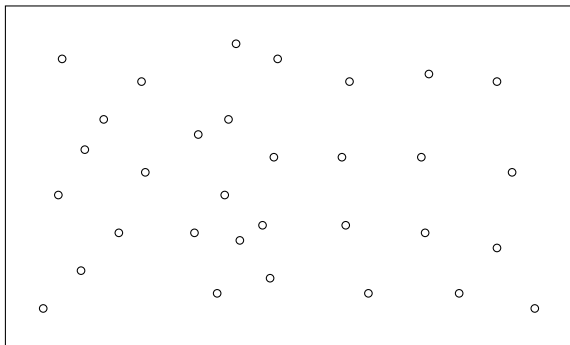
Part IV

Closest Pair

Closest Pair - the problem

Input Given a set S of n points on the plane

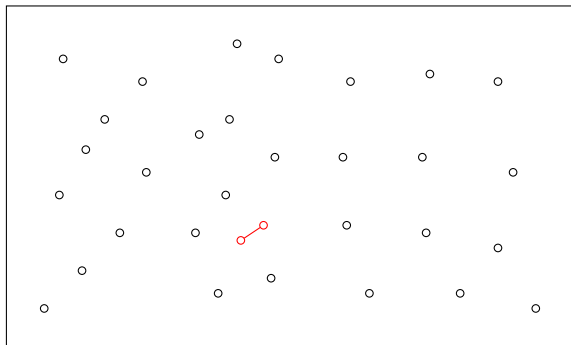
Goal Find $p, q \in S$ such that $d(p, q)$ is minimum



Closest Pair - the problem

Input Given a set S of n points on the plane

Goal Find $p, q \in S$ such that $d(p, q)$ is minimum



Applications

- 1 Basic primitive used in graphics, vision, molecular modelling
- 2 Ideas used in solving nearest neighbor, Voronoi diagrams, Euclidean MST

Algorithm: Brute Force

Algorithm: Brute Force

- 1 Compute distance between every pair of points and find minimum.
- 2 Takes $O(n^2)$ time.

Algorithm: Brute Force

- 1 Compute distance between every pair of points and find minimum.
- 2 Takes $O(n^2)$ time.
- 3 Can we do better?

Closest Pair: 1-d case

Input Given a set \mathbf{S} of \mathbf{n} points on a line

Goal Find $\mathbf{p}, \mathbf{q} \in \mathbf{S}$ such that $\mathbf{d}(\mathbf{p}, \mathbf{q})$ is minimum

Closest Pair: 1-d case

Input Given a set S of n points on a line

Goal Find $p, q \in S$ such that $d(p, q)$ is minimum

Algorithm

- 1 Sort points based on coordinate
- 2 Compute the distance between successive points, keeping track of the closest pair.

Running time $O(n \log n)$

Can we do this in better running time?

Can reduce Distinct Elements Problem (see lecture 1) to this problem in $O(n)$ time. Do you see how?

Generalizing 1-d case

Can we generalize **1**-d algorithm to **2**-d?
Sort according to **x** or **y**-coordinate??

Generalizing 1-d case

Can we generalize **1**-d algorithm to **2**-d?

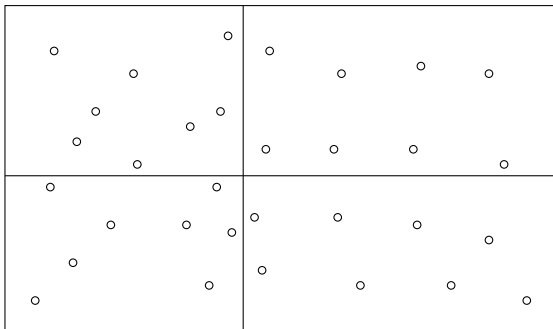
Sort according to **x** or **y**-coordinate??

No easy generalization.

First Attempt

Divide and Conquer I

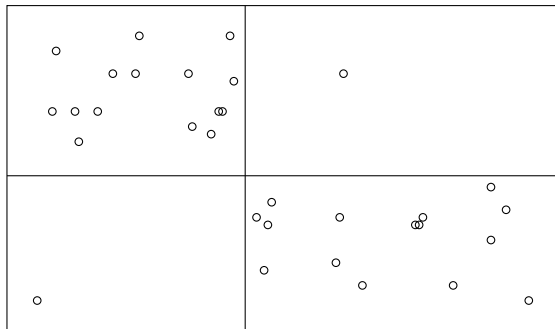
- 1 Partition into 4 quadrants of roughly equal size.
- 2 Find closest pair in each quadrant recursively
- 3 Combine solutions



First Attempt

Divide and Conquer I

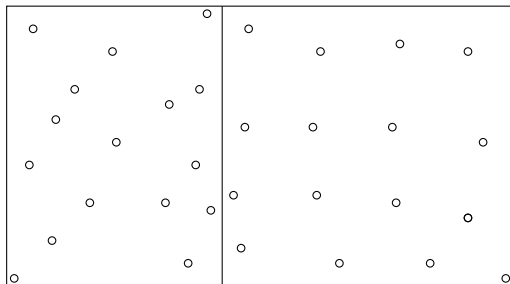
- 1 Partition into 4 quadrants of roughly equal size. Not always!
- 2 Find closest pair in each quadrant recursively
- 3 Combine solutions



New Algorithm

Divide and Conquer II

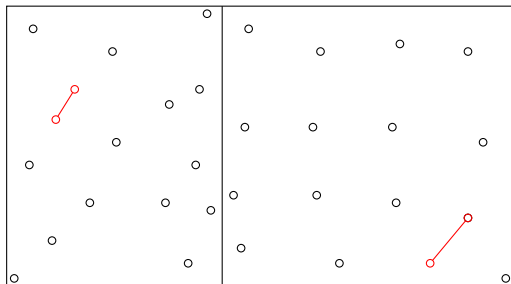
- 1 Divide the set of points into two equal parts via vertical line
- 2 Find closest pair in each half recursively
- 3 Find closest pair with one point in each half
- 4 Return the best pair among the above 3 solutions



New Algorithm

Divide and Conquer II

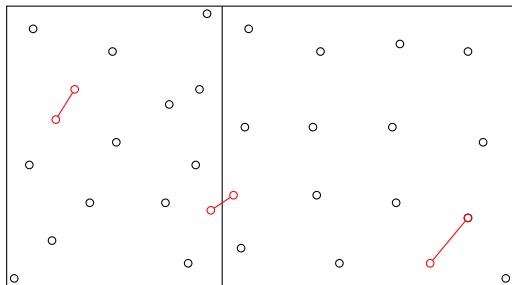
- 1 Divide the set of points into two equal parts via vertical line
- 2 Find closest pair in each half recursively
- 3 Find closest pair with one point in each half
- 4 Return the best pair among the above 3 solutions



New Algorithm

Divide and Conquer II

- 1 Divide the set of points into two equal parts via vertical line
- 2 Find closest pair in each half recursively
- 3 Find closest pair with one point in each half
- 4 Return the best pair among the above 3 solutions



Divide and Conquer II

- 1 Divide the set of points into two equal parts via vertical line
- 2 Find closest pair in each half recursively
- 3 Find closest pair with one point in each half
- 4 Return the best pair among the above 3 solutions

Divide and Conquer II

- 1 Divide the set of points into two equal parts via vertical line
- 2 Find closest pair in each half recursively
- 3 Find closest pair with one point in each half
- 4 Return the best pair among the above 3 solutions

Divide and Conquer II

- 1 Divide the set of points into two equal parts via vertical line
 - 2 Find closest pair in each half recursively
 - 3 Find closest pair with one point in each half
 - 4 Return the best pair among the above 3 solutions
-
- 1 Sort points based on **x**-coordinate and pick the median. Time = **$O(n \log n)$**

Divide and Conquer II

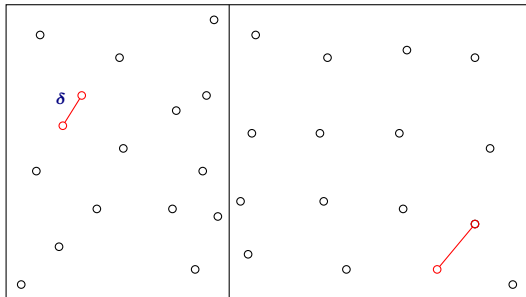
- 1 Divide the set of points into two equal parts via vertical line
 - 2 Find closest pair in each half recursively
 - 3 Find closest pair with one point in each half
 - 4 Return the best pair among the above 3 solutions
-
- 1 Sort points based on **x**-coordinate and pick the median. Time = **$O(n \log n)$**

Divide and Conquer II

- 1 Divide the set of points into two equal parts via vertical line
 - 2 Find closest pair in each half recursively
 - 3 Find closest pair with one point in each half
 - 4 Return the best pair among the above 3 solutions
-
- 1 Sort points based on x -coordinate and pick the median. Time = $O(n \log n)$
 - 2 How to find closest pair with points in different halves? $O(n^2)$ is trivial. Better?

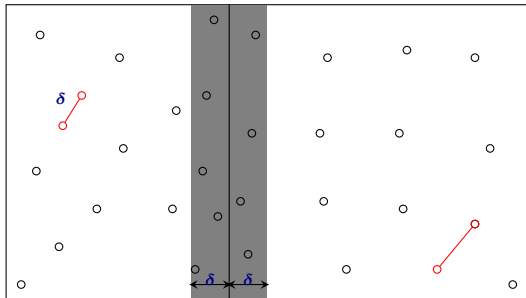
Combining Partial Solutions

- 1 Does it take $O(n^2)$ to combine solutions?
- 2 Let δ be the distance between closest pairs, where both points belong to the same half.



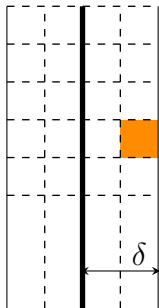
Combining Partial Solutions

- 1 Let δ be the distance between closest pairs, where both points belong to the same half.
- 2 Need to consider points within δ of dividing line



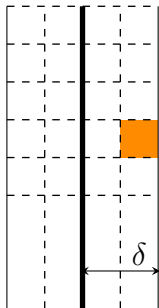
Sparsity of Band XXX

Divide the band into square boxes of size $\delta/2$



Sparsity of Band XXX

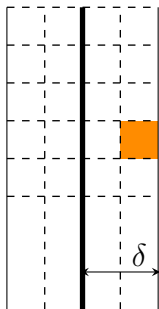
Divide the band into square boxes of size $\delta/2$



Lemma

Each box has at most one point

Sparsity of Band XXX



Divide the band into square boxes of size $\delta/2$

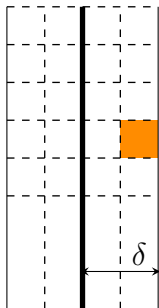
Lemma

Each box has at most one point

Proof.

If not, then there are a pair of points (both belonging to one half) that are at most $\sqrt{2}\delta/2 < \delta$ apart! □

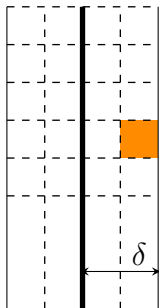
Searching within the Band



Lemma

Suppose \mathbf{a}, \mathbf{b} are both in the band
 $d(\mathbf{a}, \mathbf{b}) < \delta$ then \mathbf{a}, \mathbf{b} have at most two rows
of boxes between them.

Searching within the Band



Lemma

Suppose \mathbf{a}, \mathbf{b} are both in the band
 $d(\mathbf{a}, \mathbf{b}) < \delta$ then \mathbf{a}, \mathbf{b} have at most two rows
of boxes between them.

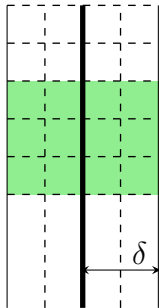
Proof.

Each row of boxes has height $\delta/2$. If more
than two rows then $d(\mathbf{a}, \mathbf{b}) > 2 \cdot \delta/2!$ \square

Searching within the Band

Corollary

Order points according to their y -coordinate. If \mathbf{p}, \mathbf{q} are such that $d(\mathbf{p}, \mathbf{q}) < \delta$ then \mathbf{p} and \mathbf{q} are within **11** positions in the sorted list.



Proof.

- 1 ≤ 2 points between them if \mathbf{p} and \mathbf{q} in same row.
- 2 ≤ 6 points between them if \mathbf{p} and \mathbf{q} in two consecutive rows.
- 3 ≤ 10 points between if \mathbf{p} and \mathbf{q} one row apart.
- 4 \implies More than ten points between them in the sorted y order than \mathbf{p} and \mathbf{q} are more than two rows apart.
- 5 $\implies d(\mathbf{p}, \mathbf{q}) > \delta$. A contradiction. ■

The Algorithm

ClosestPair(**P**):

1. Split **P** into equal-sized sets **P**₁, **P**₂ via vertical line **L**
2. $\delta_1 \leftarrow \text{ClosestPair}(\mathbf{P}_1)$.
3. $\delta_2 \leftarrow \text{ClosestPair}(\mathbf{P}_2)$.
4. $\delta = \min(\delta_1, \delta_2)$
5. Delete points from **P** further than δ from **L**
6. Sort **P** based on **y**-coordinate into an array **A**
7. **for** $i = 1$ to $|\mathbf{A}| - 1$ **do**
 for $j = i + 1$ to $\min\{i + 11, |\mathbf{A}|\}$ **do**
 if $(\text{dist}(\mathbf{A}[i], \mathbf{A}[j]) < \delta)$ update δ and closest pair

The Algorithm

ClosestPair(**P**):

1. Split **P** into equal-sized sets **P**₁, **P**₂ via vertical line **L**
2. $\delta_1 \leftarrow \text{ClosestPair}(\mathbf{P}_1)$.
3. $\delta_2 \leftarrow \text{ClosestPair}(\mathbf{P}_2)$.
4. $\delta = \min(\delta_1, \delta_2)$
5. Delete points from **P** further than δ from **L**
6. Sort **P** based on **y**-coordinate into an array **A**
7. **for** $i = 1$ to $|\mathbf{A}| - 1$ **do**
 for $j = i + 1$ to $\min\{i + 11, |\mathbf{A}|\}$ **do**
 if $(\text{dist}(\mathbf{A}[i], \mathbf{A}[j]) < \delta)$ update δ and closest pair

The Algorithm

ClosestPair(P):

1. Split P into equal-sized sets P_1, P_2 via vertical line L
2. $\delta_1 \leftarrow \text{ClosestPair}(P_1)$.
3. $\delta_2 \leftarrow \text{ClosestPair}(P_2)$.
4. $\delta = \min(\delta_1, \delta_2)$
5. Delete points from P further than δ from L
6. Sort P based on y -coordinate into an array A
7. **for** $i = 1$ to $|A| - 1$ **do**
 for $j = i + 1$ to $\min\{i + 11, |A|\}$ **do**
 if ($\text{dist}(A[i], A[j]) < \delta$) update δ and closest pair

① Step 1, involves sorting and scanning. Takes $O(n \log n)$ time.

The Algorithm

ClosestPair(P):

1. Split P into equal-sized sets P_1, P_2 via vertical line L
2. $\delta_1 \leftarrow \text{ClosestPair}(P_1)$.
3. $\delta_2 \leftarrow \text{ClosestPair}(P_2)$.
4. $\delta = \min(\delta_1, \delta_2)$
5. Delete points from P further than δ from L
6. Sort P based on y -coordinate into an array A
7. for $i = 1$ to $|A| - 1$ do
 for $j = i + 1$ to $\min\{i + 11, |A|\}$ do
 if ($\text{dist}(A[i], A[j]) < \delta$) update δ and closest pair

- 1 Step 1, involves sorting and scanning. Takes $O(n \log n)$ time.
- 2 Step 5 takes $O(n)$ time.

The Algorithm

ClosestPair(P):

1. Split P into equal-sized sets P_1, P_2 via vertical line L
2. $\delta_1 \leftarrow \text{ClosestPair}(P_1)$.
3. $\delta_2 \leftarrow \text{ClosestPair}(P_2)$.
4. $\delta = \min(\delta_1, \delta_2)$
5. Delete points from P further than δ from L
6. Sort P based on y -coordinate into an array A
7. for $i = 1$ to $|A| - 1$ do
 for $j = i + 1$ to $\min\{i + 11, |A|\}$ do
 if ($\text{dist}(A[i], A[j]) < \delta$) update δ and closest pair

- 1 Step 1, involves sorting and scanning. Takes $O(n \log n)$ time.
- 2 Step 5 takes $O(n)$ time.

The Algorithm

ClosestPair(P):

1. Split P into equal-sized sets P_1, P_2 via vertical line L
2. $\delta_1 \leftarrow \text{ClosestPair}(P_1)$.
3. $\delta_2 \leftarrow \text{ClosestPair}(P_2)$.
4. $\delta = \min(\delta_1, \delta_2)$
5. Delete points from P further than δ from L
6. Sort P based on y -coordinate into an array A
7. for $i = 1$ to $|A| - 1$ do
 for $j = i + 1$ to $\min\{i + 11, |A|\}$ do
 if ($\text{dist}(A[i], A[j]) < \delta$) update δ and closest pair

- 1 Step 1, involves sorting and scanning. Takes $O(n \log n)$ time.
- 2 Step 5 takes $O(n)$ time.
- 3 Step 6 takes $O(n \log n)$ time

The Algorithm

ClosestPair(P):

1. Split P into equal-sized sets P_1, P_2 via vertical line L
2. $\delta_1 \leftarrow \text{ClosestPair}(P_1)$.
3. $\delta_2 \leftarrow \text{ClosestPair}(P_2)$.
4. $\delta = \min(\delta_1, \delta_2)$
5. Delete points from P further than δ from L
6. Sort P based on y -coordinate into an array A
7. for $i = 1$ to $|A| - 1$ do
 for $j = i + 1$ to $\min\{i + 11, |A|\}$ do
 if ($\text{dist}(A[i], A[j]) < \delta$) update δ and closest pair

- 1 Step 1, involves sorting and scanning. Takes $O(n \log n)$ time.
- 2 Step 5 takes $O(n)$ time.
- 3 Step 6 takes $O(n \log n)$ time

The Algorithm

ClosestPair(P):

1. Split P into equal-sized sets P_1, P_2 via vertical line L
2. $\delta_1 \leftarrow \text{ClosestPair}(P_1)$.
3. $\delta_2 \leftarrow \text{ClosestPair}(P_2)$.
4. $\delta = \min(\delta_1, \delta_2)$
5. Delete points from P further than δ from L
6. Sort P based on y -coordinate into an array A
7. **for** $i = 1$ **to** $|A| - 1$ **do**
 for $j = i + 1$ **to** $\min\{i + 11, |A|\}$ **do**
 if $(\text{dist}(A[i], A[j]) < \delta)$ **update** δ **and** **closest pair**

- 1 Step 1, involves sorting and scanning. Takes $O(n \log n)$ time.
- 2 Step 5 takes $O(n)$ time.
- 3 Step 6 takes $O(n \log n)$ time

The Algorithm

ClosestPair(P):

1. Split P into equal-sized sets P_1, P_2 via vertical line L
2. $\delta_1 \leftarrow \text{ClosestPair}(P_1)$.
3. $\delta_2 \leftarrow \text{ClosestPair}(P_2)$.
4. $\delta = \min(\delta_1, \delta_2)$
5. Delete points from P further than δ from L
6. Sort P based on y -coordinate into an array A
7. **for** $i = 1$ **to** $|A| - 1$ **do**
 for $j = i + 1$ **to** $\min\{i + 11, |A|\}$ **do**
 if $(\text{dist}(A[i], A[j]) < \delta)$ **update** δ **and** **closest pair**

- 1 Step 1, involves sorting and scanning. Takes $O(n \log n)$ time.
- 2 Step 5 takes $O(n)$ time.
- 3 Step 6 takes $O(n \log n)$ time
- 4 Step 7 takes $O(n)$ time.

Running Time

The running time of the algorithm is given by

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

Running Time

The running time of the algorithm is given by

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

Thus, $T(n) = O(n \log^2 n)$.

Running Time

The running time of the algorithm is given by

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

Thus, $T(n) = O(n \log^2 n)$.

Improved Algorithm

Avoid repeated sorting of points in band: two options

- 1 Sort all points by **y**-coordinate and store the list. In conquer step use this to avoid sorting
- 2 Each recursive call returns a list of points sorted by their **y**-coordinates. Merge in conquer step in linear time.

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

Takeaway Points

- 1 Recursion tree method and guess and verify are the most reliable methods to analyze recursions in algorithms.
- 2 Recursive algorithms naturally lead to recurrences.
- 3 Some times one can look for certain type of recursive algorithms (reverse engineering) by understanding recurrences and their behavior.