COMP 6651
Algorithm Design Techniques

Lecturer: Thomas Fevens

Department of Computer Science and Software Engineering, Concordia U
thomas.fevens@concordia.ca

Table of Contents

Problem: Given an array A of n elements, find ith smallest element in it.

Ideas:

1. Find smallest, next smallest, etc.
   This is the idea behind Heapsort. Here, we would stop when finding the ith smallest value, instead of at the largest value.
   Heapsort is an example of a comparison-based sorting algorithm whose run-time is $\Theta(n \log n)$ in the worst case which is not a divide and conquer algorithm.
2. Sort elements, take $A[i]$.
3. Consider a direct divide and conquer approach for an algorithm.
   We'll develop this idea in the next few slides.

Median and Order Statistics

The ith **order statistic** of a set of n elements is the ith smallest element

The **minimum** is the first order statistic ($i = 1$)

The **maximum** is the nth order statistic ($i = n$)

A **median** is the "halfway point" of a set.

n odd: median is unique, at the $(n+1)/2$th element

n even: **lower median** is $(n/2)$th element,
upper median is $(n/2+1)$th element.

We mean lower median when we use the phrase "the median"

Selection Problem

The **Selection Problem:**

Input: $A[1..n]$ - array of n integers
Output: $x \in A$ larger than exactly $i - 1$ elements in it,
= find ith **order statistic** of A.

Sorting all elements on order to find ith order statistics needs $O(n \log n)$ time.

We will improve on this time complexity. First, we will consider finding the minimum or maximum value.

We can easily obtain an upper bound of $n - 1$ comparisons for finding the minimum of a set of n elements.

- Examine each element in turn and keep track of the smallest one.
- This is the best we can do, because each element, except the minimum, must be compared to a smaller element at least once.

The following pseudocode finds the minimum element in array $A[1..n]$:

Minimum(A, n)
  $min \leftarrow A[1]$
  for $i \leftarrow 2, n$ **do**
    if $min > A[i]$ **then**
      $min \leftarrow A[i]$
  return min

The maximum can be found in exactly the same way by replacing the $>$ with $<$ in the algorithm.

Simultaneous Minimum and Maximum (§9.1)

Some applications need both the minimum and maximum of a set of elements.

- For example, a graphics program may need to scale a set of (x, y) data to fit onto a rectangular display. To do so, the program must first find the minimum and maximum of each coordinate.

A simple algorithm to find the minimum and maximum is to find each one independently. There will be $n - 1$ comparisons for the minimum and $n - 1$ comparisons for the maximum, for a total of $2n - 2$ comparisons.

This will result in $\Theta(n)$ time.

In fact, at most $3\lfloor n/2 \rfloor$ comparisons suffice to find both the minimum and maximum:

- Maintain the minimum and maximum of elements seen so far.
- Don't compare each element to the minimum and maximum separately.
- Process elements in pairs.
- Compare the elements of a pair to each other.
- Then compare the larger element to the maximum so far, and compare the smaller element to the minimum so far.

This leads to only 3 comparisons for every 2 elements.

Setting up the initial values for the min and max depends on whether n is odd or even.

- If n is even, compare the first two elements and assign the larger to max and the smaller to min. Then process the rest of the elements in pairs.
- If n is odd, set both min and max to the first element. Then process the rest of the elements in pairs.

if $A[i] > A[i+1]$ **then**
  if $max < A[i]$ **then**
    $max \leftarrow A[i]$
  if $min > A[i+1]$ **then**
    $min \leftarrow A[i+1]$
else
  if $max < A[i+1]$ **then**
    $max \leftarrow A[i+1]$
  if $min > A[i]$ **then**
    $min \leftarrow A[i]$

- If n is even, do 1 initial comparison and then $3(n-2)/2$ more comparisons.

$$\begin{aligned}
\text{\# of comparisons} &= \frac{3(n-2)}{2} + 1 \\
&= \frac{3n-6}{2} + 1 \\
&= \frac{3n}{2} - 3 + 1 \\
&= \frac{3n}{2} - 2.
\end{aligned}$$

- If n is odd, do $3(n-1)/2 = 3\lfloor n/2 \rfloor$ comparisons.

In either case, the maximum number of comparisons is $\leq 3\lfloor n/2 \rfloor$.

Selection in Expected Linear Time (§9.2)

There is a linear expected time algorithm for the selection problem. It uses the *divide and conquer* paradigm.

The function **Randomized-Select** uses **Randomized-Partition** from the Quicksort algorithm from last week. **Randomized-Select** differs from Quicksort in that it recurses on only one side of the partition.

Initially, call
Randomized-Select$(A, 1, n, i)$

Randomized-Select(A, p, r, i)
\\\\ Find ith smallest element of A[p..r]
  if $p = r$ **then**
    return $A[p]$
  $q \leftarrow$ **Randomized-Partition**(A, p, r)
  $k \leftarrow q - p + 1$ \\\\ number of values in $[p..q]$
  if $i = k$ **then**
    return $A[q]$
  else if $i < k$ **then**
    return Randomized-Select$(A, p, q - 1, i)$
  else
    return Randomized-Select$(A, q + 1, r, i - k)$

After the call to **Randomized-Partition**, the array is partitioned into two subarrays $A[p..q-1]$ and $A[q+1..r]$, along with the pivot element at $A[q]$.

- The elements of the subarray $A[p..q-1]$ are all $\leq A[q]$
- The elements of the subarray $A[q+1..r]$ are all $> A[q]$
- The pivot element is the kth element of the array $A[p..r]$, where $k = q - p + 1$
- If the pivot element is the ith smallest element (i.e., $i = k$), return $A[q]$
- Otherwise, recursively select the appropriate element in **one of the two** partitions

The action of **Randomize-Select** as successive partitionings narrow $A[p..r]$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	p	r	i	partitioning
$A^{(0)}$	6	19	4	12	14	9	15	7	8	11	3	13	2	5	10	1	15	5	
																			1
$A^{(1)}$	6	4	12	10	9	7	8	11	3	13	2	5	14	19	15	1	12	5	
																			2
$A^{(2)}$	3	2	4	10	9	7	8	11	6	13	5	12	14	19	15	4	12	2	
																			3
$A^{(3)}$	3	2	4	10	9	7	8	11	6	12	5	13	14	19	15	4	11	2	
																			4
$A^{(4)}$	3	2	4	5	6	7	8	11	9	12	10	13	14	19	15	4	5	2	
																			5
$A^{(5)}$	3	2	4	5	6	7	8	11	9	12	10	13	14	19	15	5	5	1	

The worst-case of **Randomize-Select** is $\Theta(n^2)$.

Theorem

The expected run-time $T(n)$ of **Randomize-Select** is linear, i.e., $T(n) = \Theta(n)$.

Proof

Randomize-Partition returns any value with the same probability as the pivot.

For any k the probability of getting a subarray with k elements, $1 \leq k \leq n$ is $1/n$.

Indicator (Bernoulli) Random variable X_k:

$X_k = \mathrm{I}\{\text{subarray } A[p..q] \text{ has exactly } k \text{ elements}\}$

$E[X_k] = 1/n$

When $X_k = 1$ we recurse either on subarray of size $k-1$ or $n-k$.

We assume that we have to look in the longer of the two subarrays.

$$T(n) \leq \sum_{k=1}^{n} X_k \cdot (T(max(k-1, n-k)) + O(n))$$

$$T(n) \leq \sum_{k=1}^{n} (X_k \cdot T(max(k-1, n-k)) + O(n))$$

$$E[T(n)] \leq E[\sum_{k=1}^{n} X_k \cdot T(max(k-1, n-k)) + O(n)]$$

$$E[T(n)] \leq \sum_{k=1}^{n} E[X_k \cdot T(max(k-1, n-k))] + O(n)$$

$$E[T(n)] \leq \sum_{k=1}^{n} E[X_k] \cdot E[T(max(k-1, n-k))] + O(n)$$

$$E[T(n)] \leq \sum_{k=1}^{n} \frac{1}{n} \cdot E[T(max(k-1, n-k))] + O(n)$$

$$\begin{aligned} k = 1 + i &\quad : \quad max(k-1, n-k) = n - i - 1 \\ k = n - i &\quad : \quad max(k-1, n-k) = n - i - 1 \end{aligned}$$

$\Rightarrow$ max values are same for $k \in [1..\lfloor n/2 \rfloor - 1]$ and $k \in [\lfloor n/2 \rfloor..n]$

$$E[T(n)] \leq \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n} E[T(k)] + O(n)$$

Solve by substitution: assume $T(n) \leq cn$

$$E[T(n)] \leq \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n} ck + an$$

$$E[T(n)] \leq \frac{2c}{n}(\sum_{k=1}^{n} k - \sum_{k=1}^{\lfloor n/2 \rfloor - 1} k) + an$$
$$E[T(n)] \leq \frac{2c}{n}((n-1)n/2 - (\lfloor n/2 \rfloor - 1)\lfloor n/2 \rfloor/2) + an$$
$$E[T(n)] \leq \frac{2c}{n}((n-1)n/2 - (n/2-2)(n/2-1)/2) + an$$
$$E[T(n)] \leq \frac{c}{n}(3n^2/4 + n/2 - 2) + an$$
$$E[T(n)] \leq 3cn/4 + c/2 - 2/n + an$$
$$E[T(n)] \leq cn - (cn/4 - c/2 - an)$$

We need $(cn/4 - c/2 - an) \geq 0$ for sufficiently large n,
or $n(c/4 - a) \geq c/2$ for sufficiently large n.
Choose $c > 4a$, and we have $n \geq 2c/(c - 4a)$.
All is fine if $T(n) = O(1)$ for small values of n.

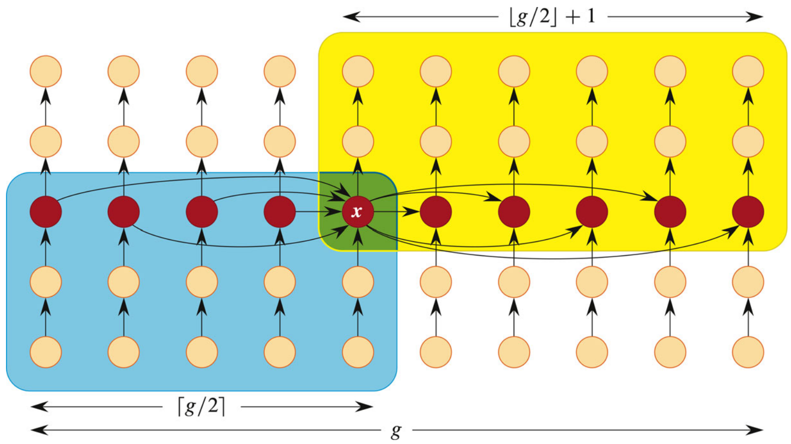
Selection in Worst-case Linear Time (§9.3)

Can we make a **Select** algorithm to be linear in the worst case?

Make the split “good” in all cases by using all the time a provably “good” pivot.

1. Divide n elements into $\lfloor n/5 \rfloor$ groups of 5 elements, and one group that can have less than 5 elements.
2. Find the median of each group.
3. Use **Select** recursively on the medians to find the median x of the $\lceil n/5 \rceil$ median values.
4. Partition the entire input $A[p..r]$ around x, with the index for x being q, as before.
5. Continue as in the **Select** algorithm recursively.

How good is the partition around x?



There are at least $3\left(\left\lceil \frac{g}{2} \right\rceil\right) = 3\left(\left\lceil \frac{1}{2} \left\lceil \frac{n}{5} \right\rceil \right\rceil\right) \geq 3n/10$ elements greater than x.

There are at least $3n/10$ elements less than x.

Analysis of Select

The code contains three recursive calls, of which at most two execute. The first recursive call to find the median of the medians always executes, taking $T(g) \leq T(\lceil n/2 \rceil)$. At most one of the other two recursive calls executes.

$$T(n) \leq \Theta(1) \text{ if } n \leq c$$

$$T(n) \leq T(\lceil n/5 \rceil) + T(7n/10) + O(n) \text{ if } n > c$$

We can solve the recurrence (by substitution for suitably large constant c) and get that

$$T(n) = cn$$

in the worst case.

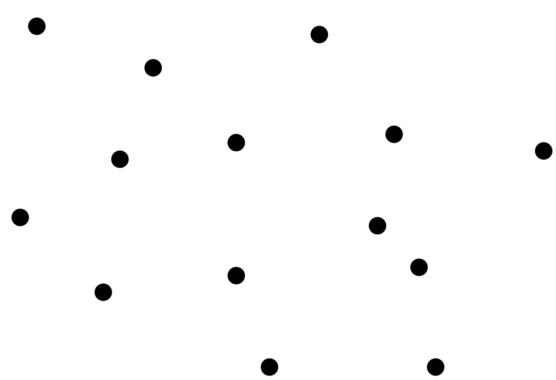
Notice that the divide-and-conquer used here breaks **Select** the ith problem into a median problem + smaller **Select** the ith problem.

Selection versus Sorting

- Sorting requires $\Omega(n \lg n)$ time in the comparison model
- Sorting algorithms that run in linear time need to make assumptions about their input
- Linear-time *Selection* algorithms do not require any assumptions about their input
- Linear-time selection algorithms solve the selection problem **without** sorting and therefore are not subject to the $\Omega(n \lg n)$ lower bound

Finding a closest pair of points
(§33.4 of 3rd Ed. of CLRS; not in 4th Ed.)

Given a set of n points Q in the plane, find a pair of points that is closest.



This is an example of a problem in **Computational Geometry**.

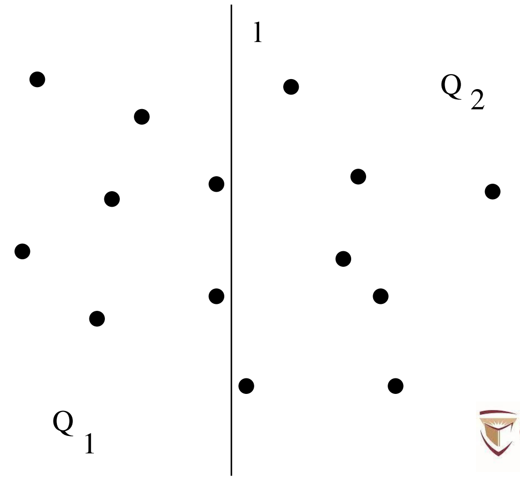
n points form $n(n-1)/2$ different pairs of points.

An algorithm calculating distances between all pairs needs time $O(n^2)$.
Can we do better using Divide and Conquer?

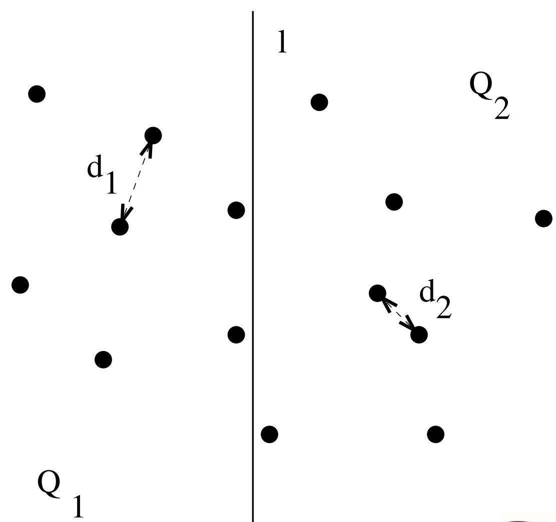
Basic idea:
If there are at most 3 points, solve the problem by calculating all pairs distances, otherwise:

Divide:
Given Q, find a vertical line l that bisects Q into 2 subsets Q_1, Q_2 of the same size.

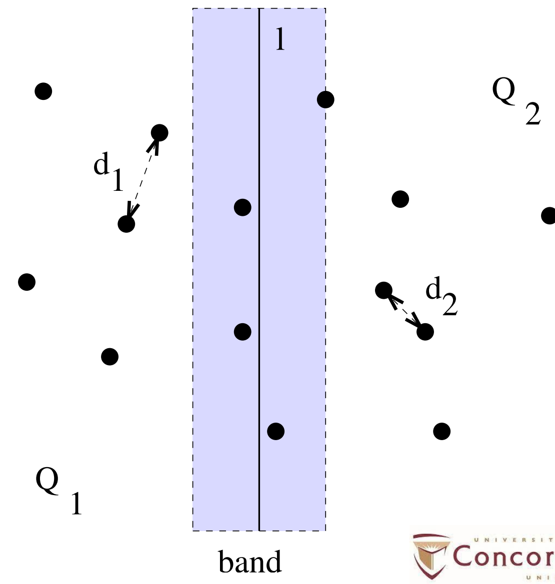
How? Select median in $O(n)$ time of the x-values.



Conquer (recursively):
a) Find the closest pair in Q_1 ,
b) Find the closest pair in Q_2,



Combine:
Let δ be the closest distance in Q_1 and Q_2. Inspect the band around l of size 2δ and see if any pair there is at distance $< \delta$. If yes, that pair is the solution, otherwise it is a solution of Q_1 or Q_2



We have to investigate whether the idea can be used to give an algorithm that is better than the trivial $O(n^2)$ algorithm.

We want to show that we can achieve a recurrence

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

which gives $T(n) = O(n \log n)$

We need to show that the *divide* and *combine* are both $O(n)$.

Before we start the algorithm. create from Q arrays

X that is sorted by x coordinate and
Y that is sorted by y coordinate.

X makes bisection very simple, and
Y is uses to make combine $O(n)$.

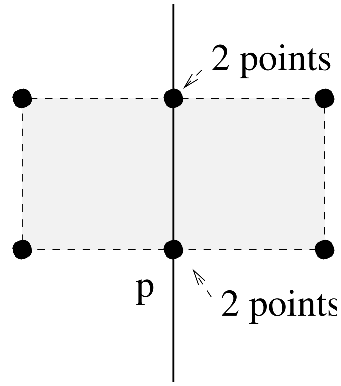
This adds *additive factor* $O(n \log n)$ to the algorithm.

Calculation of the pair of points at closest distance in the band:

1. Create Y' which contains all points of Y in the 2δ band.

2. For each $p \in Y'$ calculate the distance to the 7 points that follow p.

Why at most 7 points? ($\delta \times \delta$ square can contain at most 4 points at distance at least δ).



3. Keep the smallest distance less than δ distance so far.

This needs at most $O(n)$ time.

Time complexity: $T(n) + O(n \log n)$

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

which gives $T(n) = O(n \log n)$

Thus the time complexity is $O(n \log n)$

If we sort the points after each bisection, we would get

Time complexity: $T(n)$ where

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

which gives $T(n) = \Theta(n \log^2 n)$

How many subproblems to use?

In all examples of divide and conquer algorithms, we have seen so far we used a division of a problem into 2 sub-problems.

This is the most common situation.

Division into three is more complicated in most cases,

division into 4 subproblems is equivalent to applying two steps in division into two subproblems.

We must use what is most convenient from the point of view of the problem.