Design and Analysis of Algorithm
Divide and Conquer (I)

1. Introduction of Divide-and-Conquer
2. QuickSort
3. Chip Test
4. Selection Problem
   - Selecting Max and Min
   - Selecting the Second Largest
   - General Selection Problem
5. Closest Pair of Points

Outline

1. Introduction of Divide-and-Conquer
2. QuickSort
3. Chip Test
4. Selection Problem
   - Selecting Max and Min
   - Selecting the Second Largest
   - General Selection Problem
5. Closest Pair of Points

Divide-and-Conquer Paradigm

Divide-and-Conquer strategy solves a problem by:

Divide-and-Conquer Paradigm

Divide-and-Conquer strategy solves a problem by:

1. Divide: break original problem into several *subproblems* with smaller size that can be solved independently

Divide-and-Conquer Paradigm

Divide-and-Conquer strategy solves a problem by:

1. Divide: break original problem into several *subproblems* with smaller size that can be solved independently
2. Conquer: recursively or iteratively solving these subproblems
    - when the subproblems are so small, they are solved outright

Divide-and-Conquer Paradigm

Divide-and-Conquer strategy solves a problem by:

1. Divide: break original problem into several *subproblems* with smaller size that can be solved independently
2. Conquer: recursively or iteratively solving these subproblems
   - when the subproblems are so small, they are solved outright
3. Combine: compose solutions to subproblems into overall solution
   - coordinated by the algorithm's core recursive structure

Why Divide-and-Conquer

Not always, but usually performs better than brute-force algorithm

Why Divide-and-Conquer

Not always, but usually performs better than brute-force algorithm

Most common usage (Example)

- Divide problem of size n into two subproblems of size $n/2$ in linear time
- Solve two subproblems recursively
- Combine two solutions into overall solution in linear time

Why Divide-and-Conquer

Not always, but usually performs better than brute-force algorithm

Most common usage (Example)

- Divide problem of size n into two subproblems of size $n/2$ in linear time
- Solve two subproblems recursively
- Combine two solutions into overall solution in linear time

Brute force: $\Theta(n^2)$ vs. Divide-and-conquer: $\Theta(n \log n)$

Why Divide-and-Conquer

Not always, but usually performs better than brute-force algorithm

Most common usage (Example)

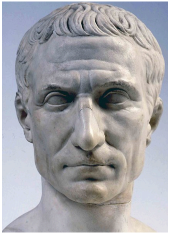
- Divide problem of size n into two subproblems of size $n/2$ in linear time
- Solve two subproblems recursively
- Combine two solutions into overall solution in linear time

Brute force: $\Theta(n^2)$ vs. Divide-and-conquer: $\Theta(n \log n)$

particularly applicable in parallel computing environment (will be more efficient)

Origin of Divide-and-Conquer: Western

Divide and rule (Latin: divide et impera), or divide and conquer, in politics and sociology is gaining and maintaining power by breaking up larger concentrations of power into pieces that individually have less power than the one implementing the strategy.


- The maxim divide et impera has been attributed to Philip II of Macedon. It was utilised by the Roman ruler Julius Caesar and the French emperor Napoleon.

Origin of Divide-and-Conquer: Eastern

> 故用兵之法，十则围之，五则攻之，倍则战之，敌则能分之，...
>
> —《孙子兵法》

Figure: 秦王扫六合时，虎视何雄哉

General Divide-and-Conquer Algorithm

Algorithm 1: Divide-and-Conquer(P)

```
if |P| ≤ s* then Solve(P);                          // direct solve
else divide P into P_1, P_2, ..., P_k;              // divide
for i ← 1 to k do
    y_i ← Divide-and-Conquer(P_i)                   // solve subproblems
end
return Merge(y_1, y_2, ..., y_k)                    // combine answers
```

Complexity of Divide-and-Conquer

Recurrence relation:

$$\begin{cases} T(n) = T(|P_1|) + T(|P_2|) + \cdots + T(|P_k|) + f(n) \\ T(s^*) = C \end{cases}$$

- $P_1, P_2, \ldots, P_k$ are subproblems after dividing
- $f(n)$ is the complexity of dividing subproblems and combining answers of subproblems to answer to the original problem
- C is the complexity of the smallest subproblem of size s^*

Next, we introduce two canonical types of recurrence relations.

Case 1: Subproblems Reduce Size by a Constant

$$T(n) = \sum_{i=1}^{k} a_i T(n-i) + f(n)$$

Solving method

1. Iteration (direct iteration or simplify-then-iteration)
2. Recursive tree

Example. Hanoi tower: $T(n) = 2T(n-1) + 1$

Case 2: Subproblems Reduce Size Linearly

$$T(n) = aT\left(\frac{n}{b}\right) + f(n), h(n) = n^{\log_b a}$$

Solving method: recursion tree, master theorem

$$T(n) = \begin{cases} \Theta(h(n)) & \text{if } f(n) = o(h(n)) \\ \Theta(h(n)\log n) & \text{if } f(n) = \Theta(h(n)) \\ \Theta(f(n)) & \text{if } f(n) = \omega(h(n)) \\ & \wedge \exists\, r < 1 \text{ s.t. } af(n/b) < rf(n) \end{cases}$$

Example 1. Binary search: $W(n) = W(n/2) + 1$

Example 2. Merge sort: $W(n) = 2W(n/2) + (n-1)$

In this section, we illustrate the main idea of divide-and-conquer by several introductory examples.

Hanoi Tower

Algorithm 2: Hanoi(A, C, n) // n disk from A to C

```
Input: A(n), B(0), C(0)
Output: A(0), B(0), C(n)
if n = 1 then move (A, C); //one disk from A to C
else
    Hanoi(A, B, n − 1);
    move (A, C);
    Hanoi(B, C, n − 1)
end
```

Complexity of Hanoi Tower

1. Reduce the original problem to two subproblem of size $n-1$
2. Continue to reduce until the size of subproblem is 1
3. From input size 1 to $n-1$, combine the answers until the size go back to n.

Let $T(n)$ be the complexity of moving n disks: the minimum number of moves required

$$\left.\begin{array}{c} T(n) = 2T(n-1) + 1 \\ T(1) = 1 \end{array}\right\} \Rightarrow T(n) = 2^n - 1$$

There is no worst-case, best-case, average-case distinctions for this problem, since the input only depend on the input size.

Binary Search

Algorithm 3: BinarySearch(A, l, r, x)

Input: sorted $A[l, r]$ in ascending order, target x
Output: j // if $x \in T$, j is the index, else $j = 0$

if $l = r$ **then** //the smallest subproblem
    if $x = A[l]$ **then return** l;
    else return 0;
end
$m \leftarrow \lfloor (l + r)/2 \rfloor$ // m is the middle position;
if $x \leq A[m]$ **then** //compare to median
    BinarySearch(A, l, m, x)
end
else
    BinarySearch$(A, m + 1, r, x)$
end

Complexity of Binary Search

1. Reduce the original problem to a subproblem with half size by comparing x with the median:
    - if $x \leq A[m]$, then $A[l, r] := A[l, m]$, else $A[l, r] := A[m + 1, r]$
2. Repeatedly search T until its size becomes 1, i.e. $l = r$
    - At this point, directly compare x and $A[l]$, return l if equal and "0" otherwise.

Worst-case complexity of binary search

$h(n) = 1$, $f(n) = \Theta(h(n)) \Rightarrow$ master theorem (case 2)

$$\left.\begin{matrix} W(n) = W(\lceil n/2 \rceil) + 1 \\ W(1) = 1 \end{matrix}\right\} \Rightarrow W(n) = \Theta(\log n)$$

Example of MergeSort

Algorithm 4: MergeSort(A, n)

Input: unsorted $A[n]$

Output: sorted $A[n]$ in ascending order

$l \leftarrow 1$, $r \leftarrow n$;
if $l < r$ **then**
    $m \leftarrow \lfloor (l + r)/2 \rfloor$ //partition by half;
    MergeSort(A, l, m) // subproblem 1;
    MergeSort$(A, m + 1, r)$ // subproblem 2;
    Merge$(A[l, m], A[m + 1, r])$ // merge sorted sub-array
end

How to implement Merge *recursively?*

Recursive Merge Algorithm

Algorithm 5: Merge$(A[1,k], B[1,l])$

if $k = 0$ **then return** $B[1,l]$;
if $l = 0$ **then return** $A[1,k]$;
if $A[1] \leq B[1]$ **then return** $A[1] \circ$ Merge$(A[2,k], B[1,l])$;
else return $B[1] \circ$ Merge$(A[1,k], B[2,l])$;

- The Merge procedure does a constant amount of work per recursive call, for a total running time of $O(k + l)$.

Complexity of MergeSort

1. Partition the original problem to 2 subproblem of size $n/2$
2. Continue the partition step until the size of subproblem is 1
3. From input size 1 to $n/2$, merge two neighbored sorted sub-array.
    - The size of sub-array doubles after each merge, until reach the original size.

Assume $n = 2^k$, the worse-case complexity of MergeSort is:

$h(n) = n$, $f(n) = \Theta(h(n)) \Rightarrow$ master theorem (case 2)

$$\left.\begin{matrix} W(n) = 2W(n/2) + n - 1 \\ W(1) = 0 \end{matrix}\right\} \Rightarrow W(n) = \Theta(n \log n)$$

Recap of MergeSort

The dividing of subproblems is already done thanks to data structure of array, all the real work need to be done is merging.

Recap of MergeSort

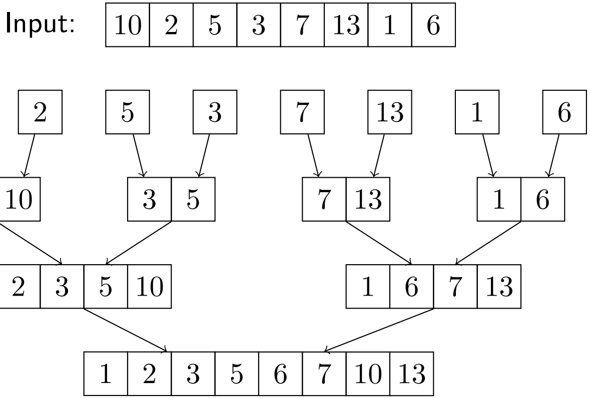
The dividing of subproblems is already done thanks to data structure of array, all the real work need to be done is merging.

This viewpoint suggests MergeSort can be made iterative from singleton arrays to the original array in the bottom-up flavor.

Recap of MergeSort

The dividing of subproblems is already done thanks to data structure of array, all the real work need to be done is merging.

This viewpoint suggests MergeSort can be made iterative from singleton arrays to the original array in the bottom-up flavor.



Recap

We exemplify the features of divide-and-conquer algorithm:

- Divide the original problem to independent subproblems with smaller size
  - the subproblem and the original problem are of the same type
  - when the subproblems are sufficiently small, they can be solved outright
- The algorithm can be solved recursively or iteratively
- Complexity analysis: solving recurrence relation

Outline

1. Introduction of Divide-and-Conquer
2. QuickSort
3. Chip Test
4. Selection Problem
   - Selecting Max and Min
   - Selecting the Second Largest
   - General Selection Problem
5. Closest Pair of Points

Basic Idea

1. Choose the first element x as pivot, partition A into two sub-array:
   - low sub-array A_L: elements less than x
   - high sub-array A_R: elements greater than x
   - x is at the right position
2. recursively sort A_L and A_R, until the size of sub-array is 1

Pseudocode of QuickSort

Algorithm 6: QuickSort(A, l, r)

```
Input: A[l ... r]
Output: sorted A in ascending order
if l = r then return; //reach the smallest case
if l < r then
    k ← Partition(A, l, r);
    A[l] ↔ A[k];
    QuickSort(A, l, k − 1);
    QuickSort(A, k + 1, r);
end
```

Pseudocode of Partition

Algorithm 7: Partition(A, l, r)

$x \leftarrow A[l]$ //set the first element as pivot;
$i \leftarrow l$, $j \leftarrow r + 1$ //initialize left/right pointer;
while *true* **do**
    repeat $j \leftarrow j - 1$ **until** $A[j] \leq x$; //less than x
    repeat $i \leftarrow i + 1$ **until** $A[i] > x$; //greater than x
    if $i < j$ **then** $A[i] \leftrightarrow A[j]$;
    else return j; //cross happen, find the position
end

Demo of Partition

27	99	0	8	13	64	86	16	7	10	88	25	90
	i										j	
27	25	0	8	13	64	86	16	7	10	88	99	90
					i				j			
27	25	0	8	13	10	86	16	7	64	88	99	90
						i		j				
27	25	0	8	13	10	7	16	86	64	88	99	90
							i	j				
16	25	0	8	13	10	7	27	86	64	88	99	90

Complexity Analysis

Worst-case:

$$\left.\begin{array}{c} W(n) = W(n-1) + n - 1 \\ W(1) = 0 \end{array}\right\} \Rightarrow W(n) = n(n-1)/2$$

Best-case:

$$\left.\begin{array}{c} T(n) = 2T(n/2) + n - 1 \\ T(1) = 0 \end{array}\right\} \Rightarrow T(n) = \Theta(n \log n)$$

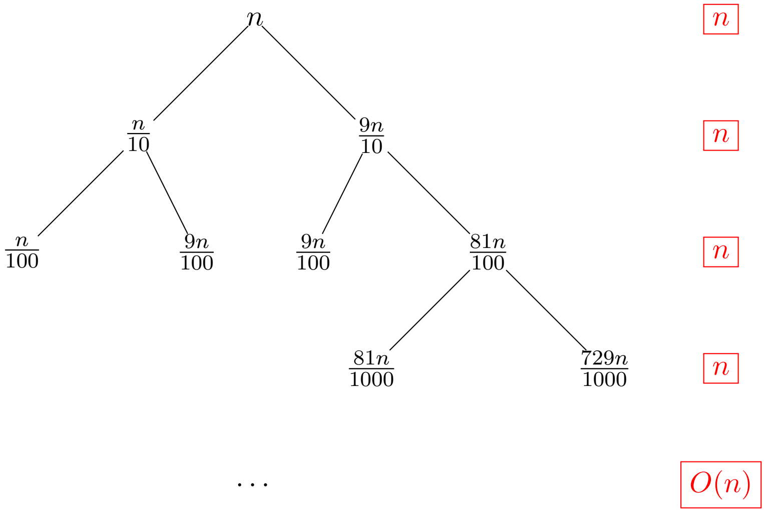
Complexity of Constant Partition

Constant Partition. The ratio of subproblems vs. original problem is a fixed constant, such as $1:9$.

$$\begin{cases} T(n) = T(n/10) + T(9n/10) + n \\ T(1) = 0 \end{cases}$$

Solving via recursion tree $\Rightarrow T(n) = \Theta(n \log n)$

Recursion Tree



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

Average-Case Complexity

Suppose the first element finally appear at position $1, 2, \ldots, n$ with equal probability, i.e., $1/n$. We analyze the size of resulting subproblems

- appear at position 1: $T(0)$, $T(n-1)$
- appear at position 2: $T(1)$, $T(n-2)$
- $\ldots$
- appear at position $n-1$: $T(n-2)$, $T(1)$
- appear at position n: $T(n-1)$, $T(0)$

The cost of all subproblems: $2(T(1) + T(2) + \cdots + T(n-1))$

The cost of partition: $n-1$ compares

Average-Case Complexity

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

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

$$T(1) = 0$$

We simplify the recurrence relation via subtraction $\Rightarrow$

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

See pp. 47 on Lec 3 if you cannot remember it.

Outline

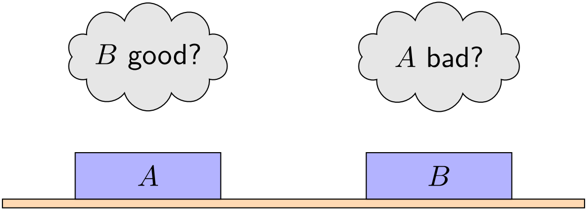
1. Introduction of Divide-and-Conquer
2. QuickSort
3. Chip Test
4. Selection Problem
   - Selecting Max and Min
   - Selecting the Second Largest
   - General Selection Problem
5. Closest Pair of Points

Chip Test

Chip factory only admits basic test method.

Basic test method. Put two chips A and B on the testbed, begin the mutual test

- the test report is "good" or "bad"



Assumption. The report from good chip is always correct, but the report from bad chip is non-deterministic (probably wrong)

Analysis of Test Report

A's report	B's report	Conclusion
B is good	A is good	A, B are both good or bad
B is good	A is bad	at least one is bad
B is bad	A is good	at least one is bad
B is bad	A is bad	at least one is bad

Problem of Chip Test

Input. n chips, $\#(\text{good}) - \#(\text{bad}) \geq 1$

Question. Devise a test method to choose one good chip from n chips

Requirement. The number of mutual tests is minimum

Problem of Chip Test

Input. n chips, $\#(\text{good}) - \#(\text{bad}) \geq 1$

Question. Devise a test method to choose one good chip from n chips

Requirement. The number of mutual tests is minimum

Starting point. Given a chip A, how to check if A is good or bad

Method. Using other $n-1$ chip to test A.

- Idea: utilize the odevity of n

Case 1: n is Odd

Example. $n = 7$, #(good chips) ≥ 4.

- A is good $\Leftrightarrow$ at least 3 among 6 reports "good"
- A is bad $\Leftrightarrow$ at least 4 among 6 reports "bad"

Generalize to n is odd, #(good chips) $\geq (n+1)/2$.

- A is good: $\Leftrightarrow$ at least $(n-1)/2$ reports "good"
- A is bad: $\Leftrightarrow$ at least $(n+1)/2$ reports "bad"

Key observation. The test result is of if and only if flavor. Thus, it constitutes a necessary and sufficient condition.

Criteria: in $n-1$ reports

- at least one half reports "good" $\Rightarrow$ A is good
- more than one half reports "bad" $\Rightarrow$ A is bad

Case 2: n is Even

Example. $n = 8$, #(good chips) ≥ 5.

- A is good $\Leftrightarrow$ at least 4 from 7 report "good"
- A is bad $\Leftrightarrow$ at least 5 from 7 report "bad"

Generalize to n is even, #(good chips) $\geq n/2 + 1$.

- A is good $\Leftrightarrow$ at least $n/2$ report "good"
- A is bad $\Leftrightarrow$ at least $n/2 + 1$ report "bad"

Key observation. The test result is also of if and only if flavor. Thus, it constitutes a necessary and sufficient condition.

Criteria: in $n - 1$ reports

- at least one half reports "good" $\Rightarrow$ A is good
- more than one half reports "bad" $\Rightarrow$ A is bad

Brute Force Algorithm

Test method. Randomly pick a chip, apply the aforementioned test. If it is good, then the test is over. Else, discard it and randomly pick another chip from the rest, until get a good chip.

- correctness: #(good chips) is always more than half.

Time complexity

- 1-st round: random one is bad, at most $n-1$ time tests
- 2-rd round: random one is bad, at most $n-2$ time tests
- ...
- i-th round: random one is bad, at most $n-i$ time tests

Brute Force Algorithm

Test method. Randomly pick a chip, apply the aforementioned test. If it is good, then the test is over. Else, discard it and randomly pick another chip from the rest, until get a good chip.

- correctness: #(good chips) is always more than half.

Time complexity

- 1-st round: random one is bad, at most $n-1$ time tests
- 2-rd round: random one is bad, at most $n-2$ time tests
- $\ldots$
- i-th round: random one is bad, at most $n-i$ time tests

> Nice discovery by [2020 江锴杰]: in $i > 1$ round, we can randomly discard one chip then test $\rightsquigarrow$ requiring at most $n-1-2i$ time tests

The complexity in the worst-case is $\Theta(n^2)$

Divide-and-Conquer

Assume n is even, divide n chips into two groups and begin mutual test; the rest chips form a subproblem and begin the next round test

Divide-and-Conquer

Assume n is even, divide n chips into two groups and begin mutual test; the rest chips form a subproblem and begin the next round test

Test-and-Elimination rules

- "good, good" $\rightsquigarrow$ pick a random one into the next round
- other cases $\rightsquigarrow$ discard them all

Divide-and-Conquer

Assume n is even, divide n chips into two groups and begin mutual test; the rest chips form a subproblem and begin the next round test

Test-and-Elimination rules

- "good, good" $\rightsquigarrow$ pick a random one into the next round
- other cases $\rightsquigarrow$ discard them all

The end condition of recursion: $n \leq 3$

- 3 chips: one test suffices (think why? retain the same property as the original problem)
  1. "good, good": randomly pick one and output it
  2. "good, bad": output the rest one
  3. "bad, bad": output the rest one
- 1 or 2 chips: both are good, no more test is needed

Correctness of Divide-and-Conquer Algorithm

Claim. When n is even, after one round of test, in the rest chips, #(good chips) - #(bad chips) ≥ 1

Proof. Consider the following three cases:

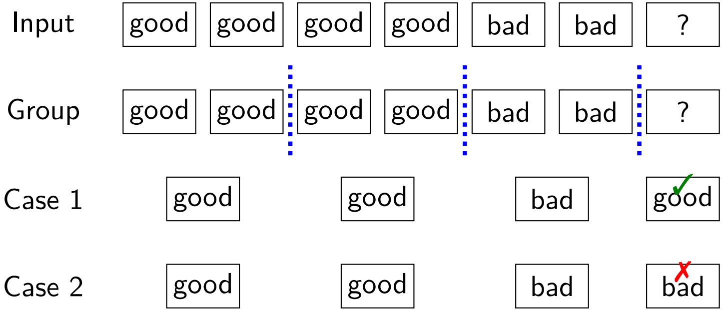
1. Both are good (i groups) $\rightsquigarrow$ keep a random one
2. One is good, one is bad (j groups) $\rightsquigarrow$ discard them all
3. Both are bad (k groups) $\rightsquigarrow$ keep a random one or discard them all

After one round test, #(good chips) $= i$, #(bad chips) $\leq k$

$$\left.\begin{cases} 2i + 2j + 2k = n & \#(\text{chips}) \text{ before test} \\ 2i + j > 2k + j & \#(\text{good chips}) > \#(\text{bad chips}) \end{cases}\right\} \Rightarrow i > k$$

Adjust when n is Odd

When n is odd, there would one chip left without group member.



Adjustment. When n is odd, add one-round direct test for the ungrouped chip

- if it is good, the algorithm is over
- else, discard it and enter into the next round (since $n-1$ chips satisfying original property)

Pseudocode

Algorithm 8: ChipTest(n)

```
if n = 3 then                                   //smallest case
    randomly pick 2 chips;
    if both are good then return a random one;
    else return the rest one;
end
if n =2 or 1 then return a random one;

divide into ⌊n/2⌋ groups;                      // adjust when n is odd
for i = 1 to ⌊n/2⌋ do
    if both are good then keep a random one;
    else discard both of them;
end
n ← #(rest chips);
ChipTest(n);
```

Complexity Analysis

For input size n, after each round test, the number of chips reduces at least by half

- #(test) (include addition adjustment when n is odd): $\Theta(n)$

Recurrence relation

$$\left.\begin{array}{c} W(n) = W(n/2) + \Theta(n) \\ W(3) = 1, W(2) = W(1) = 0 \end{array}\right\} \Rightarrow W(n) = \Theta(n)$$

Summary of Divide-and-Conquer chip test algorithm

- Adjustment $\rightsquigarrow$ guarantee the subproblem is of the same type as the original problem
- branching factor $a = 1$ & dividing-merging cost $f(n) = \Theta(n)$ $\rightsquigarrow$ ensure remarkable efficiency improvement over brute-force algorithm

Outline

1. Introduction of Divide-and-Conquer
2. QuickSort
3. Chip Test
4. Selection Problem
   - Selecting Max and Min
   - Selecting the Second Largest
   - General Selection Problem
5. Closest Pair of Points

Outline

1. Introduction of Divide-and-Conquer
2. QuickSort
3. Chip Test
4. Selection Problem
   - Selecting Max and Min
   - Selecting the Second Largest
   - General Selection Problem
5. Closest Pair of Points

General Selection Problem

Selection. Given n elements from a totally ordered universe S, find k-th smallest.

- Minimum: $k = 1$ min element
- Maximum: $k = n$ max element
- Median: $k = \lfloor (n+1)/2 \rfloor$
  - n is odd, the median is unique, $k = (n+1)/2$
  - n is even, the median has two choices: $n/2$ and $n/2+1$, typically we choose $k = n/2$

Known results

- $O(n)$ compares for min or max.
- Naive algorithms for general selection: $O(n \log n)$ compares by sorting, and $O(n \log k)$ compares with a binary heap.

Applications. order statistics; find the "top k"; bottleneck paths

Q. Can we accomplish general selection with $O(n)$ compares?

A. Yes! Selection is easier than sorting.

About Median

Median of the list of numbers is its 50th percentile: half the numbers are larger than it, and half are smaller.

Example. The median of $[45, 1, 10, 30, 25]$ is 25.

Meaning of median. Summarize a set of numbers by a single, typical value.

- The *mean* or *average* is also very commonly used for this purpose.
- But, median is in a sense more typical
  - always one of the data values, unlike the mean
  - less sensitive to outliers.

Counterexample. The median of hundreds 1's is 1, as is mean. However, if just one of these numbers gets corrupted to 10000, the mean shoots above 100, while the median is unaffected with large probability.

Selecting Max

Algorithm. Sequential compare

1	8	4	17	3	12

1

max

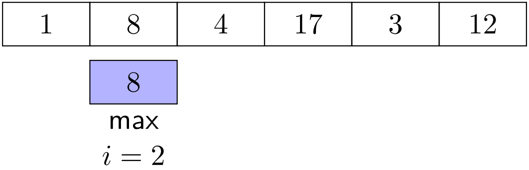
$i = 1$

Output. $max = 17$, $i = 4$

Worst-case complexity. $W(n) = n - 1$

Selecting Max

Algorithm. Sequential compare

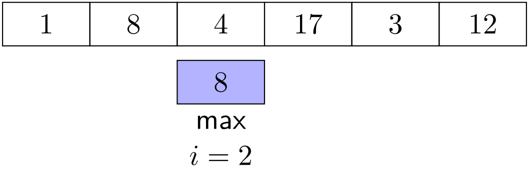


Output. $max = 17$, $i = 4$

Worst-case complexity. $W(n) = n - 1$

Selecting Max

Algorithm. Sequential compare



Output. $max = 17$, $i = 4$

Worst-case complexity. $W(n) = n - 1$

Selecting Max

Algorithm. Sequential compare

1	8	4	17	3	12

17

max

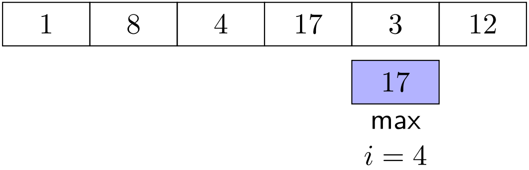
$i = 4$

Output. $max = 17$, $i = 4$

Worst-case complexity. $W(n) = n - 1$

Selecting Max

Algorithm. Sequential compare



Output. $max = 17$, $i = 4$

Worst-case complexity. $W(n) = n - 1$

Selecting Max

Algorithm. Sequential compare

1	8	4	17	3	12
					17

max

$i = 4$

Output. $max = 17$, $i = 4$

Worst-case complexity. $W(n) = n - 1$

Pseudocode

Algorithm 9: SelectMax(A, n)

```
Input: A[n]
Output: max, j
max ← A[1];
j ← 1;
for i ← 2 to n do
    if max < A[i] then
        max ← A[i];
        j ← i;
    end
end
return max, j
```

Selecting Max and Min

Naive Algorithm

1. Sequential compare, first choose max and remove it
2. Then choose min in the left list, using the same algorithm but retain smaller element after each compare.

Selecting Max and Min

Naive Algorithm

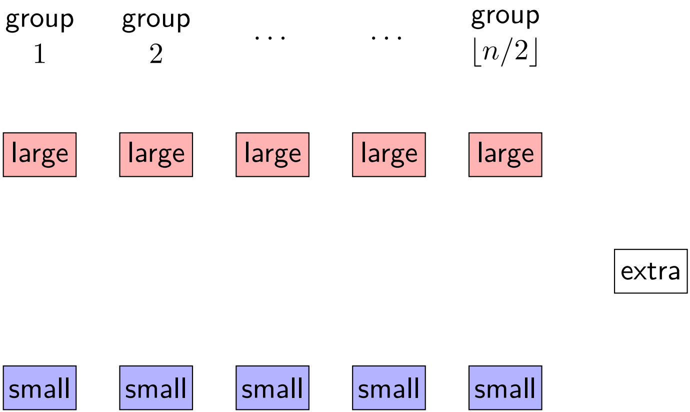
1. Sequential compare, first choose max and remove it
2. Then choose min in the left list, using the same algorithm but retain smaller element after each compare.

Worst-case time complexity

$$W(n) = n - 1 + n - 2 = 2n - 3$$

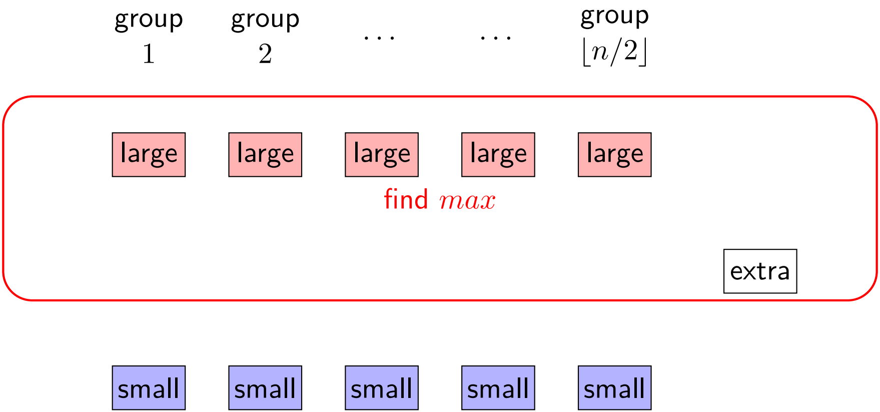
Grouping Algorithm

Idea. Split list into higher list and lower list.



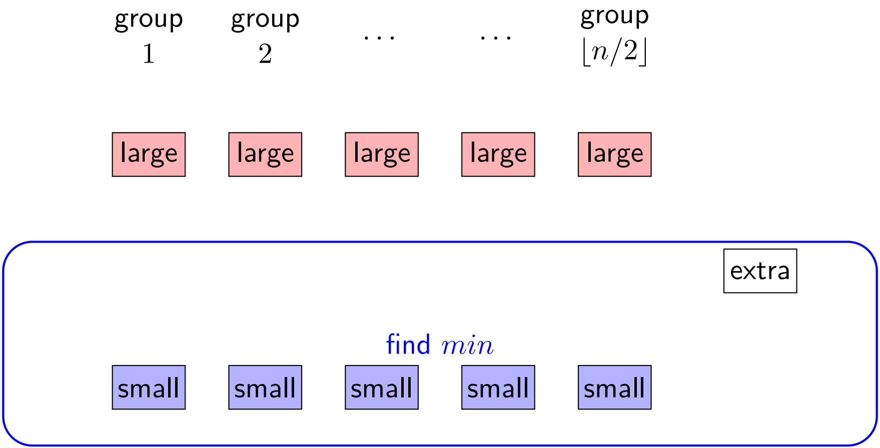
Grouping Algorithm

Idea. Split list into higher list and lower list.



Grouping Algorithm

Idea. Split list into higher list and lower list.



Pseudocode of Select Max and Min

Algorithm 10: FindMaxMin(A, n)

Input: unsorted $A[n]$

Output: max, min

1: divide n elements into $\lfloor n/2 \rfloor$ groups;
2: compare two elements in each group, obtain $\lfloor n/2 \rfloor$ smaller and $\lfloor n/2 \rfloor$ larger;
3: find max in $\lfloor n/2 \rfloor$ larger elements and the extra element;
4: find min in $\lfloor n/2 \rfloor$ smaller elements and the extra element;

Summing it up, $W(n) = 3\lfloor n/2 \rfloor$

- Group inside compare: $\lfloor n/2 \rfloor$
- When n is even: select max: $n/2 - 1$, select min: $n/2 - 1$
- When n is odd: select max: $(n-1)/2 + 1 - 1$, select min: $(n-1)/2 + 1 - 1$

Divide-and-Conquer Strategy

Q. Can we make this algorithm recursive on n?

A. Yes, but not optimal. Since selecting min in higher list (resp. selecting max in lower list) is a waste.

Divide-and-Conquer Strategy

Q. Can we make this algorithm recursive on n?

A. Yes, but not optimal. Since selecting min in higher list (resp. selecting max in lower list) is a waste.

How to solve FindMinMax via divide-and-conquer?

Divide-and-Conquer Strategy

Q. Can we make this algorithm recursive on n?

A. Yes, but not optimal. Since selecting min in higher list (resp. selecting max in lower list) is a waste.

How to solve FindMinMax via divide-and-conquer?

1. Divide A into left halve A_1 and right halve A_2
2. Recursively select max_1 and min_1 in A_1
3. Recursively select max_2 and min_2 in A_2
4. $max \leftarrow \max\{max_1, max_2\}$
5. $min \leftarrow \min\{min_1, min_2\}$

Divide-and-Conquer Strategy

Q. Can we make this algorithm recursive on n?

A. Yes, but not optimal. Since selecting min in higher list (resp. selecting max in lower list) is a waste.

How to solve FindMinMax via divide-and-conquer?

1. Divide A into left halve A_1 and right halve A_2
2. Recursively select max_1 and min_1 in A_1
3. Recursively select max_2 and min_2 in A_2
4. $max \leftarrow \max\{max_1, max_2\}$
5. $min \leftarrow \min\{min_1, min_2\}$

[2020 游泓慧] This recursive algorithm can be made iteratively like MergeSort.

Worse-Case Complexity

Assume $n = 2^k$, the recurrence relation of $W(n)$ is as below:

$$\begin{cases} W(n) = 2W(n/2) + 2 \\ W(2) = 1 \end{cases}$$

Solving for the exact value via substitute-then-iterate method:

$$\begin{aligned} W(2^k) =& 2W(2^{k-1}) + 2 \\ =& 2(2W(2^{k-2}) + 2) + 2 \\ =& 2^2 W(2^{k-2}) + 2^2 + 2 \\ =& 2^i W(2^{k-i}) + 2^i + \cdots + 2 \end{aligned}$$

The right side reach the initial value when $i = k - 1$, the summation is:

$$2^{k-1} + (2^{k-1} + \cdots + 2^2 + 2) = 3 \cdot 2^{k-1} - 2 = 3n/2 - 2$$

Summary

Select Max. Sequentially compare, requires at most $n-1$ compares

---

Select Max and Min. (worst-case)

- Naive algorithm: $2n-3$
- Grouping algorithm: $3\lfloor n/2 \rfloor$
- Divide-and-Conquer: $3n/2-2$

It can be proved that grouping algorithm and divide-and-conquer algorithm are optimal for SelectMinMax, achieving the lower bound

Outline

1. Introduction of Divide-and-Conquer
2. QuickSort
3. Chip Test
4. Selection Problem
   - Selecting Max and Min
   - Selecting the Second Largest
   - General Selection Problem
5. Closest Pair of Points

Selecting the Second Largest

Input. $A[n]$

Output. The second largest max'

Selecting the Second Largest

Input. $A[n]$

Output. The second largest max'

Naive algorithm: sequential compare

1. select max from $A[n]$ via sequential compare
2. select max' from $A[n]\backslash max$, which is exactly the second largest

Time complexity: $W(n) = (n-1) + (n-2) = 2n-3$

Optimized Method

Observation. The sufficient and necessary condition to be the second largest: only beaten by the largest

To determine the second largest element, we must know the largest element first.

Optimized Method

Observation. The sufficient and necessary condition to be the second largest: only beaten by the largest

To determine the second largest element, we must know the largest element first.

Idea. Trade space for time

- Record the elements that are beaten by the largest element in a set L along the way finding the largest
- Selecting the largest element among the elements in L.

Tournament Algorithm for Second Largest

1. Divide elements into groups of size 2
2. In each group, two elements compare, the larger one goes to the next level, and (only) records the beaten element in its list.
3. Repeat the above steps until there is only one element left, a.k.a. max
4. Select the largest element form the list of max, a.k.a. max'

The name comes from single-elimination tournament: players play in two-sided matches, and the winner is promoted to the next level up. The hierarchy continues until the final match determines the ultimate winner. The tournament determines the best player, but the player who was beaten in the final match may not be the second best - he may be inferior to other players the winner bested.

Pseudocode of SelectSecond

Algorithm 11: FindSecond(A, n)

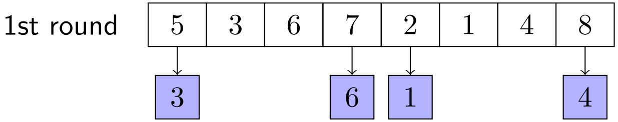
Input: $A[n]$

Output: Second largest element max'

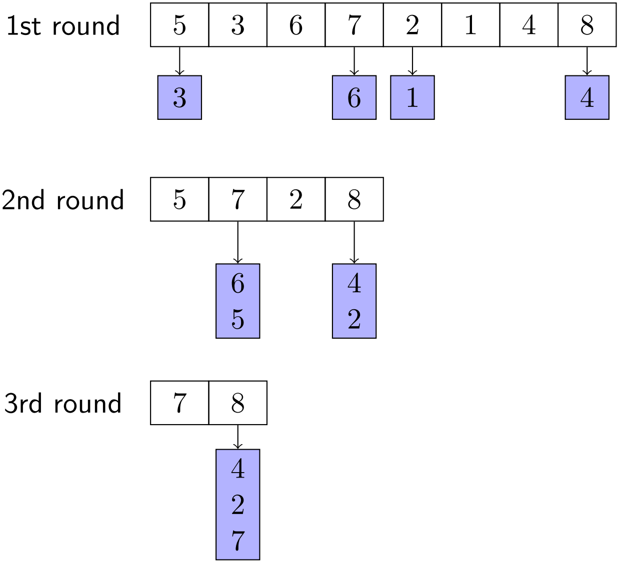
```
1: k ← n                    //number of elements;
2: divide k elements into ⌊k/2⌋ groups;
3: In each group, two elements compare to select larger;
4: record the loser into the list of winner;
5: if k is odd then k ← 1 + ⌊k/2⌋;
6: else k ← k/2;
7: if k > 1 then goto 2;
8: max ← ultimate winner;
9: max' ← max in the list of ultimate winner
```

- line 2-4: one round match
- line 5-6: compute the number of winners — $\lceil k/2 \rceil$
- line 7: next round match

Demo of SelectSecond



Demo of SelectSecond



Complexity Analysis (1/3)

Proposition 1. Assume there are n elements, at most $\lceil n/2^i \rceil$ elements are left after i-th round match.

Proof. Carry mathematical induction over i:

- Induction basis $i = 1$: divide into $\lfloor n/2 \rfloor$ groups, kick off $\lfloor n/2 \rfloor$ elements, the number of elements prompted to the next level is
$$n - \lfloor n/2 \rfloor = \lceil n/2 \rceil$$
- Induction step: $P(i) \Rightarrow P(i+1)$. Assume the number of elements after i-th match is at most $\lceil n/2^i \rceil$, then after $i+1$-th match, the number of elements is
$$\text{continuous rounding property} \Rightarrow \lceil \lceil n/2^i \rceil /2 \rceil = \lceil n/2^{i+1} \rceil$$

Complexity Analysis (2/3)

Proposition 2. max compares with $\lceil \log n \rceil$ elements

Proof. Assume max is selected after k round match. According to Proposition 1, $\lceil n/2^k \rceil = 1$.

- if $n = 2^d$ for some $d \in \mathbb{Z}$, then:

$$\log n = \lceil \log n \rceil$$
$$k = d = \lceil \log n \rceil$$

- else $2^d < n < 2^{d+1}$ for some $d \in \mathbb{Z}$, then

$$d < \log n < d + 1$$
$$k = d + 1 = \lceil \log n \rceil$$

Complexity Analysis (3/3)

Phase 1: number of elements is n

- number of compares $= n - 1 \Leftarrow n - 1$ elements are eliminated (one compare kicks off one element)

Phase 2: number of elements is $\lceil \log n \rceil$, which is exactly the size of winner's list according to Proposition 2

- number of compares $= \lceil \log n \rceil - 1 \Leftarrow$ sequential compare or tournament algorithm ($\lceil \log n \rceil - 1$ elements are eliminated)

The overall time complexity:

$$\begin{aligned} W(n) &= n - 1 + \lceil \log n \rceil - 1 \\ &= n + \lceil \log n \rceil - 2 \end{aligned}$$

Summary of Finding Second Max

Find the second max

- Naive algorithm (invoking FindMax twice) $2n - 3$
- Tournament Algorithm: $n + \lceil \log n \rceil - 2$
  - main trick: trade space for efficiency

Outline

1. Introduction of Divide-and-Conquer
2. QuickSort
3. Chip Test
4. Selection Problem
   - Selecting Max and Min
   - Selecting the Second Largest
   - General Selection Problem
5. Closest Pair of Points

Motivation of General Selection Problem

Computing the median has wide applications

- Naive algorithm: sort-then-find $\Rightarrow W(n) = n \log n$
- Ideally we expect linear complexity.
- We have reason to be hopeful. Sorting does far more than we really need — we do not care about the relative ordering of the rest of them.

When looking for a recursive algorithm, it is paradoxically more easier to work with a *more general* version of the problem – for the simple reason that this gives a more powerful step to recurse upon.

- It also generalizes find the second largest.

General Selection Problem

Problem. Select k-th smallest

Input. list $A[n]$, integer $k \in [n]$

Output. the k-th smallest

Example 1. $A = \{3, 4, 8, 2, 5, 9, 10\}$, $k = 4$, solution $= 5$

Example 2. Statistical data set S, $|S| = n$, select the median, $k = \lceil n/2 \rceil$

Naive Algorithms

Algorithm 1

- invoke algorithm SelectMin k times
- time complexity is: $O(kn)$

Naive Algorithms

Algorithm 1

- invoke algorithm SelectMin k times
- time complexity is: $O(kn)$

Algorithm 2

- sort, then output the k-smallest number
- time complexity is: $O(n \log n)$

Divide-and-Conquer: QuickSelect

Assume n elements are distinct, the idea is 3-way partition:

Divide-and-Conquer: QuickSelect

Assume n elements are distinct, the idea is 3-way partition:

- Using some m^* as pivot to split S so that m^* is in place, smaller elements in left subarray S_1 and larger elements in right subarray S_2
  1. If $k \leq |S_1|$, then find the k-smallest in S_1
  2. If $k = |S_1| + 1$, then m^* is the k-smallest
  3. If $k > |S_1| + 1$, then find the $k - |S_1| - 1$-smallest in S_2

Divide-and-Conquer: QuickSelect

Assume n elements are distinct, the idea is 3-way partition:

- Using some m^* as pivot to split S so that m^* is in place, smaller elements in left subarray S_1 and larger elements in right subarray S_2
  1. If $k \leq |S_1|$, then find the k-smallest in S_1
  2. If $k = |S_1| + 1$, then m^* is the k-smallest
  3. If $k > |S_1| + 1$, then find the $k - |S_1| - 1$-smallest in S_2

The efficiency is determined by the size of subproblems.

Divide-and-Conquer: QuickSelect

Assume n elements are distinct, the idea is 3-way partition:

- Using some m^* as pivot to split S so that m^* is in place, smaller elements in left subarray S_1 and larger elements in right subarray S_2
  1. If $k \leq |S_1|$, then find the k-smallest in S_1
  2. If $k = |S_1| + 1$, then m^* is the k-smallest
  3. If $k > |S_1| + 1$, then find the $k - |S_1| - 1$-smallest in S_2

The efficiency is determined by the size of subproblems.

Question. How to choose m^* to control the size of subproblems?

Divide-and-Conquer: QuickSelect

Assume n elements are distinct, the idea is 3-way partition:

- Using some m^* as pivot to split S so that m^* is in place, smaller elements in left subarray S_1 and larger elements in right subarray S_2
    1. If $k \leq |S_1|$, then find the k-smallest in S_1
    2. If $k = |S_1| + 1$, then m^* is the k-smallest
    3. If $k > |S_1| + 1$, then find the $k - |S_1| - 1$-smallest in S_2

The efficiency is determined by the size of subproblems.

Question. How to choose m^* to control the size of subproblems?

Ideal case: find the exact median, but this means we have to solve a problem of the same size first

Divide-and-Conquer: QuickSelect

Assume n elements are distinct, the idea is 3-way partition:

- Using some m^* as pivot to split S so that m^* is in place, smaller elements in left subarray S_1 and larger elements in right subarray S_2
  1. If $k \leq |S_1|$, then find the k-smallest in S_1
  2. If $k = |S_1| + 1$, then m^* is the k-smallest
  3. If $k > |S_1| + 1$, then find the $k - |S_1| - 1$-smallest in S_2

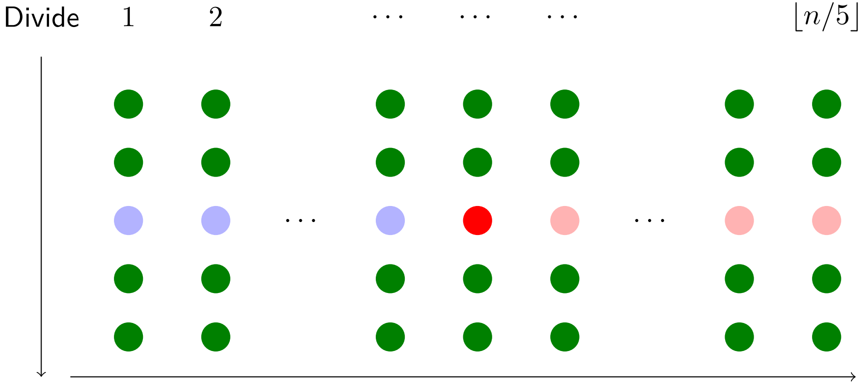
The efficiency is determined by the size of subproblems.

Question. How to choose m^* to control the size of subproblems?

Ideal case: find the exact median, but this means we have to solve a problem of the same size first

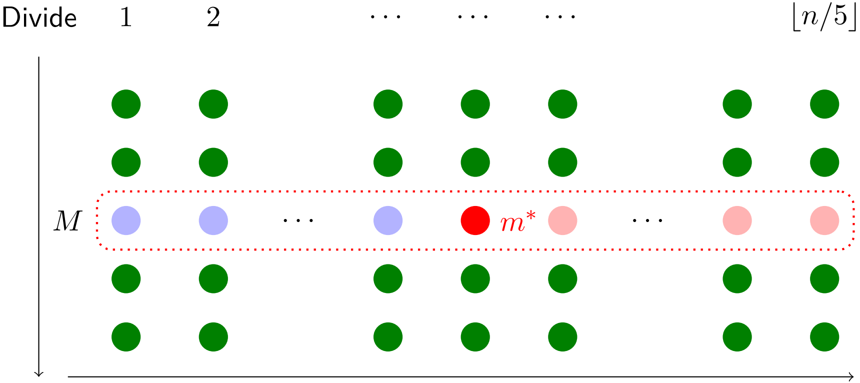
Real case: use a quasi-median instead — the median of subarrary median

The Selection of m^* and Dividing

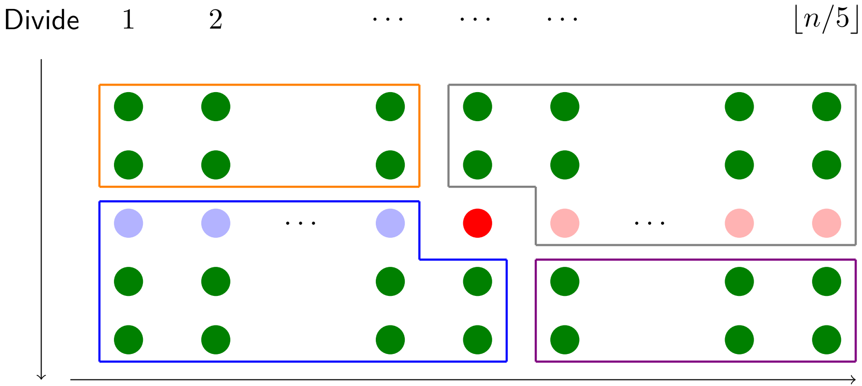


1. sort each group in descending order

The Selection of m^* and Dividing



1. sort each group in descending order
2. find the median of median, then re-organize the group to place m^* in the middle



- A zone: require compares with m^*
- B zone: larger than m^*
- C zone: smaller than m^*
- D zone: require compares with m^*

Demo: $n = 15$, $k = 6$

8	2	3	5	7	6	11	14	1	9	13	10	4	12	15

	8	14	15	
	7	11	13	
M	5	9	12	$m^* = 9$
	3	6	10	
	2	1	4	

	8	14	15	
A	7	11	13	B
	5	9	12	
C	3	6	10	D
	2	1	4	

Divide to Subproblems

	8	14	15	
	7	11	13	
S_1	5	9	12	S_2
	3	6	10	
	2	1	4	

subproblem: $\{8, 7, 5, 3, 2, 6, 1, 4\}$

size of subproblem $= 8$, $k = 6$

Pseudocode of QuickSelect

Algorithm 12: QuickSelect$(A[n], k)$

1: divide elements in A into groups of size 5, there are totally $m = \lceil n/5 \rceil$ groups;
2: sort each group and place the medians into M;
3: $m^* \leftarrow$ QuickSelect$(M, \lceil |M|/2 \rceil)$ //split S into A, B, C, D;
4: For elements in A and D, record the ones smaller than m^* into S_1, the ones larger than m^* into S_2;
5: $S_1 \leftarrow S_1 \cup C$, $S_2 \leftarrow S_2 \cup B$;
6: **if** $k = |S_1| + 1$ **then** output m^*;
7: **else if** $k \leq |S_1|$ **then**
8: QuickSelect(S_1, k);
9: **else** QuickSelect$(S_2, k - |S_1| - 1)$;

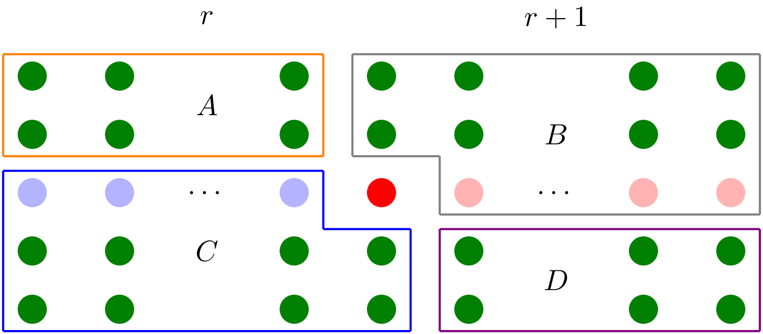
- line 4-5: split
- line 7-9: recursively solve subproblems

Complexity Analysis

Each round of QuickSelect algorithm consists of two recursive calls of QuickSelect

1. select median from median M as pivot for dividing
2. the real subproblem

The overall complexity of the algorithm is determined by the quality of dividing



unbalanced divide $\rightsquigarrow$ bad complexity

We consider an extreme case: elements in A zone and D zone go to the same side.

- $n = 5(2r+1)$, $|A| = |D| = 2r$
- the size of subproblems is at most: $2r + 2r + 3r + 2 = 7r + 2$

Estimation of the Size of Subproblems

Assume $n = 5(2r+1)$, $|A| = |D| = 2r$

$$r = \frac{n/5 - 1}{2} = \frac{n}{10} - \frac{1}{2}$$

The size of subproblem after dividing is at most:

$$\begin{aligned} 7r + 2 &= 7\left(\frac{n}{10} - \frac{1}{2}\right) + 2 \\ &= \frac{7n}{10} - \frac{3}{2} < \frac{7n}{10} \end{aligned}$$

Recurrence Relation for Worst-Case Complexity

Worse-case complexity $W(n)$

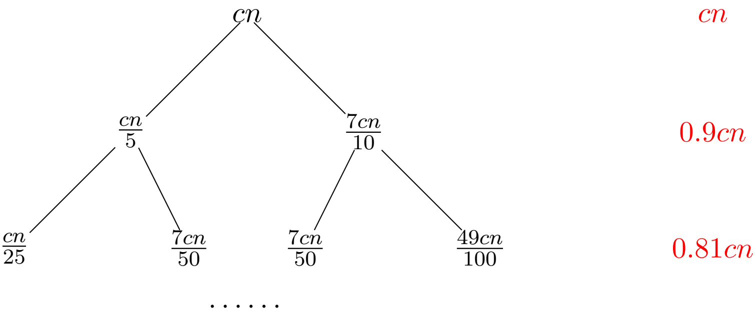
- line 2: $\Theta(n)$ //select median among each 5 elements (constant time), form M
- line 3: $W(n/5)$ //find median m^* of M
- line 4: $\Theta(n)$ //divide S using m^* (only need to compare A and D)
- line 8-9: $W(7n/10)$ //recursive call to the subproblem

The recurrence relation is:

$$W(n) \leq W(n/5) + W(7n/10) + \Theta(n)$$

Solving via Recurrence Tree

$$W(n) \leq W(n/5) + W(7n/10) + \Theta(n)$$



- the depth of tree is $\Theta(\log n) \Rightarrow$ the number of leaf nodes is $\Theta(n)$; the cost of solving smallest problem is constant $\Rightarrow$ the cost of all smallest problem is $\Theta(n)$

$$W(n) \leq cn(1 + 0.9 + 0.9^2 + \dots) + \Theta(n) = \Theta(n)$$

Discussion

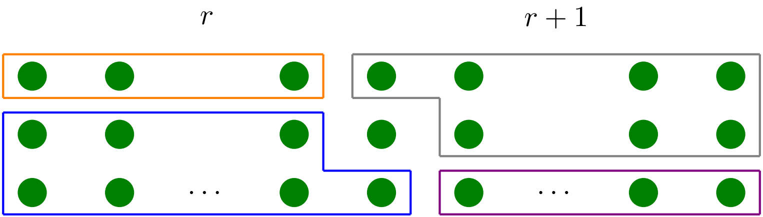
Q. Why we have to divide the elements into groups of size 5? Can we choose the group size as 3 or 7?

Analysis. The group size will affect the overall complexity. Let t be the group size.

1. The cost of selecting m^* is related to $|M| = n/t$. The larger is t, the smaller is $|M|$.
2. The size of subproblem after dividing is related to t. The larger is t, the larger is $|S_i|$.

We have to hit the sweet balance.

Case Study: $t = 3$



$n = 3(2r + 1)$, $r = (n/3 - 1)/2 = n/6 - 1/2$

The subproblem size is at most: $4r + 1 = 4n/6 - 1$

Recurrence relation of worst-case complexity is:

$$W(n) = W(n/3) + W(4n/6) + cn$$

Solving by recurrence tree $\Rightarrow W(n) = \Theta(n \log n)$

Summary

Crux. When $|M| + |S_i| < n$, then the total cost on the inner nodes of recurrence tree forms geometric series with common ratio less than 1. $W(n)$ is $\Theta(n)$ only in this case.

Selecting max or min

- Naive sequential compare: $W(n) = n - 1$

Selecting max and min

- grouping algorithm: $W(n) = 3\lfloor n/2 \rfloor$
- divide-and-conquer: $W(n) = 3n/2 - 2$

Selecting the second largest: the tournament algorithm

$$W(n) = n + \lceil \log n \rceil - 2$$

General selecting problem: divide-and-conquer algorithm

$$W(n) = \Theta(n)(\approx 44n)$$

Follow-up Work on Linear-time Selection (Median of Medians)

Proposition. [Blum-Floyd-Pratt-Rivest-Tarjan 1973] There exists a compare-based selection algorithm whose $W(n) = O(n)$.

Time Bounds for Selection

by .

Manuel Blum, Robert W. Floyd, Vaughan Pratt,
Ronald L. Rivest, and Robert E. Tarjan

Abstract

The number of comparisons required to select the i-th smallest of n numbers is shown to be at most a linear function of n by analysis of a new selection algorithm -- PICK. Specifically, no more than 5.4305 n comparisons are ever required. This bound is improved for

Theory.

- Optimized version of BFPRT: $\leq 5.4305n$ compares.
- Best known upper bound [Dor-Zwick 1995]: $\leq 2.95n$.
- Best known lower bound [Dor-Zwick 1999]: $\geq (2 + \epsilon)n$.

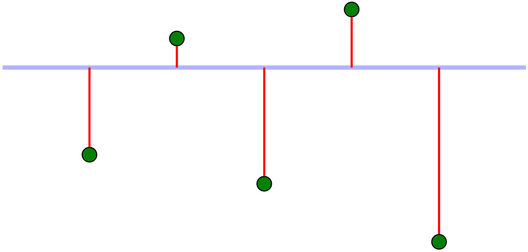
Practice.

- Constant and overhead (currently) too large to be useful.

Application of Selecting Median: Optimal Pipeline Design

Problem. Assume there are n oil wells, the task is building a pipeline system to connect n oil wells. The pipeline system consists of a horizontal main pipeline, each oil well connects to the main pipeline via a vertical pipeline.

Optimization Goal. How to choose the position of main pipeline to minimize the total length of vertical pipelines?



Optimal Solution: the Median of Y Coordinates

The main pipeline is horizontal $\Rightarrow$ optimal solution is independent of the distribution of X coordinates

y_n

$\vdots$

$y_{n/2+1}$

y

$y_{n/2}$

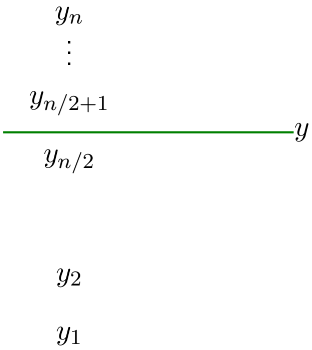
y_2

y_1

Optimal Solution: the Median of Y Coordinates

The main pipeline is horizontal $\Rightarrow$ optimal solution is independent of the distribution of X coordinates

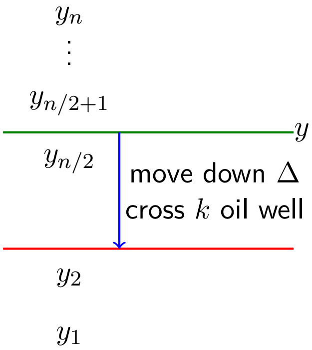
If the median is unique, then select it; else, choose any median of the two is fine (any horizontal line between the medians is also fine).



Optimal Solution: the Median of Y Coordinates

The main pipeline is horizontal $\Rightarrow$ optimal solution is independent of the distribution of X coordinates

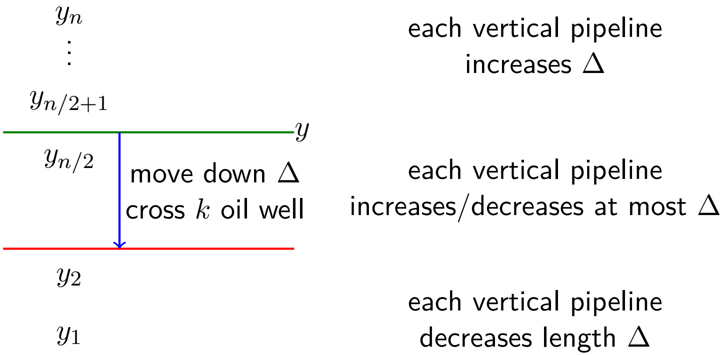
If the median is unique, then select it; else, choose any median of the two is fine (any horizontal line between the medians is also fine).



Optimal Solution: the Median of Y Coordinates

The main pipeline is horizontal $\Rightarrow$ optimal solution is independent of the distribution of X coordinates

If the median is unique, then select it; else, choose any median of the two is fine (any horizontal line between the medians is also fine).



Analysis

We first consider the effect of moving down:

Analysis

We first consider the effect of moving down:

if n is odd: the median is unique (number of wells that is above/below of median $n' = (n-1)/2$)

- variation: $+(n'+1)\Delta$, at most $\pm k\Delta$, $-(n'-k)\Delta$, $1 \leq k \leq n'$

$$\text{sum of variation} = \Delta \pm k\Delta + k\Delta > 0$$

Analysis

We first consider the effect of moving down:

if n is odd: the median is unique (number of wells that is above/below of median $n' = (n-1)/2$)

- variation: $+(n'+1)\Delta$, at most $\pm k\Delta$, $-(n'-k)\Delta$, $1 \leq k \leq n'$

$$\text{sum of variation} = \Delta \pm k\Delta + k\Delta > 0$$

if n is even: w.l.o.g. (above $n' = n/2$, below $n' = n/2$)

- variation: $+n'\Delta$, at most $\pm k\Delta$, $-(n'-k)\Delta$, $1 \leq k \leq n'$

$$\text{sum of variation} = \pm k\Delta + k\Delta \geq 0$$

Analysis

We first consider the effect of moving down:

if n is odd: the median is unique (number of wells that is above/below of median $n' = (n-1)/2$)

- variation: $+(n'+1)\Delta$, at most $\pm k\Delta$, $-(n'-k)\Delta$, $1 \leq k \leq n'$

$$\text{sum of variation} = \Delta \pm k\Delta + k\Delta > 0$$

if n is even: w.l.o.g. (above $n' = n/2$, below $n' = n/2$)

- variation: $+n'\Delta$, at most $\pm k\Delta$, $-(n'-k)\Delta$, $1 \leq k \leq n'$

$$\text{sum of variation} = \pm k\Delta + k\Delta \geq 0$$

In summary, the total length of vertical pipelines increases if the main pipeline moving down from median.

Analysis

We first consider the effect of moving down:

if n is odd: the median is unique (number of wells that is above/below of median $n' = (n-1)/2$)

- variation: $+(n'+1)\Delta$, at most $\pm k\Delta$, $-(n'-k)\Delta$, $1 \leq k \leq n'$

$$\text{sum of variation} = \Delta \pm k\Delta + k\Delta > 0$$

if n is even: w.l.o.g. (above $n' = n/2$, below $n' = n/2$)

- variation: $+n'\Delta$, at most $\pm k\Delta$, $-(n'-k)\Delta$, $1 \leq k \leq n'$

$$\text{sum of variation} = \pm k\Delta + k\Delta \geq 0$$

In summary, the total length of vertical pipelines increases if the main pipeline moving down from median.

The same analysis also applies to the case of moving up, with the same effect.

Outline

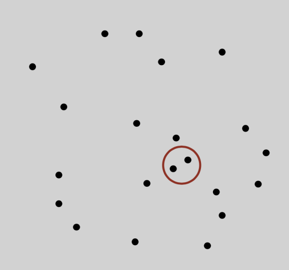
1. Introduction of Divide-and-Conquer
2. QuickSort
3. Chip Test
4. Selection Problem
   - Selecting Max and Min
   - Selecting the Second Largest
   - General Selection Problem
5. Closest Pair of Points

Finding the Closest Pair of Points

Input. Given $n > 1$ points in the plane P, find a pair of points with smallest Euclidean distance between them.

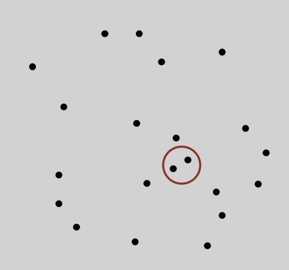
Finding the Closest Pair of Points

Input. Given $n > 1$ points in the plane P, find a pair of points with smallest Euclidean distance between them.



Finding the Closest Pair of Points

Input. Given $n > 1$ points in the plane P, find a pair of points with smallest Euclidean distance between them.



Fundamental geometric primitive.

- Graphics, computer vision, geographic information systems, molecular modeling, air traffic control.
- Special case of $\underbrace{\text{nearest neighbor, Euclidean MST, Voronoi}}$

fast closest pair inspired fast algorithms for these problems

Attempts

Brute-force. Check all C_n^2 pairs with distance calculations $\rightsquigarrow$ time complexity $\Theta(n^2)$

Attempts

Brute-force. Check all C_n^2 pairs with distance calculations $\rightsquigarrow$ time complexity $\Theta(n^2)$

1d version. $O(n \log n)$ algorithms if points are on a line (sort then compute and compare).

Attempts

Brute-force. Check all C_n^2 pairs with distance calculations $\rightsquigarrow$ time complexity $\Theta(n^2)$

1d version. $O(n \log n)$ algorithms if points are on a line (sort then compute and compare).

Non-degeneracy assumption. No two points have the same x-coordinate or y-coordinate.

Attempts

Brute-force. Check all C_n^2 pairs with distance calculations $\rightsquigarrow$ time complexity $\Theta(n^2)$

1d version. $O(n \log n)$ algorithms if points are on a line (sort then compute and compare).

Non-degeneracy assumption. No two points have the same x-coordinate or y-coordinate.

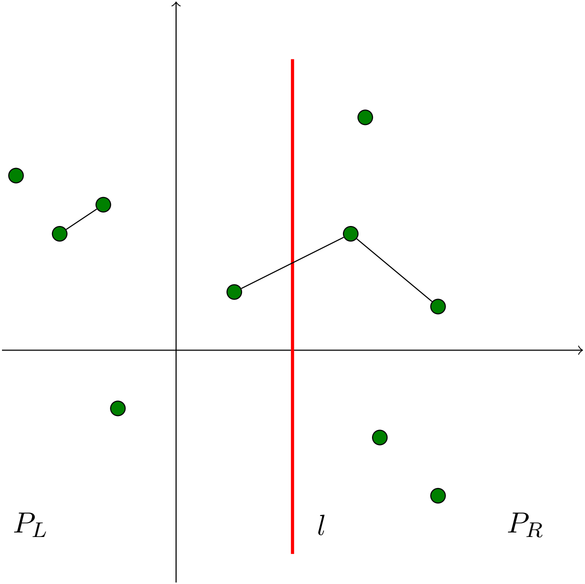
$\Theta(n^2)$ *complexity seems inevitable. Can we do better?*

Divide-and-Conquer

Idea. Partition P into P_L and P_R of roughly the same size

- Divide: draw vertical line l so that $n/2$ points on each side: P_L and P_R
- Conquer: find closest pair of points in each side recursively
- Combine: find closest pair with one point in each side.
- Return the closest one among 3 solutions.

Demo of Partition: $n = 10$



Pseudocode of MinDistance

Algorithm 13: MinDistance(P, X, Y)

Input: Points set P, coordinates set X and Y

Output: the closest pair of points and distance

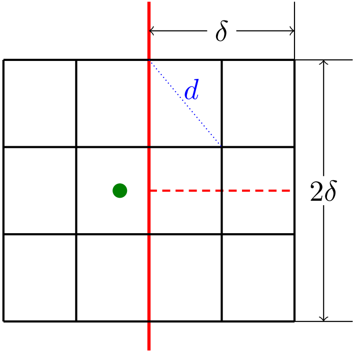
```
if |P| < 3 then direct compute;
sorted X and Y;
draw midline l to partition P into P_L and P_R;
δ_L ← MinDistance(P_L, X_L, Y_L);
δ_R ← MinDistance(P_R, X_R, Y_R);
δ = min(δ_L, δ_R);    // δ_L, δ_R are solutions to subproblems ;
check nodes within certain distance to l;
if distance is smaller than δ then update δ as this value;
```

The combine step seems again require $\Theta(n^2)$.

Step 7 requires delicated desgin and analysis.

How to find closest pair with one point in each side?

Observation 1. Only need to consider points within δ of line l.



$$d = \sqrt{(\delta/2)^2 + (2\delta/3)^2}$$
$$= \sqrt{\delta^2/4 + 4\delta^2/9}$$
$$= \sqrt{25\delta^2/36} = 5\delta/6$$

Observation 2. In each rectangles on the right side: # point ≤ 1

- each point at most is required to compare with 6 points in the opposite side (because there are at most 1 point in 1 cell)
- checking one point requires constant time $\rightsquigarrow$ compare $\Theta(n)$ points requires $\Theta(n)$ time

Treatment for Points Cross Midline

How to implement this idea? For a given point, how to find the corresponding 6 *points efficiently?*

Treatment for Points Cross Midline

How to implement this idea? For a given point, how to find the corresponding 6 points efficiently?

Sort points in 2δ-strip by their y-coordinate.

- this sorted list can be derived from sorted Y in time $\Theta(n)$ (think how?).

Treatment for Points Cross Midline

How to implement this idea? For a given point, how to find the corresponding 6 points efficiently?

Sort points in 2δ-strip by their y-coordinate.

- this sorted list can be derived from sorted Y in time $\Theta(n)$ (think how?).

Then sequentially test:

1. selects neighbors of current point within δ vertical distance: at most 7 upstairs, at most 7 downstairs
2. if its neighbor is on the same side, then skip; otherwise, compute its distance to this neighbor

Treatment for Points Cross Midline

How to implement this idea? For a given point, how to find the corresponding 6 points efficiently?

Sort points in 2δ-strip by their y-coordinate.

- this sorted list can be derived from sorted Y in time $\Theta(n)$ (think how?).

Then sequentially test:

1. selects neighbors of current point within δ vertical distance: at most 7 upstairs, at most 7 downstairs
2. if its neighbor is on the same side, then skip; otherwise, compute its distance to this neighbor

For one point, selection-then-calculation can be done in constant time, totally $\Theta(n)$ for n points

Treatment for Points Cross Midline

How to implement this idea? For a given point, how to find the corresponding 6 points efficiently?

Sort points in 2δ-strip by their y-coordinate.

- this sorted list can be derived from sorted Y in time $\Theta(n)$ (think how?).

Then sequentially test:

1. selects neighbors of current point within δ vertical distance: at most 7 upstairs, at most 7 downstairs
2. if its neighbor is on the same side, then skip; otherwise, compute its distance to this neighbor

For one point, selection-then-calculation can be done in constant time, totally $\Theta(n)$ for n points

> Why not sort two δ-strips separately instead? Cause fix one point at one side, it could be complicated to locate "the 6 neighbors" on the opposite side.

Complexity Analysis

step	operation	time complexity
1	smallest problem	$O(1)$
2	sort X and Y	$\Theta(n \log n)$
3	partition	$O(n)$
4-5	subproblems	$2W(n/2)$
6	$\delta = \min\{\delta_L, \delta_R\}$	$O(1)$
7	cross midline treatment	$\Theta(n)$

Why sorting X and Y is necessary?

- sort X: partition P to P_L and P_R
- sort Y: deal with the strip

$$\begin{cases} W(n) = 2W(n/2) + \Theta(n \log n) \\ W(n) = O(1), n \leq 3 \end{cases}$$

Applying master theorem (case 2), we have:

$$\text{recursion tree} \Rightarrow W(n) = \Theta(n \log^2 n)$$

Recap

Compared with the brute-force algorithm, the complexity of divide-and-conquer algorithm is much better.

Recap

Compared with the brute-force algorithm, the complexity of divide-and-conquer algorithm is much better.

Can we further improve it? Especially decrease the complexity of sorting.

Recap

Compared with the brute-force algorithm, the complexity of divide-and-conquer algorithm is much better.

Can we further improve it? Especially decrease the complexity of sorting.

Original approach. Re-sort the coordinates of subproblems after partition

Recap

Compared with the brute-force algorithm, the complexity of divide-and-conquer algorithm is much better.

Can we further improve it? Especially decrease the complexity of sorting.

Original approach. Re-sort the coordinates of subproblems after partition

Improved approach.

1. Preprocessing. sort X and Y before recursion
2. split sorted X and Y when partitioning, obtaining sorted X_L, Y_L for P_L, and sorted X_R, Y_R for P_R
   - splitting X is simple: split by the median
   - splitting Y: according to the split result of X

When the size of original problem is n, splitting complexity is $\Theta(n)$

Recap

Compared with the brute-force algorithm, the complexity of divide-and-conquer algorithm is much better.

Can we further improve it? Especially decrease the complexity of sorting.

Original approach. Re-sort the coordinates of subproblems after partition

Improved approach.

1. Preprocessing. sort X and Y before recursion
2. split sorted X and Y when partitioning, obtaining sorted X_L, Y_L for P_L, and sorted X_R, Y_R for P_R
    - splitting X is simple: split by the median
    - splitting Y: according to the split result of X

When the size of original problem is n, splitting complexity is $\Theta(n)$

sorting points from scratch each time $\rightsquigarrow$ sorting once and then splitting

Details of Sorting and Splitting

Data structure. two lists $X[n], Y[n]$, each element is a label i, sorted in ascending order of x, y coordinates once

Splitting of X: simple, but additional trick is needed to facilitate splitting of Y. Let n be the current size of problem

- generate an indication map H of size n, $H[i] = 0$ indicates point i in the left, $H[i] = 1$ indicates point in the right.
- time complexity is $\Theta(n)$

Splitting of Y: sequentially scanning $Y[n]$

- if $H[Y[i]] = 0$, classify $Y[i]$ to the left side, else classify it to the right side, yielding sorted Y_L and Y_R.
- time complexity is $\Theta(n)$

Demo of Splitting in Recursion

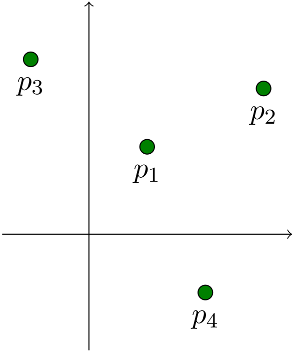


Table: Input

P	1	2	3	4
x	0.5	2	−2	1
y	2	3	4	−1

Table: Preprocessing: sort

X	3	1	4	2
Y	4	1	2	3

Table: Splitting

X_L	3	1
Y_L	1	3

Table: Splitting

X_R	4	2
Y_R	4	2

Improved Divide-and-Conquer Algorithm

$T(n)$ is the overall complexity, $\Theta(n \log n)$ is the complexity of global preprocessing, $T'(n)$ is the complexity of main recursive algorithm,

$$\begin{cases} T(n) = T'(n) + \Theta(n \log n) \\ T'(n) = 2T'(n/2) + \Theta(n) \\ T'(n) = O(1) \quad n \leq 3 \end{cases}$$

master theorem (case 2) $\Rightarrow T'(n) = \Theta(n \log n)$

Putting all the above together, $T(n) = \Theta(n \log n)$

Lower bound. In quadratic decision tree model (compute the Euclidean distance then compare), any algorithm for closest pair (even in 1D) requires $\Theta(n \log n)$ quadratic tests.