Divide and Conquer Algorithms

T. M. Murali

September 19 and 24, 2018
Divide and Conquer

- Break up a problem into several parts.
- Solve each part recursively.
- Solve base cases by brute force.
- Efficiently combine solutions for sub-problems into final solution.
Divide and Conquer

- Break up a problem into several parts.
- Solve each part recursively.
- Solve base cases by brute force.
- Efficiently combine solutions for sub-problems into final solution.
- Common use:
  - Partition problem into two equal sub-problems of size $n/2$.
  - Solve each part recursively.
  - Combine the two solutions in $O(n)$ time.
  - Resulting running time is $O(n \log n)$. 
Mergesort

Sort

**INSTANCE:** Nonempty list $L = x_1, x_2, \ldots, x_n$ of integers.

**SOLUTION:** A permutation $y_1, y_2, \ldots, y_n$ of $x_1, x_2, \ldots, x_n$ such that $y_i \leq y_{i+1}$, for all $1 \leq i < n$.

Mergesort is a divide-and-conquer algorithm for sorting.

1. Partition $L$ into two lists $A$ and $B$ of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ respectively.
2. Recursively sort $A$.
3. Recursively sort $B$.
4. Merge the sorted lists $A$ and $B$ into a single sorted list.
Merging Two Sorted Lists

- Merge two sorted lists $A = a_1, a_2, \ldots, a_k$ and $B = b_1, b_2, \ldots, b_l$.
  
  Maintain a current pointer for each list.
  
  Initialise each pointer to the front of the list.
  
  While both lists are nonempty:
  
  Let $a_i$ and $b_j$ be the elements pointed to by the current pointers.
  
  Append the smaller of the two to the output list.
  
  Advance the current pointer in the list that the smaller element belonged to.
  
EndWhile
  
Append the rest of the non-empty list to the output.

Running time of this algorithm is $O(k + l)$. 

T. M. Murali September 19 and 24, 2018 Divide and Conquer Algorithms
Merging Two Sorted Lists

- Merge two sorted lists $A = a_1, a_2, \ldots, a_k$ and $B = b_1, b_2, \ldots b_l$.
  
  Maintain a *current* pointer for each list.
  Initialise each pointer to the front of the list.
  While both lists are nonempty:
    
    Let $a_i$ and $b_j$ be the elements pointed to by the *current* pointers.
    Append the smaller of the two to the output list.
    Advance the current pointer in the list that the smaller element belonged to.
  
  EndWhile
  Append the rest of the non-empty list to the output.

- Running time of this algorithm is $O(k + l)$. 
Analysing Mergesort

1. Partition $L$ into two lists $A$ and $B$ of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ respectively.
2. Recursively sort $A$.
3. Recursively sort $B$.
4. Merge the sorted lists $A$ and $B$ into a single sorted list.
Analysing Mergesort

1. Partition $L$ into two lists $A$ and $B$ of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ respectively.
2. Recursively sort $A$.
3. Recursively sort $B$.
4. Merge the sorted lists $A$ and $B$ into a single sorted list.

Worst-case running time for $n$ elements ≤

- Worst-case running time for $\lfloor n/2 \rfloor$ elements +
- Worst-case running time for $\lceil n/2 \rceil$ elements +
- Time to split the input into two lists +
- Time to merge two sorted lists.
Analysing Mergesort

1. Partition \( L \) into two lists \( A \) and \( B \) of size \( \lfloor n/2 \rfloor \) and \( \lceil n/2 \rceil \) respectively.
2. Recursively sort \( A \).
3. Recursively sort \( B \).
4. Merge the sorted lists \( A \) and \( B \) into a single sorted list.

Worst-case running time for \( n \) elements \( \leq \)

- Worst-case running time for \( \lfloor n/2 \rfloor \) elements +
- Worst-case running time for \( \lceil n/2 \rceil \) elements +
- Time to split the input into two lists +
- Time to merge two sorted lists.

- Assume \( n \) is a power of 2.
- Define \( T(n) \equiv \) Worst-case running time for \( n \) elements, for every \( n \geq 1 \).
Analysing Mergesort

1. Partition $L$ into two lists $A$ and $B$ of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ respectively.
2. Recursively sort $A$.
3. Recursively sort $B$.
4. Merge the sorted lists $A$ and $B$ into a single sorted list.

Worst-case running time for $n$ elements $\leq$

- Worst-case running time for $\lfloor n/2 \rfloor$ elements $+$
- Worst-case running time for $\lceil n/2 \rceil$ elements $+$
- Time to split the input into two lists $+$
- Time to merge two sorted lists.

- Assume $n$ is a power of 2.
- Define $T(n) \equiv$ Worst-case running time for $n$ elements, for every $n \geq 1$.

\[ T(n) \leq 2T(n/2) + cn, \quad n > 2 \]
\[ T(2) \leq c \]
**Analysing Mergesort**

1. Partition $L$ into two lists $A$ and $B$ of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ respectively.
2. Recursively sort $A$.
3. Recursively sort $B$.
4. Merge the sorted lists $A$ and $B$ into a single sorted list.

Worst-case running time for $n$ elements $\leq$

Worst-case running time for $\lfloor n/2 \rfloor$ elements +
Worst-case running time for $\lceil n/2 \rceil$ elements +
Time to split the input into two lists +
Time to merge two sorted lists.

- Assume $n$ is a power of 2.
- Define $T(n) \equiv$ Worst-case running time for $n$ elements, for every $n \geq 1$.

\[
T(n) \leq 2T(n/2) + cn, \quad n > 2
\]
\[
T(2) \leq c
\]

- Three basic ways of solving this recurrence relation:
  1. “Unroll” the recurrence (somewhat informal method).
  2. Guess a solution and substitute into recurrence to check.
  3. Guess solution in $O()$ form and substitute into recurrence to determine the constants.
Recurrence Relations

Unrolling the recurrence

Level 0: $cn$

Level 1: $\frac{cn}{2} + \frac{cn}{2} = cn$ total

Level 2: $4(\frac{cn}{4}) = cn$ total

Figure 5.1 Unrolling the recurrence $T(n) \leq 2T(n/2) + O(n)$. 
Unrolling the recurrence

Recursion tree has log \( n \) levels.

Total work done at each level is \( cn \).

Running time of the algorithm is \( cn \log n \).

Use this method only to get an idea of the solution.
Substituting a Solution into the Recurrence

- Guess that the solution is $T(n) \leq cn \log n$ (logarithm to the base 2).
- Use induction to check if the solution satisfies the recurrence relation.
Substituting a Solution into the Recurrence

- Guess that the solution is $T(n) \leq cn \log n$ (logarithm to the base 2).
- Use induction to check if the solution satisfies the recurrence relation.
- Base case: $n = 2$. Is $T(2) = c \leq 2c \log 2$? Yes.
Substituting a Solution into the Recurrence

- Guess that the solution is $T(n) \leq cn \log n$ (logarithm to the base 2).
- Use induction to check if the solution satisfies the recurrence relation.
- Base case: $n = 2$. Is $T(2) = c \leq 2c \log 2$? Yes.
- (Strong) Inductive hypothesis: assume $T(m) \leq cm \log_2 m$ for all $m < n$. 

Why is $T(n) \leq kn^2$ a "loose" bound?

Why doesn't an attempt to prove $T(n) \leq kn$, for some $k > 0$, work?
Substituting a Solution into the Recurrence

- Guess that the solution is $T(n) \leq cn \log n$ (logarithm to the base 2).
- Use induction to check if the solution satisfies the recurrence relation.
- Base case: $n = 2$. Is $T(2) = c \leq 2c \log 2$? Yes.
- (Strong) Inductive hypothesis: assume $T(m) \leq cm \log_2 m$ for all $m < n$.
  Therefore,
  $$T(n/2) \leq (cn/2) \log(n/2).$$
Substituting a Solution into the Recurrence

- Guess that the solution is $T(n) \leq cn \log n$ (logarithm to the base 2).
- Use induction to check if the solution satisfies the recurrence relation.
- Base case: $n = 2$. Is $T(2) = c \leq 2c \log 2$? Yes.
- (Strong) Inductive hypothesis: assume $T(m) \leq cm \log_2 m$ for all $m < n$.
  Therefore,
  
  \[ T(n/2) \leq (cn/2) \log(n/2). \]

- Inductive step: Prove $T(n) \leq cn \log n$. 

T. M. Murali September 19 and 24, 2018
Divide and Conquer Algorithms
Substituting a Solution into the Recurrence

- Guess that the solution is $T(n) \leq cn \log n$ (logarithm to the base 2).
- Use induction to check if the solution satisfies the recurrence relation.
- Base case: $n = 2$. Is $T(2) = c \leq 2c \log 2$? Yes.
- (Strong) Inductive hypothesis: assume $T(m) \leq cm \log_2 m$ for all $m < n$.
  Therefore,
  \[
  T(n/2) \leq (cn/2) \log(n/2).
  \]

- Inductive step: Prove $T(n) \leq cn \log n$.
  \[
  T(n) \leq 2T\left(\frac{n}{2}\right) + cn
  \]
  \[
  \leq 2\left(\frac{cn}{2} \log \left(\frac{n}{2}\right)\right) + cn, \text{ by the inductive hypothesis}
  \]
  \[
  = cn \log \left(\frac{n}{2}\right) + cn
  \]
  \[
  = cn \log n - cn + cn
  \]
  \[
  = cn \log n.
  \]
Substituting a Solution into the Recurrence

- Guess that the solution is $T(n) \leq cn \log n$ (logarithm to the base 2).
- Use induction to check if the solution satisfies the recurrence relation.
- Base case: $n = 2$. Is $T(2) = c \leq 2c \log 2$? Yes.
- (Strong) Inductive hypothesis: assume $T(m) \leq cm \log_2 m$ for all $m < n$. Therefore,
  \[
  T(n/2) \leq \left(\frac{cn}{2}\right) \log\left(\frac{n}{2}\right).
  \]

- Inductive step: Prove $T(n) \leq cn \log n$.
  \[
  T(n) \leq 2T\left(\frac{n}{2}\right) + cn \\
  \leq 2\left(\frac{cn}{2} \log\left(\frac{n}{2}\right)\right) + cn, \text{ by the inductive hypothesis} \\
  = cn \log\left(\frac{n}{2}\right) + cn \\
  = cn \log n - cn + cn \\
  = cn \log n.
  \]

- Why is $T(n) \leq kn^2$ a “loose” bound?
Substituting a Solution into the Recurrence

- Guess that the solution is $T(n) \leq cn \log n$ (logarithm to the base 2).
- Use induction to check if the solution satisfies the recurrence relation.
- Base case: $n = 2$. Is $T(2) = c \leq 2c \log 2$? Yes.
- (Strong) Inductive hypothesis: assume $T(m) \leq cm \log_2 m$ for all $m < n$.
  Therefore,
  $$T(n/2) \leq (cn/2) \log(n/2).$$

- Inductive step: Prove $T(n) \leq cn \log n$.
  $$T(n) \leq 2T\left(\frac{n}{2}\right) + cn$$
  $$\leq 2\left(\frac{cn}{2} \log \left(\frac{n}{2}\right)\right) + cn, \text{ by the inductive hypothesis}$$
  $$= cn \log \left(\frac{n}{2}\right) + cn$$
  $$= cn \log n - cn + cn$$
  $$= cn \log n.$$

- Why is $T(n) \leq kn^2$ a “loose” bound?
- Why doesn’t an attempt to prove $T(n) \leq kn$, for some $k > 0$ work?
Partial Substitution

- Guess that the solution is $kn \log n$ (logarithm to the base 2).
- Substitute guess into the recurrence relation to check what value of $k$ will satisfy the recurrence relation.
Partial Substitution

- Guess that the solution is $kn \log n$ (logarithm to the base 2).
- Substitute guess into the recurrence relation to check what value of $k$ will satisfy the recurrence relation.
- $k \geq c$ will work.
Proof for All Values of $n$

- We assumed $n$ is a power of 2.
- How do we generalise the proof?
Proof for All Values of $n$

- We assumed $n$ is a power of 2.
- How do we generalise the proof?
- Basic axiom: $T(n) \leq T(n + 1)$, for all $n$: worst case running time increases as input size increases.
- Let $m$ be the smallest power of 2 larger than $n$.
- $T(n) \leq T(m) = O(m \log m)$
Proof for All Values of $n$

- We assumed $n$ is a power of 2.
- How do we generalise the proof?
- Basic axiom: $T(n) \leq T(n + 1)$, for all $n$: worst case running time increases as input size increases.
- Let $m$ be the smallest power of 2 larger than $n$.
- $T(n) \leq T(m) = O(m \log m) = O(n \log n)$, because $m \leq 2n$. 


Other Recurrence Relations

- Divide into $q$ sub-problems of size $n/2$ and merge in $O(n)$ time. Two distinct cases: $q = 1$ and $q > 2$.
- Divide into two sub-problems of size $n/2$ and merge in $O(n^2)$ time.
\[ T(n) = qT(n/2) + cn, \quad q = 1 \]

Each invocation reduces the problem size by a factor of 2 \( \Rightarrow \) there are \( \log n \) levels in the recursion tree.

At level \( i \) of the tree, the problem size is \( n^{1/2^i} \) and the work done is \( cn^{1/2^i} \).

Therefore, the total work done is

\[
\sum_{i=0}^{\log n} cn^{1/2^i} = O(n) .
\]

**Figure 5.3**  Unrolling the recurrence \( T(n) \leq T(n/2) + O(n) \).
\[ T(n) = q T(n/2) + cn, \quad q = 1 \]

- Each invocation reduces the problem size by a factor of 2 ⇒ there are \( \log n \) levels in the recursion tree.
- At level \( i \) of the tree, the problem size is \( n/2^i \) and the work done is \( cn/2^i \).
- Therefore, the total work done is
  \[
  \sum_{i=0}^{i=\log n} \frac{cn}{2^i} = O(n).
  \]
\( T(n) = qT(n/2) + cn, \, q > 2 \)

**Figure 5.2** Unrolling the recurrence \( T(n) \leq 3T(n/2) + O(n) \).
There are log \( n \) levels in the recursion tree.
- At level \( i \) of the tree, there are \( q^i \) sub-problems, each of size \( n/2^i \).
- The total work done at level \( i \) is \( q^i cn/2^i \). Therefore, the total work done is

\[
T(n) \leq \sum_{i=0}^{i=\log_2 n} q^i \frac{cn}{2^i} \leq
\]
There are log \( n \) levels in the recursion tree.

At level \( i \) of the tree, there are \( q^i \) sub-problems, each of size \( n/2^i \).

The total work done at level \( i \) is \( q^i cn/2^i \). Therefore, the total work done is

\[
T(n) = \sum_{i=0}^{\log_2 n} q^i \frac{cn}{2^i} \leq cn \sum_{i=0}^{\log_2 n} \left( \frac{q}{2} \right)^i
\]

\[
= O\left( cn \left( \frac{q}{2} \right)^{\log_2 n} \right) = O\left( cn \left( \frac{q}{2} \right)^{\log_2 n} \log_2 q/2 \right)
\]

\[
= O\left( cn n^{\log_2 q/2} \right) = O\left( n^{\log_2 q} \right).
\]
\[ T(n) = 2T(n/2) + cn^2 \]

- Total work done is

\[
\sum_{i=0}^{\log n} 2^i \left( \frac{cn}{2^i} \right)^2 \leq \]

\[ O(n^2) \]
\( T(n) = 2T(n/2) + cn^2 \)

- Total work done is

\[
\sum_{i=0}^{i=\log n} 2^i \left( \frac{cn}{2^i} \right)^2 \leq O(n^2).
\]
Computational Geometry

- Algorithms for geometric objects: points, lines, segments, triangles, spheres, polyhedra, …
- Started in 1975 by Shamos and Hoey.
- Problems studied have applications in a vast number of fields: ecology, molecular biology, statistics, computational finance, computer graphics, computer vision, …
Computational Geometry

- Algorithms for geometric objects: points, lines, segments, triangles, spheres, polyhedra, etc.
- Started in 1975 by Shamos and Hoey.
- Problems studied have applications in a vast number of fields: ecology, molecular biology, statistics, computational finance, computer graphics, computer vision, etc.

Closest Pair of Points

**INSTANCE:** A set $P$ of $n$ points in the plane

**SOLUTION:** The pair of points in $P$ that are the closest to each other.
Computational Geometry

- Algorithms for geometric objects: points, lines, segments, triangles, spheres, polyhedra, ... 
- Started in 1975 by Shamos and Hoey. 
- Problems studied have applications in a vast number of fields: ecology, molecular biology, statistics, computational finance, computer graphics, computer vision, ... 

Closest Pair of Points

**INSTANCE:** A set $P$ of $n$ points in the plane 

**SOLUTION:** The pair of points in $P$ that are the closest to each other. 

- At first glance, it seems any algorithm must take $\Omega(n^2)$ time. 
- Shamos and Hoey figured out an ingenious $O(n \log n)$ divide and conquer algorithm.
Closest Pair: Set-up

- Let $P = \{p_1, p_2, \ldots, p_n\}$ with $p_i = (x_i, y_i)$.
- Use $d(p_i, p_j)$ to denote the Euclidean distance between $p_i$ and $p_j$. For a specific pair of points, can compute $d(p_i, p_j)$ in $O(1)$ time.
- Goal: find the pair of points $p_i$ and $p_j$ that minimise $d(p_i, p_j)$. 

How do we solve the problem in 1D?

- **Sort**: closest pair must be adjacent in the sorted order.
- **Divide and conquer after sorting**:
  - 1 closest pair in left half: distance $\delta_l$.
  - 2 closest pair in right half: distance $\delta_r$.
  - 3 closest among pairs that span the left and right halves and are at most $\min(\delta_l, \delta_r)$ apart. How many such pairs do we need to consider?

  Just one!

Generalize the second idea to 2D.
Closest Pair: Set-up

- Let $P = \{p_1, p_2, \ldots, p_n\}$ with $p_i = (x_i, y_i)$.
- Use $d(p_i, p_j)$ to denote the Euclidean distance between $p_i$ and $p_j$. For a specific pair of points, can compute $d(p_i, p_j)$ in $O(1)$ time.
- Goal: find the pair of points $p_i$ and $p_j$ that minimise $d(p_i, p_j)$.
- How do we solve the problem in 1D?
Closest Pair: Set-up

- Let \( P = \{p_1, p_2, \ldots, p_n\} \) with \( p_i = (x_i, y_i) \).
- Use \( d(p_i, p_j) \) to denote the Euclidean distance between \( p_i \) and \( p_j \). For a specific pair of points, can compute \( d(p_i, p_j) \) in \( O(1) \) time.
- Goal: find the pair of points \( p_i \) and \( p_j \) that minimise \( d(p_i, p_j) \).
- How do we solve the problem in 1D?
  - Sort: closest pair must be adjacent in the sorted order.
Closest Pair: Set-up

- Let \( P = \{p_1, p_2, \ldots, p_n\} \) with \( p_i = (x_i, y_i) \).
- Use \( d(p_i, p_j) \) to denote the Euclidean distance between \( p_i \) and \( p_j \). For a specific pair of points, can compute \( d(p_i, p_j) \) in \( O(1) \) time.
- Goal: find the pair of points \( p_i \) and \( p_j \) that minimise \( d(p_i, p_j) \).
- How do we solve the problem in 1D?
  - Sort: closest pair must be adjacent in the sorted order.
  - Divide and conquer after sorting: closest pair must be closest of
    1. closest pair in left half: distance \( \delta_l \).
    2. closest pair in right half: distance \( \delta_r \).
    3. closest among pairs that span the left and right halves and are at most \( \min(\delta_l, \delta_r) \) apart. How many such pairs do we need to consider?
Closest Pair: Set-up

- Let $P = \{p_1, p_2, \ldots, p_n\}$ with $p_i = (x_i, y_i)$.
- Use $d(p_i, p_j)$ to denote the Euclidean distance between $p_i$ and $p_j$. For a specific pair of points, can compute $d(p_i, p_j)$ in $O(1)$ time.
- Goal: find the pair of points $p_i$ and $p_j$ that minimise $d(p_i, p_j)$.
- How do we solve the problem in 1D?
  - Sort: closest pair must be adjacent in the sorted order.
  - Divide and conquer after sorting: closest pair must be closest of
    1. closest pair in left half: distance $\delta_l$.
    2. closest pair in right half: distance $\delta_r$.
    3. closest among pairs that span the left and right halves and are at most $\min(\delta_l, \delta_r)$ apart. How many such pairs do we need to consider? Just one!

\[ \delta_Q \quad \delta_R \]

\[ \bullet \bullet \bullet \quad \bullet \quad \bullet \bullet \bullet \]
Closest Pair: Set-up

- Let \( P = \{p_1, p_2, \ldots, p_n\} \) with \( p_i = (x_i, y_i) \).
- Use \( d(p_i, p_j) \) to denote the Euclidean distance between \( p_i \) and \( p_j \). For a specific pair of points, can compute \( d(p_i, p_j) \) in \( O(1) \) time.
- Goal: find the pair of points \( p_i \) and \( p_j \) that minimise \( d(p_i, p_j) \).
- How do we solve the problem in 1D?
  - Sort: closest pair must be adjacent in the sorted order.
  - Divide and conquer after sorting: closest pair must be closest of
    1. closest pair in left half: distance \( \delta_l \).
    2. closest pair in right half: distance \( \delta_r \).
    3. closest among pairs that span the left and right halves and are at most \( \min(\delta_l, \delta_r) \) apart. How many such pairs do we need to consider? Just one!

- Generalize the second idea to 2D.
Closest Pair: Algorithm Skeleton

1. Divide $P$ into two sets $Q$ and $R$ of $n/2$ points such that each point in $Q$ has $x$-coordinate less than any point in $R$.
2. Recursively compute closest pair in $Q$ and in $R$, respectively.
**Closest Pair: Algorithm Skeleton**

1. Divide $P$ into two sets $Q$ and $R$ of $n/2$ points such that each point in $Q$ has $x$-coordinate less than any point in $R$.
2. Recursively compute closest pair in $Q$ and in $R$, respectively.
3. Let $\delta_Q$ be the distance computed for $Q$, $\delta_R$ be the distance computed for $R$, and $\delta = \min(\delta_Q, \delta_R)$.

![Diagram of points and distances $\delta_Q$ and $\delta_R$]
Closest Pair: Algorithm Skeleton

1. Divide $P$ into two sets $Q$ and $R$ of $n/2$ points such that each point in $Q$ has $x$-coordinate less than any point in $R$.

2. Recursively compute closest pair in $Q$ and in $R$, respectively.

3. Let $\delta_Q$ be the distance computed for $Q$, $\delta_R$ be the distance computed for $R$, and $\delta = \min(\delta_Q, \delta_R)$.

4. Compute pair $(q, r)$ of points such that $q \in Q$, $r \in R$, $d(q, r) < \delta$ and $d(q, r)$ is the smallest possible.
Closest Pair: Proof Sketch

- Prove by induction: Let \((s, t)\) be the closest pair.
  1. both are in \(Q\): computed correctly by recursive call.
  2. both are in \(R\): computed correctly by recursive call.
  3. one is in \(Q\) and the other is in \(R\): computed correctly in \(O(n)\) time by the procedure we will discuss.

- Strategy: Pairs of points for which we do not compute the distance between cannot be the closest pair.

- Overall running time is \(O(n \log n)\).
Closest Pair: Conquer Step

- Line $L$ passes through right-most point in $Q$.
- Let $S$ be the set of points within distance $\delta$ of $L$. (In image, $\delta = \delta_R$.)

Claim: There exist $q \in Q$, $r \in R$ such that $d(q, r) < \delta$ if and only if $q, r \in S$.

Corollary: If $t \in Q - S$ or $u \in R - S$, then $(t, u)$ cannot be the closest pair.
Closest Pair: Conquer Step

- Line $L$ passes through right-most point in $Q$.
- Let $S$ be the set of points within distance $\delta$ of $L$. (In image, $\delta = \delta_R$.)
- Claim: There exist $q \in Q$, $r \in R$ such that $d(q, r) < \delta$ if and only if $q, r \in S$. 

\[ \delta = \min(\delta_Q, \delta_R) \]
Closest Pair: Conquer Step

- Line $L$ passes through right-most point in $Q$.
- Let $S$ be the set of points within distance $\delta$ of $L$. (In image, $\delta = \delta_R$.)
- Claim: There exist $q \in Q$, $r \in R$ such that $d(q, r) < \delta$ if and only if $q, r \in S$.
- Corollary: If $t \in Q - S$ or $u \in R - S$, then $(t, u)$ cannot be the closest pair.
Closest Pair: Packing Argument

- Intuition: “too many” points in $S$ that are closer than $\delta$ to each other $\Rightarrow$ there must be a pair in $Q$ or in $R$ that are less than $\delta$ apart.
Closest Pair: Packing Argument

- Intuition: “too many” points in $S$ that are closer than $\delta$ to each other $\Rightarrow$ there must be a pair in $Q$ or in $R$ that are less than $\delta$ apart.

- Let $S_y$ denote the set of points in $S$ sorted by increasing $y$-coordinate and let $s_y$ denote the $y$-coordinate of a point $s \in S$. 

\[ \text{Claim: If there exist } s, s' \in S \text{ such that } d(s, s') < \delta \text{ then } s \text{ and } s' \text{ are at most 15 indices apart in } S_y. \]

\[ \text{Converse of the claim: If there exist } s, s' \in S \text{ such that } s' \text{ appears } 16 \text{ or more indices after } s \text{ in } S_y, \text{ then } s_y' - s_y \geq \delta. \]

Use the claim in the algorithm: For every point $s \in S_y$, compute distances only to the next 15 points in $S_y$. Other pairs of points cannot be candidates for the closest pair.
Closest Pair: Packing Argument

- Intuition: “too many” points in $S$ that are closer than $\delta$ to each other $\Rightarrow$ there must be a pair in $Q$ or in $R$ that are less than $\delta$ apart.

- Let $S_y$ denote the set of points in $S$ sorted by increasing $y$-coordinate and let $s_y$ denote the $y$-coordinate of a point $s \in S$.

- Claim: If there exist $s, s' \in S$ such that $d(s, s') < \delta$ then $s$ and $s'$ are at most 15 indices apart in $S_y$. 
Closest Pair: Packing Argument

- Intuition: “too many” points in $S$ that are closer than $\delta$ to each other $\Rightarrow$ there must be a pair in $Q$ or in $R$ that are less than $\delta$ apart.

- Let $S_y$ denote the set of points in $S$ sorted by increasing $y$-coordinate and let $s_y$ denote the $y$-coordinate of a point $s \in S$.

- Claim: If there exist $s, s' \in S$ such that $d(s, s') < \delta$ then $s$ and $s'$ are at most 15 indices apart in $S_y$.

- Converse of the claim: If there exist $s, s' \in S$ such that $s'$ appears 16 or more indices after $s$ in $S_y$, then $s'_y - s_y \geq \delta$. 
Closest Pair: Packing Argument

- Intuition: “too many” points in $S$ that are closer than $\delta$ to each other $\Rightarrow$ there must be a pair in $Q$ or in $R$ that are less than $\delta$ apart.

- Let $S_y$ denote the set of points in $S$ sorted by increasing $y$-coordinate and let $s_y$ denote the $y$-coordinate of a point $s \in S$.

- Claim: If there exist $s, s' \in S$ such that $d(s, s') < \delta$ then $s$ and $s'$ are at most 15 indices apart in $S_y$.

- Converse of the claim: If there exist $s, s' \in S$ such that $s'$ appears 16 or more indices after $s$ in $S_y$, then $s'_y - s_y \geq \delta$.

- Use the claim in the algorithm: For every point $s \in S_y$, compute distances only to the next 15 points in $S_y$.

- Other pairs of points cannot be candidates for the closest pair.
Closest Pair: Proof of Packing Argument

- Claim: If there exist $s, s' \in S$ such that $s'$ appears 16 or more indices after $s$ in $S_y$, then $s'_y - s_y \geq \delta$.

![Diagram showing closest pair proof]

Pack the plane with squares of side $\delta/2$. Each square contains at most one point. Let $s$ lie in one of the squares. Any point in the third row of the packing below $s$ has a $y$-coordinate at least $\delta$ more than $s_y$. We get a count of 12 or more indices (textbook says 16).
**Closest Pair: Proof of Packing Argument**

- Claim: If there exist \( s, s' \in S \) such that \( s' \) appears 16 or more indices after \( s \) in \( S_y \), then \( s'_y - s_y \geq \delta \).
- Pack the plane with squares of side \( \delta/2 \).
Closest Pair: Proof of Packing Argument

- Claim: If there exist $s, s' \in S$ such that $s'$ appears 16 or more indices after $s$ in $S_y$, then $s'_y - s_y \geq \delta$.
- Pack the plane with squares of side $\delta/2$.
- Each square contains at most one point.
Closest Pair: Proof of Packing Argument

- Claim: If there exist $s, s' \in S$ such that $s'$ appears 16 or more indices after $s$ in $S_y$, then $s'_y - s_y \geq \delta$.
- Pack the plane with squares of side $\delta/2$.
- Each square contains at most one point.
- Let $s$ lie in one of the squares.
Closest Pair: Proof of Packing Argument

- Claim: If there exist \( s, s' \in S \) such that \( s' \) appears 16 or more indices after \( s \) in \( S_y \), then \( s'_y - s_y \geq \delta \).

- Pack the plane with squares of side \( \delta/2 \).
- Each square contains at most one point.
- Let \( s \) lie in one of the squares.
- Any point in the third row of the packing below \( s \) has a \( y \)-coordinate at least \( \delta \) more than \( s_y \).
Claim: If there exist $s, s' \in S$ such that $s'$ appears 16 or more indices after $s$ in $S_y$, then $s'_y - s_y \geq \delta$.

Pack the plane with squares of side $\delta/2$.

Each square contains at most one point.

Let $s$ lie in one of the squares.

Any point in the third row of the packing below $s$ has a $y$-coordinate at least $\delta$ more than $s_y$.

We get a count of 12 or more indices (textbook says 16).
Closest Pair: Final Algorithm

Closest-Pair(P)
Construct $P_x$ and $P_y$ (O(n log n) time)
$q^*_x, p^*_y = $ Closest-Pair-Rec($P_x, P_y$)

Closest-Pair-Rec($P_x, P_y$)
If $|P| \leq 3$ then
  find closest pair by measuring all pairwise distances
Endif

Construct $Q_x, Q_y, R_x, R_y$ (O(n) time)
$q^*_x, q^*_y = $ Closest-Pair-Rec($Q_x, Q_y$)
$r^*_x, r^*_y = $ Closest-Pair-Rec($R_x, R_y$)

Let $x^*$ = maximum $x$-coordinate of a point in set $Q$
$L = \{(x,y) : x = x^*\}$
S = points in $P$ within distance $\delta$ of $L$.

Construct $S_y$ (O(n) time)
For each point $s \in S_y$, compute distance from $s$
  to each of next 15 points in $S_y$
  Let $s, s'$ be pair achieving minimum of these distances
  (O(n) time)

If $d(s, s') < \delta$ then
  Return $(s, s')$
Else if $d(q^*_x, q^*_y) < d(r^*_x, r^*_y)$ then
  Return $(q^*_x, q^*_y)$
Else
  Return $(r^*_x, r^*_y)$
Endif
Closest Pair: Final Algorithm

Closest-Pair($P$)

Construct $P_x$ and $P_y$ \((O(n \log n)\) time) 

\((p_0^*, p_1^*) = \text{Closest-Pair-Rec}(P_x, P_y)\)

Closest-Pair-Rec($P_x$, $P_y$)

If \(|P| \leq 3\) then 

find closest pair by measuring all pairwise distances

Endif

Construct $Q_x$, $Q_y$, $R_x$, $R_y$ \((O(n)\) time) 

\((q_0^*, q_1^*) = \text{Closest-Pair-Rec}(Q_x, Q_y)\) 

\((r_0^*, r_1^*) = \text{Closest-Pair-Rec}(R_x, R_y)\)

\[ \delta = \min(d(q_0^*, q_1^*), \ d(r_0^*, r_1^*)) \]

\[ x^* = \text{maximum } x\text{-coordinate of a point in set } Q \\]

\[ r^* = \text{point with } x^* \text{-coordinate in set } Q \]
Closest Pair: Final Algorithm

\( x^* = \) maximum \( x \)-coordinate of a point in set \( Q \)
\( L = \{ (x, y) : x = x^* \} \)
\( S = \) points in \( P \) within distance \( \delta \) of \( L \).

Construct \( S_y (O(n) \) time) 
For each point \( s \in S_y \), compute distance from \( s \)
  to each of next 15 points in \( S_y \)
  Let \( s, s' \) be pair achieving minimum of these distances
  \( (O(n) \) time)

If \( d(s, s') < \delta \) then
  Return \( (s, s') \)
Else if \( d(q_0^*, q_1^*) < d(r_0^*, r_1^*) \) then
  Return \( (q_0^*, q_1^*) \)
Else
  Return \( (r_0^*, r_1^*) \)
End if