Divide and Conquer Algorithms

T. M. Murali

March 10, 12, 2021

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.

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.

   Running time for L =

   Running time for A +
   Running time for B +
   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 $=$

Running time for A $+$
Running time for B $+$
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.

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 $n \geq 1$.

$$\begin{aligned} T(n) &\leq 2T(n/2) + cn, n > 2 \\ T(2) &\leq c \end{aligned}$$

For Homework 4, assume $T(n) = O(n \log n)$.

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 $n \geq 1$.

$$\begin{aligned} T(n) &\leq 2T(n/2) + cn, n > 2 \\ T(2) &\leq c \end{aligned}$$

For Homework 4, assume $T(n) = O(n \log n)$.

- 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. *Read from the textbook.*

Unrolling the recurrence

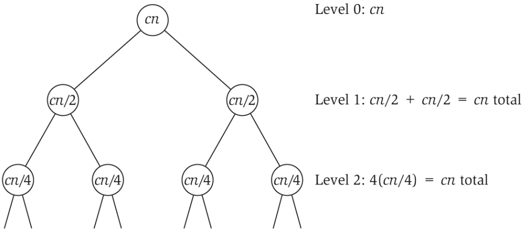


Figure 5.1 Unrolling the recurrence $T(n) \leq 2T(n/2) + O(n)$.

Unrolling the recurrence

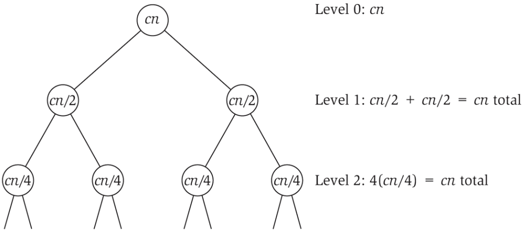


Figure 5.1 Unrolling the recurrence $T(n) \leq 2T(n/2) + O(n)$.

- Input to each sub-problem on level i has size Poll .
- Recursion tree has Poll levels.
- Number of sub-problems on level i has size Poll .

Unrolling the recurrence

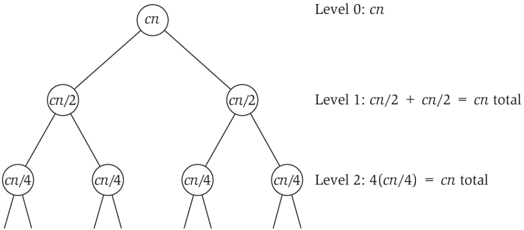


Figure 5.1 Unrolling the recurrence $T(n) \leq 2T(n/2) + O(n)$.

- Input to each sub-problem on level i has size $n/2^i$.
- Recursion tree has $\log n$ levels.
- Number of sub-problems on level i has size 2^i.

Unrolling the recurrence

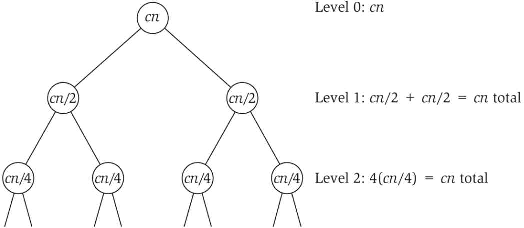


Figure 5.1 Unrolling the recurrence $T(n) \le 2T(n/2) + O(n)$.

- Input to each sub-problem on level i has size $n/2^i$.
- Recursion tree has $\log n$ levels.
- Number of sub-problems on level i has size 2^i.
- Total work done at each level is Poll .
- Running time of the algorithm is Poll .

Unrolling the recurrence

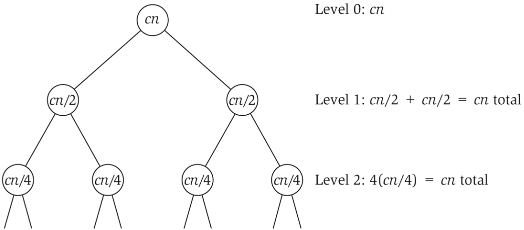


Figure 5.1 Unrolling the recurrence $T(n) \leq 2T(n/2) + O(n)$.

- Input to each sub-problem on level i has size $n/2^i$.
- Recursion tree has $\log n$ levels.
- Number of sub-problems on level i has size 2^i.
- 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.

  Inductive hypothesis: ??

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.

  Inductive hypothesis: ??

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

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.

  Inductive hypothesis: ??

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

$$T(n) \leq 2T\left(\frac{n}{2}\right) + cn, \text{ from the recurrence itself}$$

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.

  Inductive hypothesis: Must include $n/2$.

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

$$T(n) \leq 2T\left(\frac{n}{2}\right) + cn, \text{ from the recurrence itself}$$

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: Must include $n/2$.
  Assume $T(m) \leq cm \log_2 m$, for all $m < n$. Therefore,

$$T\left(\frac{n}{2}\right) \leq \frac{cn}{2} \log\left(\frac{n}{2}\right).$$

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

$$T(n) \quad \leq \quad 2T\left(\frac{n}{2}\right) + cn, \text{ from the recurrence itself}$$

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: Must include $n/2$.
  Assume $T(m) \leq cm \log_2 m$, for all $m < n$. Therefore,
  $$T\left(\frac{n}{2}\right) \leq \frac{cn}{2} \log\left(\frac{n}{2}\right).$$
- Inductive step: Prove $T(n) \leq cn \log n$.
  $$\begin{aligned} T(n) &\leq 2T\left(\frac{n}{2}\right) + cn, \text{ from the recurrence itself} \\ &\leq 2\left(\frac{cn}{2}\log\left(\frac{n}{2}\right)\right) + cn, \text{ by the inductive hypothesis} \\ &= cn \log\left(\frac{n}{2}\right) + cn \\ &= cn \log n - cn + cn \\ &= cn \log n. \end{aligned}$$

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: Must include $n/2$.
  Assume $T(m) \leq cm \log_2 m$, for all $m < n$. Therefore,

$$T\left(\frac{n}{2}\right) \leq \frac{cn}{2} \log\left(\frac{n}{2}\right).$$

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

$$\begin{aligned}
T(n) &\leq 2T\left(\frac{n}{2}\right) + cn, \text{ from the recurrence itself} \\
&\leq 2\left(\frac{cn}{2} \log\left(\frac{n}{2}\right)\right) + cn, \text{ by the inductive hypothesis} \\
&= cn \log\left(\frac{n}{2}\right) + cn \\
&= cn \log n - cn + cn \\
&= cn \log n.
\end{aligned}$$

- 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: Must include $n/2$.
  Assume $T(m) \leq cm \log_2 m$, for all $m < n$. Therefore,
  $$T\left(\frac{n}{2}\right) \leq \frac{cn}{2} \log\left(\frac{n}{2}\right).$$
- Inductive step: Prove $T(n) \leq cn \log n$.
  $$\begin{aligned} T(n) &\leq 2T\left(\frac{n}{2}\right) + cn, \text{ from the recurrence itself} \\ &\leq 2\left(\frac{cn}{2}\log\left(\frac{n}{2}\right)\right) + cn, \text{ by the inductive hypothesis} \\ &= cn \log\left(\frac{n}{2}\right) + cn \\ &= cn \log n - cn + cn \\ &= cn \log n. \end{aligned}$$
- 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?

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.

$T(n) = qT(n/2) + cn, q = 1$

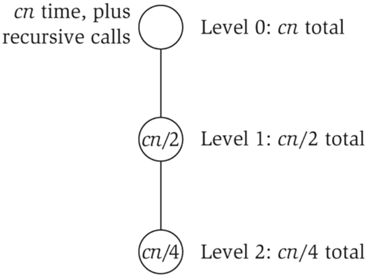


Figure 5.3 Unrolling the recurrence $T(n) \leq T(n/2) + O(n)$.

$T(n) = qT(n/2) + cn, q = 1$

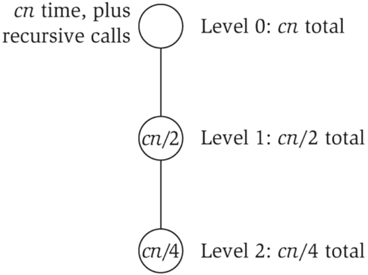


Figure 5.3 Unrolling the recurrence $T(n) \le T(n/2) + O(n)$.

- 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 $n/2^i$ and the work done is $cn/2^i$.
- Therefore, the total work done is

$$\sum_{i=0}^{i=\log n} \frac{cn}{2^i} = \text{Poll}.$$

$T(n) = qT(n/2) + cn, q = 1$

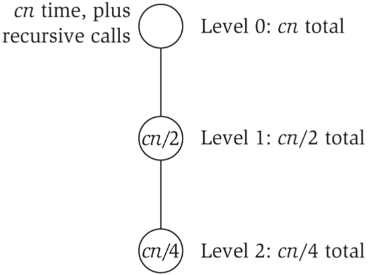


Figure 5.3 Unrolling the recurrence $T(n) \leq T(n/2) + O(n)$.

- 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 $n/2^i$ and the work done is $cn/2^i$.
- Therefore, the total work done is

$$\sum_{i=0}^{i=\log n} \frac{cn}{2^i} = O(n).$$

$T(n) = qT(n/2) + cn, q > 2$

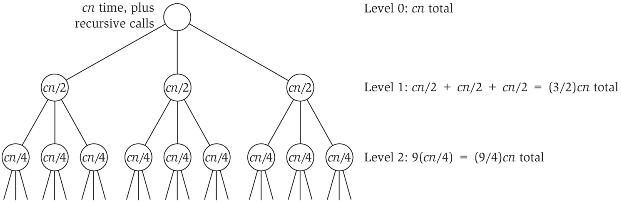


Figure 5.2 Unrolling the recurrence $T(n) \leq 3T(n/2) + O(n)$.

$T(n) = qT(n/2) + cn, q > 2$

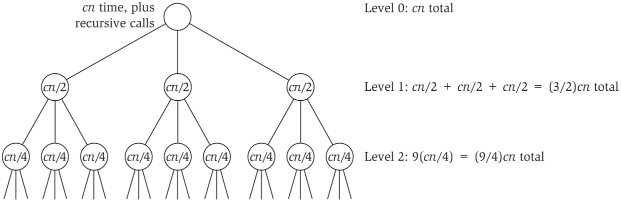


Figure 5.2 Unrolling the recurrence $T(n) \leq 3T(n/2) + O(n)$.

- 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

$$T(n) \leq \sum_{i=0}^{i=\log_2 n} q^i \frac{cn}{2^i} \leq$$

$T(n) = qT(n/2) + cn, q > 2$

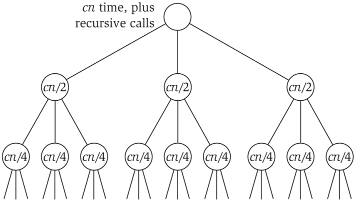


Figure 5.2 Unrolling the recurrence $T(n) \leq 3T(n/2) + O(n)$.

- 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

$$\begin{aligned} T(n) &\leq \sum_{i=0}^{i=\log_2 n} q^i \frac{cn}{2^i} \leq cn \sum_{i=0}^{i=\log_2 n} \left(\frac{q}{2}\right)^i \\ &= O\left(cn \left(\frac{q}{2}\right)^{\log_2 n}\right) = O\left(cn \left(\frac{q}{2}\right)^{(\log_{q/2} n)(\log_2 q/2)}\right) \\ &= O(cn\, n^{\log_2 q/2}) = O(n^{\log_2 q}). \end{aligned}$$

$T(n) = 2T(n/2) + cn^2$

- Total work done is

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

$T(n) = 2T(n/2) + cn^2$

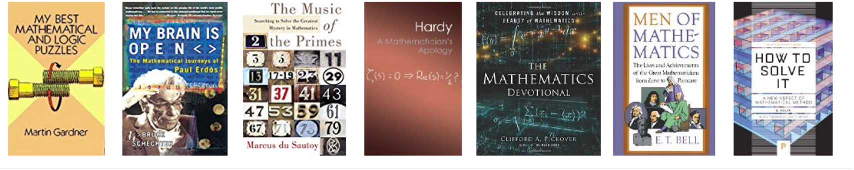
- Total work done is

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

Motivation

Inspired by your shopping trends

More top picks for you



- Collaborative filtering: match one user's preferences to those of other users, e.g., purchases, books, music.
- Meta-search engines: merge results of multiple search engines into a better search result.

Fundamental Question

- How do we compare a pair of rankings?
  - My ranking of songs: ordered list of integers from 1 to n.
  - Your ranking of songs: $a_1, a_2, \ldots, a_n$, a permutation of the integers from 1 to n.

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

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

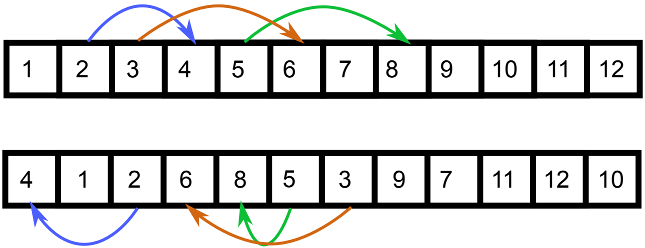
Comparing Rankings

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

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

- Suggestion: two rankings of songs are very similar if they have few inversions.

Comparing Rankings



- Suggestion: two rankings of songs are very similar if they have few inversions.
  - The second ranking has an *inversion* if there exist i, j such that $i < j$ but $a_i > a_j$.
  - The number of inversions s is a measure of the difference between the rankings.
- Question also arises in statistics: *Kendall's rank correlation* of two lists of numbers is $1 - 2s/(n(n-1))$.

Counting Inversions

Count Inversions

INSTANCE: A list $L = x_1, x_2, \ldots, x_n$ of distinct integers between 1 and n.

SOLUTION: The number of pairs $(i, j), 1 \leq i < j \leq n$ such $x_i > x_j$.

Counting Inversions

Count Inversions

INSTANCE: A list $L = x_1, x_2, \dots, x_n$ of distinct integers between 1 and n.

SOLUTION: The number of pairs $(i, j), 1 \leq i < j \leq n$ such $x_i > x_j$.

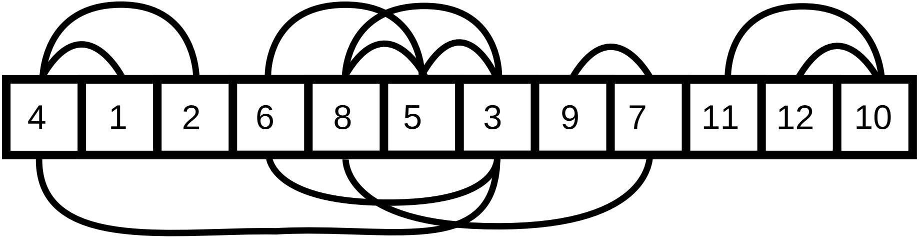
4	1	2	6	8	5	3	9	7	11	12	10

Counting Inversions

Count Inversions

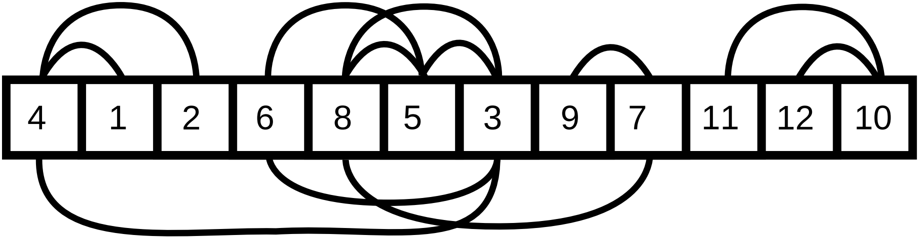
INSTANCE: A list $L = x_1, x_2, \ldots, x_n$ of distinct integers between 1 and n.

SOLUTION: The number of pairs $(i, j), 1 \leq i < j \leq n$ such $x_i > x_j$.



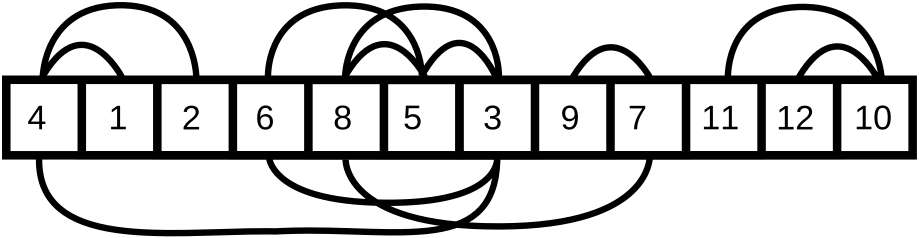
Counting Inversions: Algorithm

- How many inversions can be there in a list of n numbers? Poll



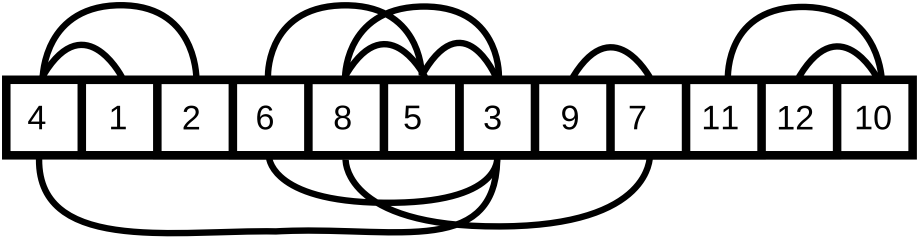
Counting Inversions: Algorithm

- How many inversions can be there in a list of n numbers? $\Omega(n^2)$. We cannot afford to compute each inversion explicitly.



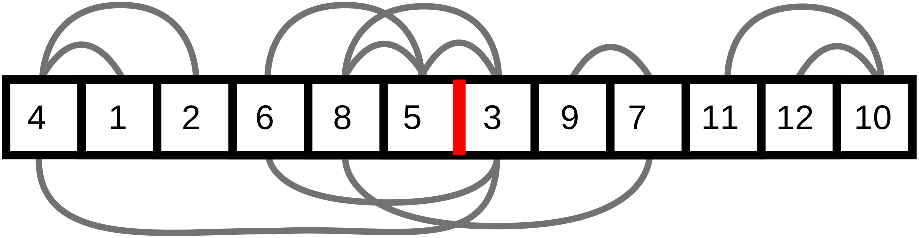
Counting Inversions: Algorithm

- How many inversions can be there in a list of n numbers? $\Omega(n^2)$. We cannot afford to compute each inversion explicitly.
- Sorting removes all inversions in $O(n \log n)$ time. Can we modify the Mergesort algorithm to count inversions?



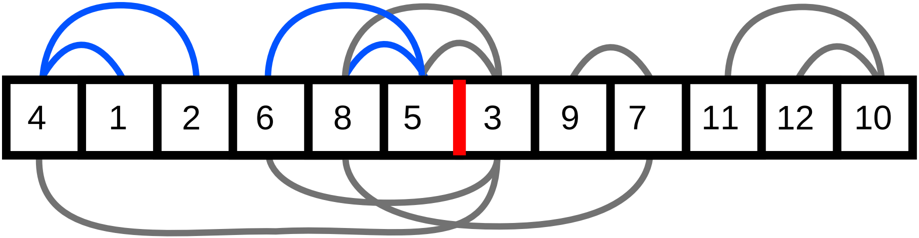
Counting Inversions: Algorithm

- How many inversions can be there in a list of n numbers? $\Omega(n^2)$. We cannot afford to compute each inversion explicitly.
- Sorting removes all inversions in $O(n \log n)$ time. Can we modify the Mergesort algorithm to count inversions?
- Candidate algorithm:
  1. Partition L into two lists A and B of size $n/2$ each.
  2. Recursively count the number of inversions in A.
  3. Recursively count the number of inversions in B.
  4. Count the number of inversions involving one element in A and one element in B.



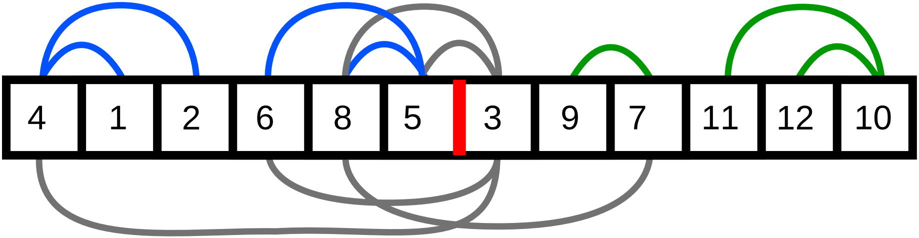
Counting Inversions: Algorithm

- How many inversions can be there in a list of n numbers? $\Omega(n^2)$. We cannot afford to compute each inversion explicitly.
- Sorting removes all inversions in $O(n \log n)$ time. Can we modify the Mergesort algorithm to count inversions?
- Candidate algorithm:
  1. Partition L into two lists A and B of size $n/2$ each.
  2. Recursively count the number of inversions in A.
  3. Recursively count the number of inversions in B.
  4. Count the number of inversions involving one element in A and one element in B.



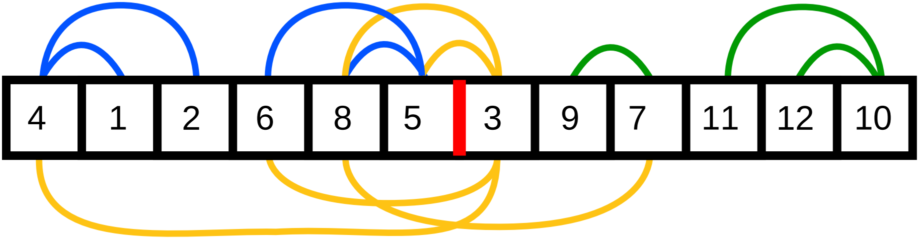
Counting Inversions: Algorithm

- How many inversions can be there in a list of n numbers? $\Omega(n^2)$. We cannot afford to compute each inversion explicitly.
- Sorting removes all inversions in $O(n \log n)$ time. Can we modify the Mergesort algorithm to count inversions?
- Candidate algorithm:
  1. Partition L into two lists A and B of size $n/2$ each.
  2. Recursively count the number of inversions in A.
  3. Recursively count the number of inversions in B.
  4. Count the number of inversions involving one element in A and one element in B.

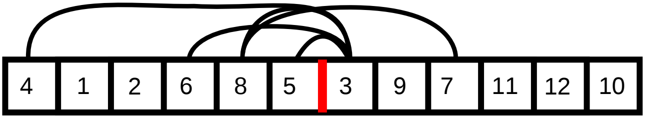


Counting Inversions: Algorithm

- How many inversions can be there in a list of n numbers? $\Omega(n^2)$. We cannot afford to compute each inversion explicitly.
- Sorting removes all inversions in $O(n \log n)$ time. Can we modify the Mergesort algorithm to count inversions?
- Candidate algorithm:
  1. Partition L into two lists A and B of size $n/2$ each.
  2. Recursively count the number of inversions in A.
  3. Recursively count the number of inversions in B.
  4. Count the number of inversions involving one element in A and one element in B.

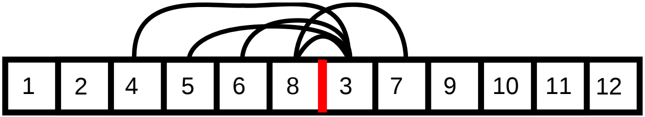


Counting Inversions: Conquer Step



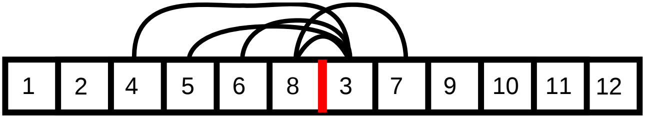
- Given lists $A = a_1, a_2, \ldots, a_m$ and $B = b_1, b_2, \ldots b_m$, compute the number of pairs a_i and b_j such $a_i > b_j$.

Counting Inversions: Conquer Step



- Given lists $A = a_1, a_2, \ldots, a_m$ and $B = b_1, b_2, \ldots b_m$, compute the number of pairs a_i and b_j such $a_i > b_j$.
- Key idea: problem is much easier if A and B are sorted!

Counting Inversions: Conquer Step

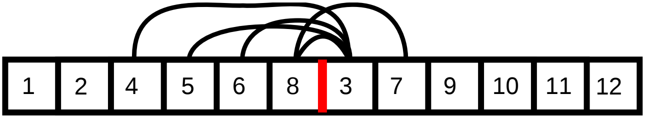


- Given lists $A = a_1, a_2, \ldots, a_m$ and $B = b_1, b_2, \ldots b_m$, compute the number of pairs a_i and b_j such $a_i > b_j$.
- Key idea: problem is much easier if A and B are sorted!
- MERGE procedure:
  1. Maintain a *current* pointer for each list.

  3. Initialise each pointer to the front of the list.
  4. While both lists are nonempty:
     1. Let a_i and b_j be the elements pointed to by the *current* pointers.
     2. Append the smaller of the two to the output list.

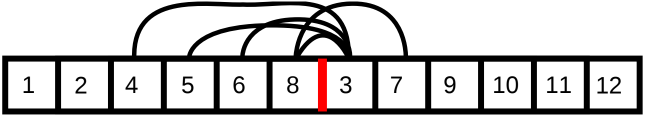
     4. Advance *current* in the list containing the smaller element.
  5. Append the rest of the non-empty list to the output.
  6. Return the merged list.

Counting Inversions: Conquer Step



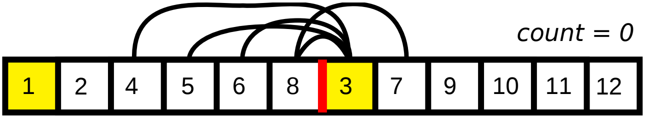
- Given lists $A = a_1, a_2, \ldots, a_m$ and $B = b_1, b_2, \ldots b_m$, compute the number of pairs a_i and b_j such $a_i > b_j$.
- Key idea: problem is much easier if A and B are sorted!
- Merge-and-Count procedure:
  1. Maintain a *current* pointer for each list.
  2. Maintain a variable *count* initialised to 0.
  3. Initialise each pointer to the front of the list.
  4. While both lists are nonempty:
     1. Let a_i and b_j be the elements pointed to by the *current* pointers.
     2. Append the smaller of the two to the output list.
     3. Do something clever in $O(1)$ time.
     4. Advance *current* in the list containing the smaller element.
  5. Append the rest of the non-empty list to the output.
  6. Return *count* and the merged list.

Counting Inversions: Conquer Step



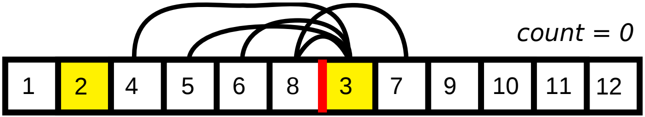
- Given lists $A = a_1, a_2, \ldots, a_m$ and $B = b_1, b_2, \ldots b_m$, compute the number of pairs a_i and b_j such $a_i > b_j$.
- Key idea: problem is much easier if A and B are sorted!
- MERGE-AND-COUNT procedure:
  1. Maintain a *current* pointer for each list.
  2. Maintain a variable *count* initialised to 0.
  3. Initialise each pointer to the front of the list.
  4. While both lists are nonempty:
     1. Let a_i and b_j be the elements pointed to by the *current* pointers.
     2. Append the smaller of the two to the output list.
     3. Do something clever in $O(1)$ time.
     4. Advance *current* in the list containing the smaller element.
  5. Append the rest of the non-empty list to the output.
  6. Return *count* and the merged list.
- Running time of this algorithm is $O(m)$.

Counting Inversions: Conquer Step



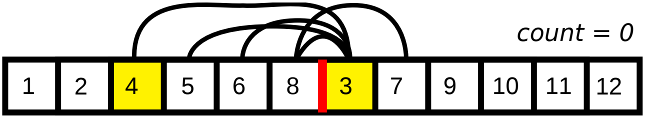
- Given lists $A = a_1, a_2, \ldots, a_m$ and $B = b_1, b_2, \ldots b_m$, compute the number of pairs a_i and b_j such $a_i > b_j$.
- Key idea: problem is much easier if A and B are sorted!
- MERGE-AND-COUNT procedure:
  1. Maintain a *current* pointer for each list.
  2. Maintain a variable *count* initialised to 0.
  3. Initialise each pointer to the front of the list.
  4. While both lists are nonempty:
     1. Let a_i and b_j be the elements pointed to by the *current* pointers.
     2. Append the smaller of the two to the output list.
     3. Do something clever in $O(1)$ time.
     4. Advance *current* in the list containing the smaller element.
  5. Append the rest of the non-empty list to the output.
  6. Return *count* and the merged list.
- Running time of this algorithm is $O(m)$.

Counting Inversions: Conquer Step



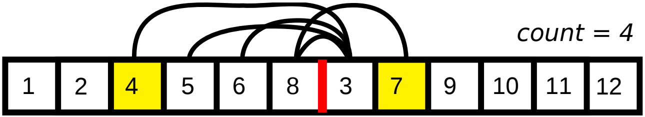
- Given lists $A = a_1, a_2, \ldots, a_m$ and $B = b_1, b_2, \ldots b_m$, compute the number of pairs a_i and b_j such $a_i > b_j$.
- Key idea: problem is much easier if A and B are sorted!
- MERGE-AND-COUNT procedure:
  1. Maintain a *current* pointer for each list.
  2. Maintain a variable *count* initialised to 0.
  3. Initialise each pointer to the front of the list.
  4. While both lists are nonempty:
     1. Let a_i and b_j be the elements pointed to by the *current* pointers.
     2. Append the smaller of the two to the output list.
     3. Do something clever in $O(1)$ time.
     4. Advance *current* in the list containing the smaller element.
  5. Append the rest of the non-empty list to the output.
  6. Return *count* and the merged list.
- Running time of this algorithm is $O(m)$.

Counting Inversions: Conquer Step



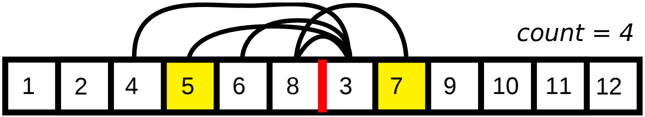
- Given lists $A = a_1, a_2, \ldots, a_m$ and $B = b_1, b_2, \ldots b_m$, compute the number of pairs a_i and b_j such $a_i > b_j$.
- Key idea: problem is much easier if A and B are sorted!
- MERGE-AND-COUNT procedure:
  1. Maintain a *current* pointer for each list.
  2. Maintain a variable *count* initialised to 0.
  3. Initialise each pointer to the front of the list.
  4. While both lists are nonempty:
     1. Let a_i and b_j be the elements pointed to by the *current* pointers.
     2. Append the smaller of the two to the output list.
     3. If $b_j < a_i$, Poll
     4. Advance *current* in the list containing the smaller element.
  5. Append the rest of the non-empty list to the output.
  6. Return *count* and the merged list.
- Running time of this algorithm is $O(m)$.

Counting Inversions: Conquer Step



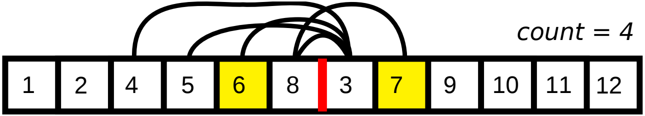
- Given lists $A = a_1, a_2, \ldots, a_m$ and $B = b_1, b_2, \ldots b_m$, compute the number of pairs a_i and b_j such $a_i > b_j$.
- Key idea: problem is much easier if A and B are sorted!
- MERGE-AND-COUNT procedure:
  1. Maintain a *current* pointer for each list.
  2. Maintain a variable *count* initialised to 0.
  3. Initialise each pointer to the front of the list.
  4. While both lists are nonempty:
     1. Let a_i and b_j be the elements pointed to by the *current* pointers.
     2. Append the smaller of the two to the output list.
     3. If $b_j < a_i$, increment *count* by the number of elements remaining in A.
     4. Advance *current* in the list containing the smaller element.
  5. Append the rest of the non-empty list to the output.
  6. Return *count* and the merged list.
- Running time of this algorithm is $O(m)$.

Counting Inversions: Conquer Step



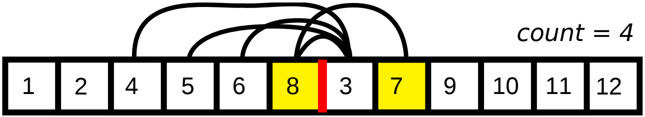
- Given lists $A = a_1, a_2, \ldots, a_m$ and $B = b_1, b_2, \ldots b_m$, compute the number of pairs a_i and b_j such $a_i > b_j$.
- Key idea: problem is much easier if A and B are sorted!
- MERGE-AND-COUNT procedure:
  1. Maintain a *current* pointer for each list.
  2. Maintain a variable *count* initialised to 0.
  3. Initialise each pointer to the front of the list.
  4. While both lists are nonempty:
     1. Let a_i and b_j be the elements pointed to by the *current* pointers.
     2. Append the smaller of the two to the output list.
     3. If $b_j < a_i$, increment *count* by the number of elements remaining in A.
     4. Advance *current* in the list containing the smaller element.
  5. Append the rest of the non-empty list to the output.
  6. Return *count* and the merged list.
- Running time of this algorithm is $O(m)$.

Counting Inversions: Conquer Step



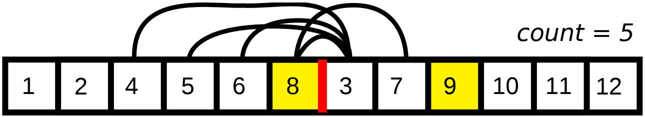
- Given lists $A = a_1, a_2, \ldots, a_m$ and $B = b_1, b_2, \ldots b_m$, compute the number of pairs a_i and b_j such $a_i > b_j$.
- Key idea: problem is much easier if A and B are sorted!
- MERGE-AND-COUNT procedure:
  1. Maintain a *current* pointer for each list.
  2. Maintain a variable *count* initialised to 0.
  3. Initialise each pointer to the front of the list.
  4. While both lists are nonempty:
     1. Let a_i and b_j be the elements pointed to by the *current* pointers.
     2. Append the smaller of the two to the output list.
     3. If $b_j < a_i$, increment *count* by the number of elements remaining in A.
     4. Advance *current* in the list containing the smaller element.
  5. Append the rest of the non-empty list to the output.
  6. Return *count* and the merged list.
- Running time of this algorithm is $O(m)$.

Counting Inversions: Conquer Step



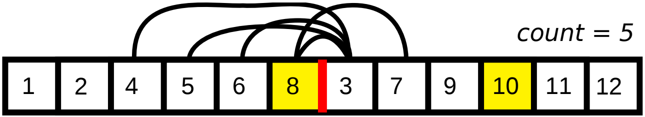
- Given lists $A = a_1, a_2, \ldots, a_m$ and $B = b_1, b_2, \ldots b_m$, compute the number of pairs a_i and b_j such $a_i > b_j$.
- Key idea: problem is much easier if A and B are sorted!
- MERGE-AND-COUNT procedure:
  1. Maintain a *current* pointer for each list.
  2. Maintain a variable *count* initialised to 0.
  3. Initialise each pointer to the front of the list.
  4. While both lists are nonempty:
     1. Let a_i and b_j be the elements pointed to by the *current* pointers.
     2. Append the smaller of the two to the output list.
     3. If $b_j < a_i$, increment *count* by the number of elements remaining in A.
     4. Advance *current* in the list containing the smaller element.
  5. Append the rest of the non-empty list to the output.
  6. Return *count* and the merged list.
- Running time of this algorithm is $O(m)$.

Counting Inversions: Conquer Step



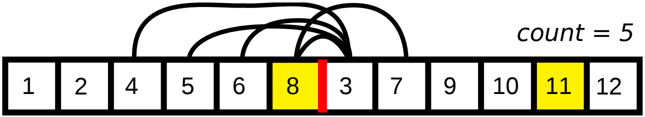
- Given lists $A = a_1, a_2, \ldots, a_m$ and $B = b_1, b_2, \ldots b_m$, compute the number of pairs a_i and b_j such $a_i > b_j$.
- Key idea: problem is much easier if A and B are sorted!
- MERGE-AND-COUNT procedure:
  1. Maintain a *current* pointer for each list.
  2. Maintain a variable *count* initialised to 0.
  3. Initialise each pointer to the front of the list.
  4. While both lists are nonempty:
     1. Let a_i and b_j be the elements pointed to by the *current* pointers.
     2. Append the smaller of the two to the output list.
     3. If $b_j < a_i$, increment *count* by the number of elements remaining in A.
     4. Advance *current* in the list containing the smaller element.
  5. Append the rest of the non-empty list to the output.
  6. Return *count* and the merged list.
- Running time of this algorithm is $O(m)$.

Counting Inversions: Conquer Step



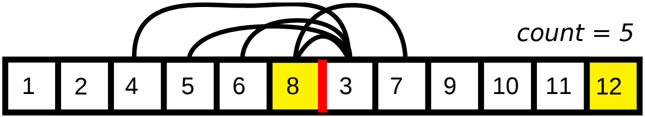
- Given lists $A = a_1, a_2, \ldots, a_m$ and $B = b_1, b_2, \ldots b_m$, compute the number of pairs a_i and b_j such $a_i > b_j$.
- Key idea: problem is much easier if A and B are sorted!
- MERGE-AND-COUNT procedure:
  1. Maintain a *current* pointer for each list.
  2. Maintain a variable *count* initialised to 0.
  3. Initialise each pointer to the front of the list.
  4. While both lists are nonempty:
     1. Let a_i and b_j be the elements pointed to by the *current* pointers.
     2. Append the smaller of the two to the output list.
     3. If $b_j < a_i$, increment *count* by the number of elements remaining in A.
     4. Advance *current* in the list containing the smaller element.
  5. Append the rest of the non-empty list to the output.
  6. Return *count* and the merged list.
- Running time of this algorithm is $O(m)$.

Counting Inversions: Conquer Step



- Given lists $A = a_1, a_2, \ldots, a_m$ and $B = b_1, b_2, \ldots b_m$, compute the number of pairs a_i and b_j such $a_i > b_j$.
- Key idea: problem is much easier if A and B are sorted!
- MERGE-AND-COUNT procedure:
  1. Maintain a *current* pointer for each list.
  2. Maintain a variable *count* initialised to 0.
  3. Initialise each pointer to the front of the list.
  4. While both lists are nonempty:
     1. Let a_i and b_j be the elements pointed to by the *current* pointers.
     2. Append the smaller of the two to the output list.
     3. If $b_j < a_i$, increment *count* by the number of elements remaining in A.
     4. Advance *current* in the list containing the smaller element.
  5. Append the rest of the non-empty list to the output.
  6. Return *count* and the merged list.
- Running time of this algorithm is $O(m)$.

Counting Inversions: Conquer Step



- Given lists $A = a_1, a_2, \ldots, a_m$ and $B = b_1, b_2, \ldots b_m$, compute the number of pairs a_i and b_j such $a_i > b_j$.
- Key idea: problem is much easier if A and B are sorted!
- MERGE-AND-COUNT procedure:
  1. Maintain a *current* pointer for each list.
  2. Maintain a variable *count* initialised to 0.
  3. Initialise each pointer to the front of the list.
  4. While both lists are nonempty:
     1. Let a_i and b_j be the elements pointed to by the *current* pointers.
     2. Append the smaller of the two to the output list.
     3. If $b_j < a_i$, increment *count* by the number of elements remaining in A.
     4. Advance *current* in the list containing the smaller element.
  5. Append the rest of the non-empty list to the output.
  6. Return *count* and the merged list.
- Running time of this algorithm is $O(m)$.

Counting Inversions: Final Algorithm

```
Sort-and-Count(L)
  If the list has one element then
      there are no inversions
  Else
      Divide the list into two halves:
         A contains the first ⌈n/2⌉ elements
         B contains the remaining ⌊n/2⌋ elements
      (r_A, A) = Sort-and-Count(A)
      (r_B, B) = Sort-and-Count(B)
      (r, L) = Merge-and-Count(A, B)
   Endif
   Return r = r_A + r_B + r, and the sorted list L
```

Counting Inversions: Final Algorithm

```
Sort-and-Count(L)
  If the list has one element then
      there are no inversions
  Else
      Divide the list into two halves:
         A contains the first ⌈n/2⌉  elements
         B contains the remaining ⌊n/2⌋ elements
      (r_A, A) = Sort-and-Count(A)
      (r_B, B) = Sort-and-Count(B)
      (r, L) = Merge-and-Count(A, B)
   Endif
   Return r = r_A + r_B + r, and the sorted list L
```

- Running time $T(n)$ of the algorithm is $O(n \log n)$ because $T(n) \leq 2T(n/2) + O(n)$.

Counting Inversions: Correctness of Sort-and-Count

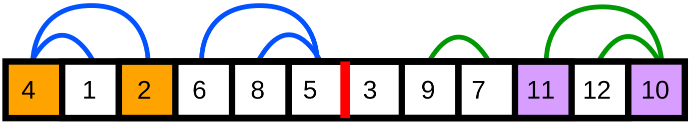
- Prove by induction. Strategy: (a) every inversion in the data is counted exactly once and (b) No non-inversion is counted.

Counting Inversions: Correctness of Sort-and-Count

- Prove by induction. Strategy: (a) every inversion in the data is counted exactly once and (b) No non-inversion is counted.
- Base case: $n = 1$.
- Inductive hypothesis: Algorithm counts number of inversions correctly for all sets of $n - 1$ or fewer numbers.
- Inductive step: Consider an arbitrary inversion, i.e., any pair k and l such that $k < l$ but $x_k > x_l$. When is this inversion counted by the algorithm?
  - $k, l \leq \lfloor n/2 \rfloor$:
  - $k, l \geq \lceil n/2 \rceil$:
  - $k \leq \lfloor n/2 \rfloor, l \geq \lceil n/2 \rceil$:

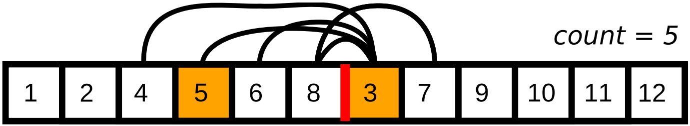
Counting Inversions: Correctness of Sort-and-Count

- Prove by induction. Strategy: (a) every inversion in the data is counted exactly once and (b) No non-inversion is counted.
- Base case: $n = 1$.
- Inductive hypothesis: Algorithm counts number of inversions correctly for all sets of $n - 1$ or fewer numbers.
- Inductive step: Consider an arbitrary inversion, i.e., any pair k and l such that $k < l$ but $x_k > x_l$. When is this inversion counted by the algorithm?
  - $k, l \leq \lfloor n/2 \rfloor$: $x_k, x_l \in A$, counted in r_A, by the inductive hypothesis.
  - $k, l \geq \lceil n/2 \rceil$: $x_k, x_l \in B$, counted in r_B, by the inductive hypothesis.
  - $k \leq \lfloor n/2 \rfloor, l \geq \lceil n/2 \rceil$:



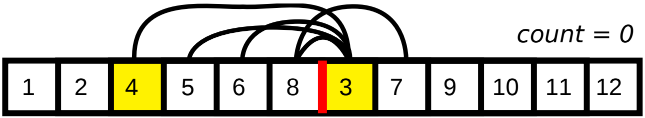
Counting Inversions: Correctness of Sort-and-Count

- Prove by induction. Strategy: (a) every inversion in the data is counted exactly once and (b) No non-inversion is counted.
- Base case: $n = 1$.
- Inductive hypothesis: Algorithm counts number of inversions correctly for all sets of $n - 1$ or fewer numbers.
- Inductive step: Consider an arbitrary inversion, i.e., any pair k and l such that $k < l$ but $x_k > x_l$. When is this inversion counted by the algorithm?
  - $k, l \leq \lfloor n/2 \rfloor$: $x_k, x_l \in A$, counted in r_A, by the inductive hypothesis.
  - $k, l \geq \lceil n/2 \rceil$: $x_k, x_l \in B$, counted in r_B, by the inductive hypothesis.
  - $k \leq \lfloor n/2 \rfloor, l \geq \lceil n/2 \rceil$: $x_k \in A, x_l \in B$. Is this inversion counted by MERGE-AND-COUNT?



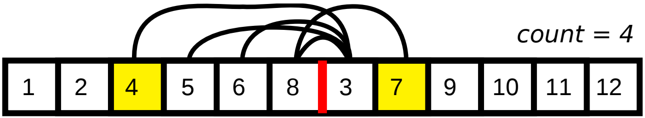
Counting Inversions: Correctness of Sort-and-Count

- Prove by induction. Strategy: (a) every inversion in the data is counted exactly once and (b) No non-inversion is counted.
- Base case: $n = 1$.
- Inductive hypothesis: Algorithm counts number of inversions correctly for all sets of $n - 1$ or fewer numbers.
- Inductive step: Consider an arbitrary inversion, i.e., any pair k and l such that $k < l$ but $x_k > x_l$. When is this inversion counted by the algorithm?
  - $k, l \leq \lfloor n/2 \rfloor$: $x_k, x_l \in A$, counted in r_A, by the inductive hypothesis.
  - $k, l \geq \lceil n/2 \rceil$: $x_k, x_l \in B$, counted in r_B, by the inductive hypothesis.
  - $k \leq \lfloor n/2 \rfloor, l \geq \lceil n/2 \rceil$: $x_k \in A, x_l \in B$. Is this inversion counted by MERGE-AND-COUNT? Yes, when x_l is output.



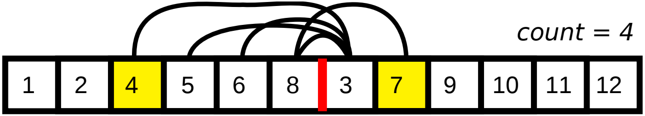
Counting Inversions: Correctness of Sort-and-Count

- Prove by induction. Strategy: (a) every inversion in the data is counted exactly once and (b) No non-inversion is counted.
- Base case: $n = 1$.
- Inductive hypothesis: Algorithm counts number of inversions correctly for all sets of $n-1$ or fewer numbers.
- Inductive step: Consider an arbitrary inversion, i.e., any pair k and l such that $k < l$ but $x_k > x_l$. When is this inversion counted by the algorithm?
  - $k, l \leq \lfloor n/2 \rfloor$: $x_k, x_l \in A$, counted in r_A, by the inductive hypothesis.
  - $k, l \geq \lceil n/2 \rceil$: $x_k, x_l \in B$, counted in r_B, by the inductive hypothesis.
  - $k \leq \lfloor n/2 \rfloor, l \geq \lceil n/2 \rceil$: $x_k \in A, x_l \in B$. Is this inversion counted by MERGE-AND-COUNT? Yes, when x_l is output.



Counting Inversions: Correctness of Sort-and-Count

- Prove by induction. Strategy: (a) every inversion in the data is counted exactly once and (b) No non-inversion is counted.
- Base case: $n = 1$.
- Inductive hypothesis: Algorithm counts number of inversions correctly for all sets of $n - 1$ or fewer numbers.
- Inductive step: Consider an arbitrary inversion, i.e., any pair k and l such that $k < l$ but $x_k > x_l$. When is this inversion counted by the algorithm?
  - $k, l \leq \lfloor n/2 \rfloor$: $x_k, x_l \in A$, counted in r_A, by the inductive hypothesis.
  - $k, l \geq \lceil n/2 \rceil$: $x_k, x_l \in B$, counted in r_B, by the inductive hypothesis.
  - $k \leq \lfloor n/2 \rfloor, l \geq \lceil n/2 \rceil$: $x_k \in A, x_l \in B$. Is this inversion counted by MERGE-AND-COUNT? Yes, when x_l is output.
  - Why is no non-inversion counted, i.e., Why does every pair counted correspond to an inversion? When x_l is output, it is smaller than all remaining elements in A, since A is sorted.



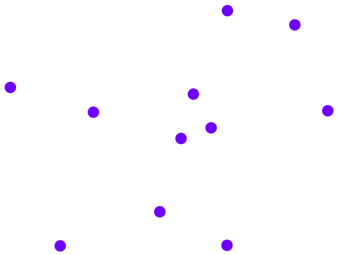
Computational Geometry

- Algorithms for geometric objects: points, lines, segments, triangles, spheres, polyhedra, ldots.
- 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, ldots.
- 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 on the Plane

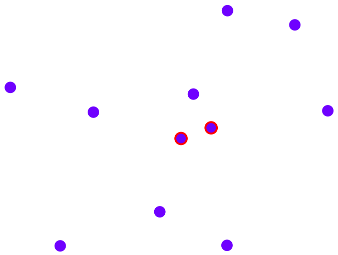


Closest Pair of Points

INSTANCE: A set P of n points in the plane

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

Closest Pair of Points on the Plane



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.

Closest Pair of Points on the Plane

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 $d(p_i, p_j)$.

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 δ_Q.
    2. closest pair in right half: distance δ_R.
    3. closest among pairs that span the left and right halves and are at most $\min(\delta_Q, \delta_R)$ apart. How many such pairs do we need to consider?

Closest Pair: Set-up

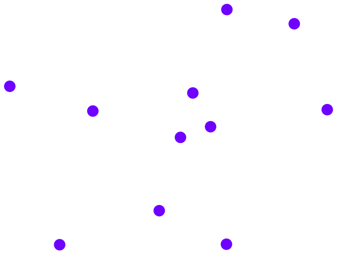
- Let $P = \{p_1, p_2, \ldots, p_n\}$ with $p_i = (x_i, y_i)$.
- Use $d(p_i, p_j)$ to denote the Euclidean distance between p_i and p_j. For a specific pair of points, can compute $d(p_i, p_j)$ in $O(1)$ time.
- Goal: find the pair of points p_i and p_j that minimise $d(p_i, p_j)$.
- How do we solve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

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 δ_Q.
    2. closest pair in right half: distance δ_R.
    3. closest among pairs that span the left and right halves and are at most $\min(\delta_Q, \delta_R)$ apart. How many such pairs do we need to consider? Just one!
- Generalize the second idea to 2D.

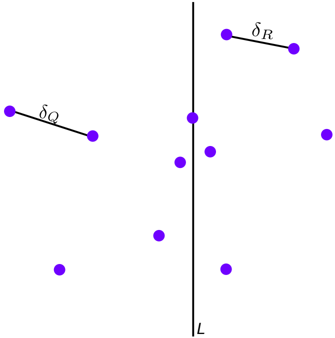
Closest Pair: Algorithm Skeleton

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.



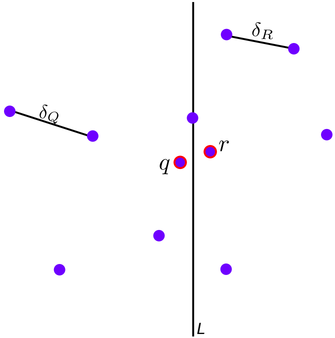
Closest Pair: Algorithm Skeleton

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



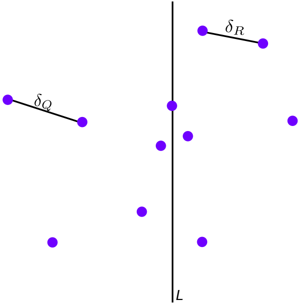
Closest Pair: Algorithm Skeleton

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 δ_Q be the distance computed for Q, δ_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.



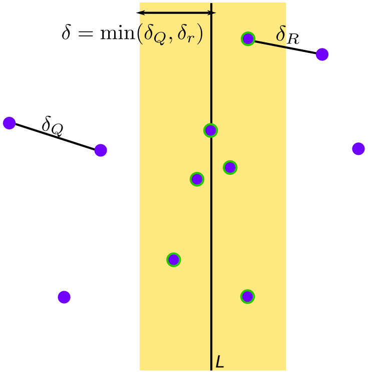
Closest Pair: Proof Sketch

- Prove by induction: Let (s, t) be the closest pair.
  1. (i) both are in Q: computed correctly by recursive call.
  2. (ii) both are in R: computed correctly by recursive call.
  3. (iii) 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 $O(n \log n)$.



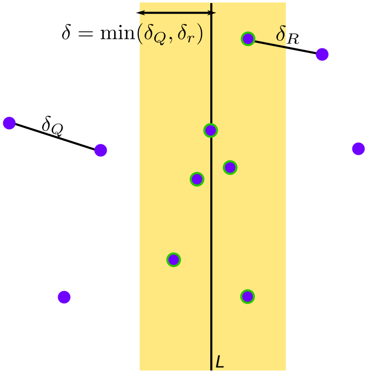
Closest Pair: Conquer Step

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



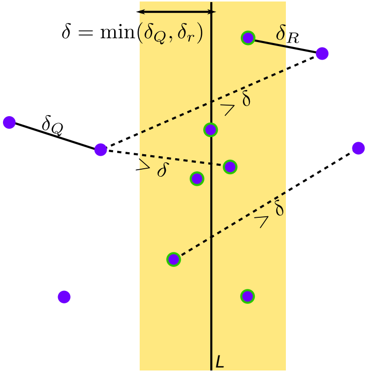
Closest Pair: Conquer Step

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



Closest Pair: Conquer Step

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

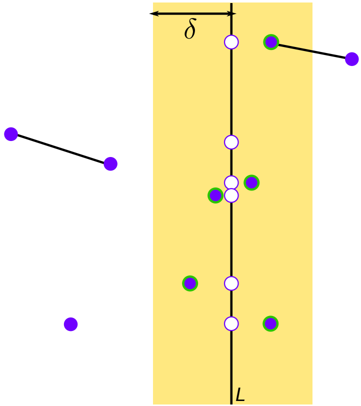


Closest Pair: Packing Argument

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

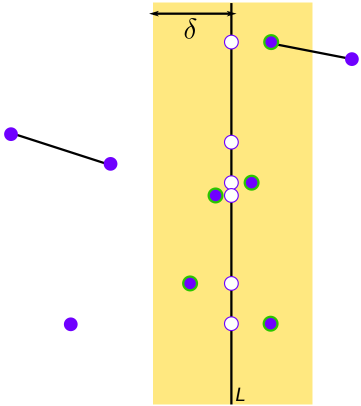
Closest Pair: Packing Argument

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



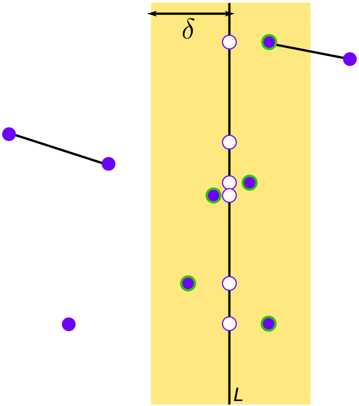
Closest Pair: Packing Argument

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



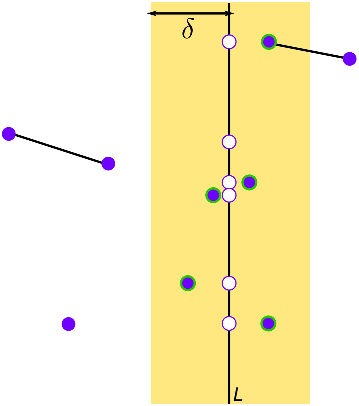
Closest Pair: Packing Argument

- Intuition: "too many" points in S that are closer than δ to each other $\Rightarrow$ there must be a pair in Q or in R that are less than δ 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 $s'_y - s_y \geq \delta$.



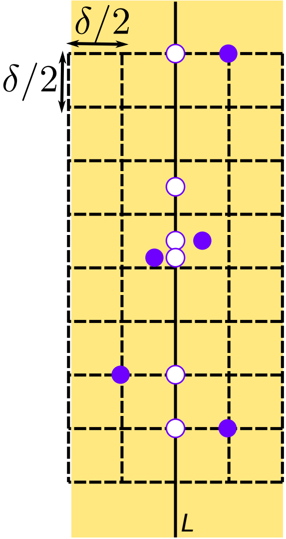
Closest Pair: Packing Argument

- Intuition: "too many" points in S that are closer than δ to each other $\Rightarrow$ there must be a pair in Q or in R that are less than δ 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

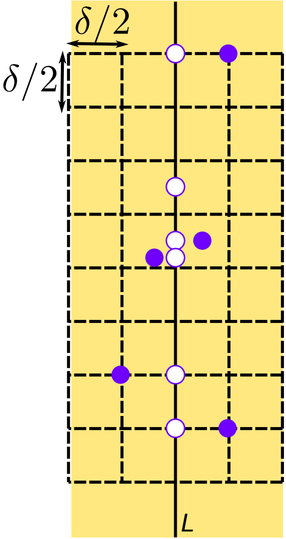
- Claim: If there exist $s, s' \in S$ such that s' appears 16 or more indices after s in S_y, then $s'_y - s_y \geq \delta$.



Closest Pair: Proof of Packing Argument

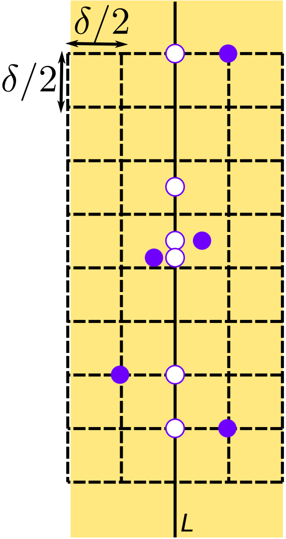
- Claim: If there exist $s, s' \in S$ such that s' appears 16 or more indices after s in S_y, then $s'_y - s_y \geq \delta$.
- Pack the plane with squares of side $\delta/2$.

Poll



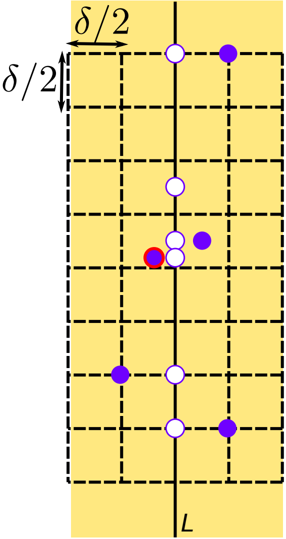
Closest Pair: Proof of Packing Argument

- 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 $\delta/2$.
  ▸ Poll
- Each square contains at most one point.



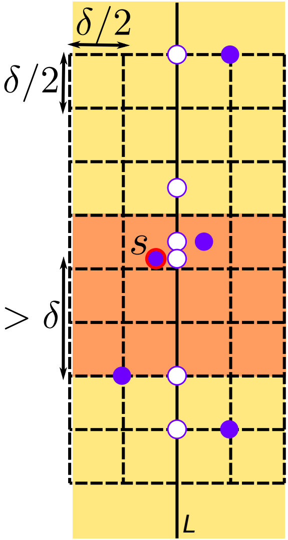
Closest Pair: Proof of Packing Argument

- 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 $\delta/2$.
  Poll
- Each square contains at most one point.
- Let s lie in one of the squares.



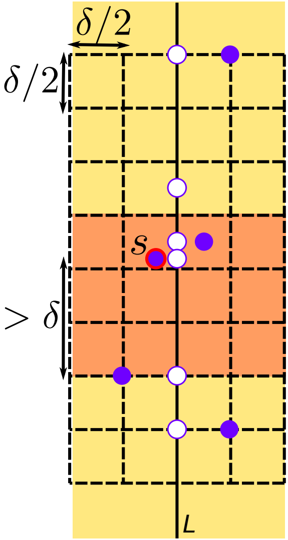
Closest Pair: Proof of Packing Argument

- 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 $\delta/2$.
  Poll
- 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 δ more than s_y.



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 $\delta/2$.
  Poll
- 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 δ 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)
  (p*_0, p*_1) = Closest-Pair-Rec(P_x,P_y)

Closest-Pair-Rec(P_x, P_y)
  If |P| ≤ 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) = Closest-Pair-Rec(Q_x, Q_y)
  (r*_0,r*_1) = Closest-Pair-Rec(R_x, R_y)

  δ = min(d(q*_0,q*_1), d(r*_0,r*_1))
  x* = maximum x-coordinate of a point in set Q
  L = {(x,y) : x = x*}
  S = points in P within distance δ of L.

  Construct S_y (O(n) time)
  For each point s ∈ 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') < δ 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)
  Endif
```

Closest Pair: Final Algorithm

```
Closest-Pair(P)
  Construct P_x and P_y   (O(n log n) time)
  (p*_0, p*_1) = Closest-Pair-Rec(P_x,P_y)

Closest-Pair-Rec(P_x, P_y)
  If |P| ≤ 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) = Closest-Pair-Rec(Q_x, Q_y)
  (r*_0,r*_1) = Closest-Pair-Rec(R_x, R_y)

  δ = min(d(q*_0,q*_1), d(r*_0,r*_1))
  x* = maximum x-coordinate of a point in set Q
  L = {(x, y) : x = x*}
```

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 δ 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^*)

Endif