Divide and Conquer Algorithms

T. M. Murali

March 3 and 15, 2016

Divide and Conquer Algorithms

- Study three divide and conquer algorithms:
  - Counting inversions.
  - Finding the closest pair of points.
  - Integer multiplication.
- First two problems use clever conquer strategies.
- Third problem uses a clever divide strategy.

Motivation

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

Motivation

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

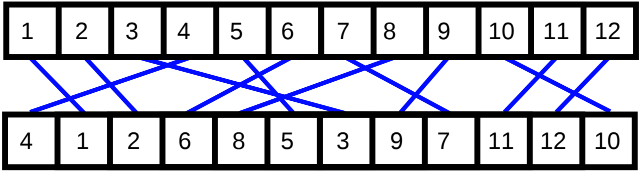
Comparing Rankings

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 $1 - 2s/\left(n(n-1)\right)$.

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

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

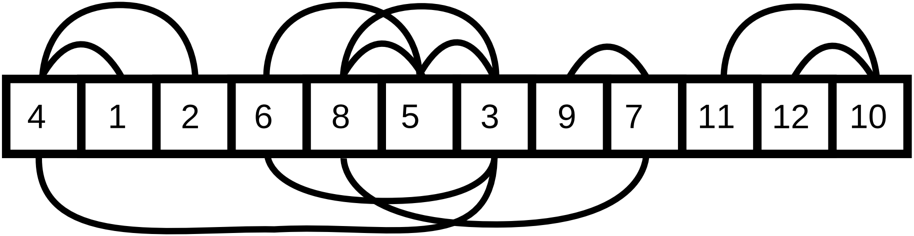
4	1	2	6	8	5	3	9	7	11	12	10

Counting Inversions

COUNT INVERSIONS

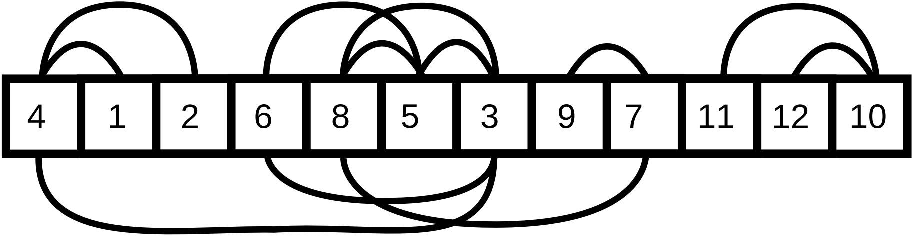
INSTANCE: A list $L = x_1, x_2, \ldots, x_n$ of distinct integers between 1 and n.

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



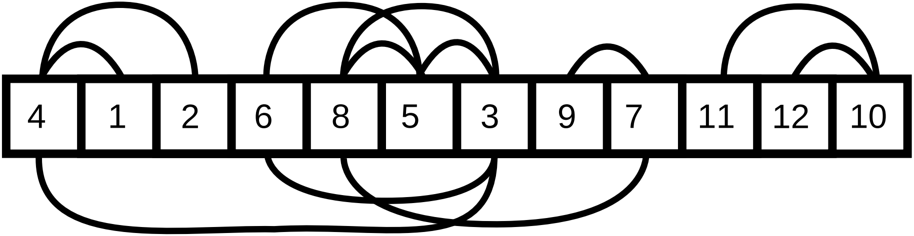
Counting Inversions: Algorithm

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



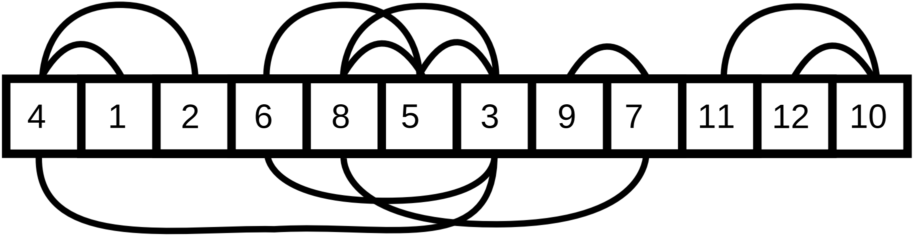
Counting Inversions: Algorithm

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



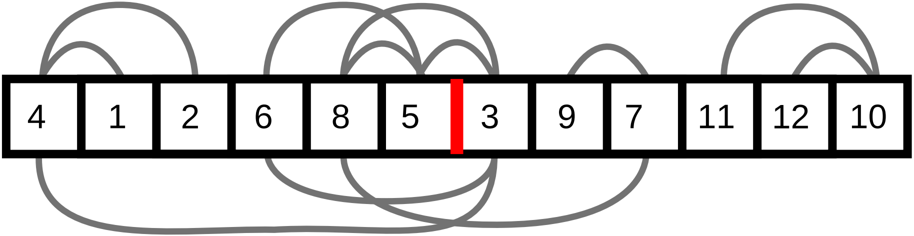
Counting Inversions: Algorithm

- 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?



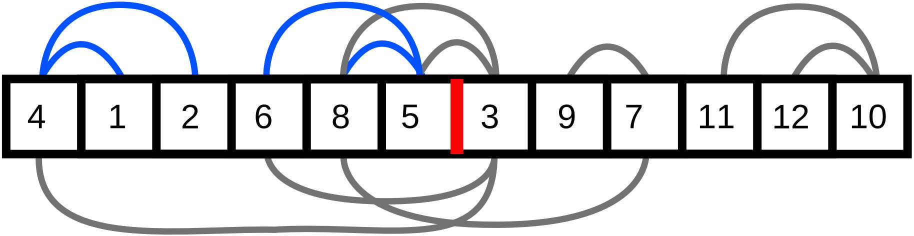
Counting Inversions: Algorithm

- 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.



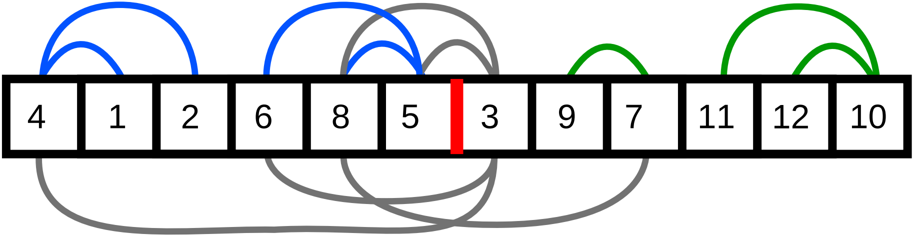
Counting Inversions: Algorithm

- 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.



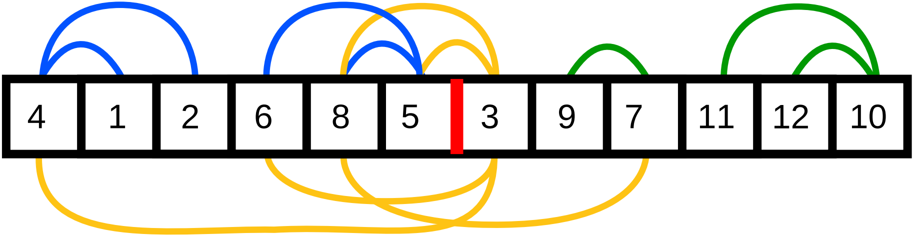
Counting Inversions: Algorithm

- 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.

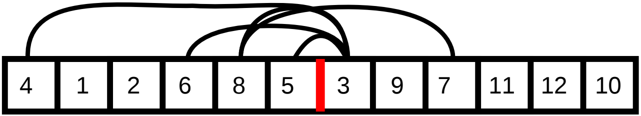


Counting Inversions: Algorithm

- 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.

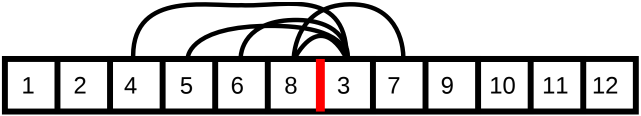


Counting Inversions: Conquer Step



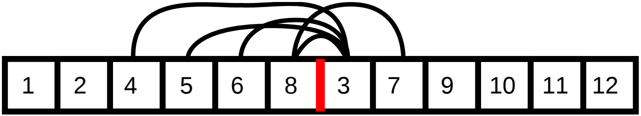
- Given lists $A = a_1, a_2, \ldots, a_m$ and $B = b_1, b_2, \ldots b_m$, compute the number of pairs a_i and b_j such $a_i > b_j$.

Counting Inversions: Conquer Step



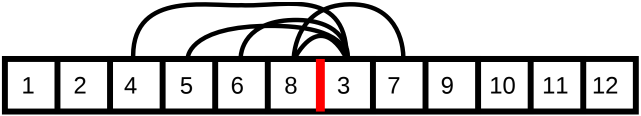
- Given lists $A = a_1, a_2, \ldots, a_m$ and $B = b_1, b_2, \ldots b_m$, compute the number of pairs a_i and b_j such $a_i > b_j$.
- Key idea: problem is much easier if A and B are sorted!

Counting Inversions: Conquer Step



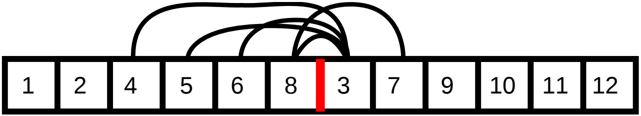
- Given lists $A = a_1, a_2, \ldots, a_m$ and $B = b_1, b_2, \ldots b_m$, compute the number of pairs a_i and b_j such $a_i > b_j$.
- Key idea: problem is much easier if A and B are sorted!
- MERGE procedure:
  1. Maintain a *current* pointer for each list.
  3. Initialise each pointer to the front of the list.
  4. While both lists are nonempty:
     4.1 Let a_i and b_j be the elements pointed to by the *current* pointers.
     4.2 Append the smaller of the two to the output list.
     4.4 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



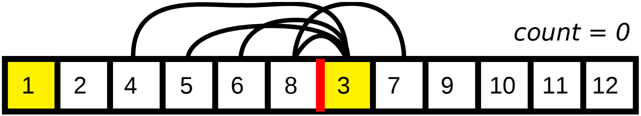
- Given lists $A = a_1, a_2, \ldots, a_m$ and $B = b_1, b_2, \ldots b_m$, compute the number of pairs a_i and b_j such $a_i > b_j$.
- Key idea: problem is much easier if A and B are sorted!
- MERGE-AND-COUNT procedure:
  1. Maintain a *current* pointer for each list.
  2. Maintain a variable *count* initialised to 0.
  3. Initialise each pointer to the front of the list.
  4. While both lists are nonempty:
     - 4.1 Let a_i and b_j be the elements pointed to by the *current* pointers.
     - 4.2 Append the smaller of the two to the output list.
     - 4.3 If $b_j < a_i$, increment *count* by the number of elements remaining in A.
     - 4.4 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



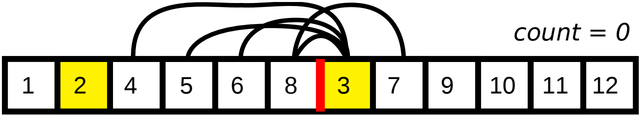
- Given lists $A = a_1, a_2, \ldots, a_m$ and $B = b_1, b_2, \ldots b_m$, compute the number of pairs a_i and b_j such $a_i > b_j$.
- Key idea: problem is much easier if A and B are sorted!
- MERGE-AND-COUNT procedure:
  1. Maintain a *current* pointer for each list.
  2. Maintain a variable *count* initialised to 0.
  3. Initialise each pointer to the front of the list.
  4. While both lists are nonempty:
     4.1 Let a_i and b_j be the elements pointed to by the *current* pointers.
     4.2 Append the smaller of the two to the output list.
     4.3 If $b_j < a_i$, increment *count* by the number of elements remaining in A.
     4.4 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



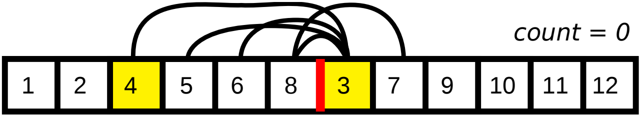
- Given lists $A = a_1, a_2, \ldots, a_m$ and $B = b_1, b_2, \ldots b_m$, compute the number of pairs a_i and b_j such $a_i > b_j$.
- Key idea: problem is much easier if A and B are sorted!
- MERGE-AND-COUNT procedure:
  1. Maintain a *current* pointer for each list.
  2. Maintain a variable *count* initialised to 0.
  3. Initialise each pointer to the front of the list.
  4. While both lists are nonempty:
     4.1 Let a_i and b_j be the elements pointed to by the *current* pointers.
     4.2 Append the smaller of the two to the output list.
     4.3 If $b_j < a_i$, increment *count* by the number of elements remaining in A.
     4.4 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



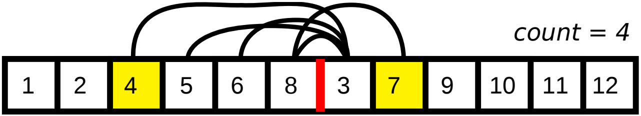
- Given lists $A = a_1, a_2, \ldots, a_m$ and $B = b_1, b_2, \ldots b_m$, compute the number of pairs a_i and b_j such $a_i > b_j$.
- Key idea: problem is much easier if A and B are sorted!
- MERGE-AND-COUNT procedure:
  1. Maintain a *current* pointer for each list.
  2. Maintain a variable *count* initialised to 0.
  3. Initialise each pointer to the front of the list.
  4. While both lists are nonempty:
     4.1 Let a_i and b_j be the elements pointed to by the *current* pointers.
     4.2 Append the smaller of the two to the output list.
     4.3 If $b_j < a_i$, increment *count* by the number of elements remaining in A.
     4.4 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



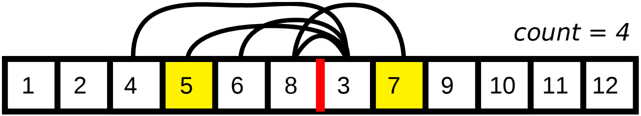
- Given lists $A = a_1, a_2, \ldots, a_m$ and $B = b_1, b_2, \ldots b_m$, compute the number of pairs a_i and b_j such $a_i > b_j$.
- Key idea: problem is much easier if A and B are sorted!
- MERGE-AND-COUNT procedure:
  1. Maintain a *current* pointer for each list.
  2. Maintain a variable *count* initialised to 0.
  3. Initialise each pointer to the front of the list.
  4. While both lists are nonempty:
     - 4.1 Let a_i and b_j be the elements pointed to by the *current* pointers.
     - 4.2 Append the smaller of the two to the output list.
     - 4.3 If $b_j < a_i$, increment *count* by the number of elements remaining in A.
     - 4.4 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



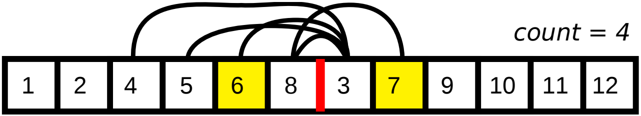
- Given lists $A = a_1, a_2, \dots, a_m$ and $B = b_1, b_2, \dots b_m$, compute the number of pairs a_i and b_j such $a_i > b_j$.
- Key idea: problem is much easier if A and B are sorted!
- MERGE-AND-COUNT procedure:
  1. Maintain a *current* pointer for each list.
  2. Maintain a variable *count* initialised to 0.
  3. Initialise each pointer to the front of the list.
  4. While both lists are nonempty:
     4.1 Let a_i and b_j be the elements pointed to by the *current* pointers.
     4.2 Append the smaller of the two to the output list.
     4.3 If $b_j < a_i$, increment *count* by the number of elements remaining in A.
     4.4 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



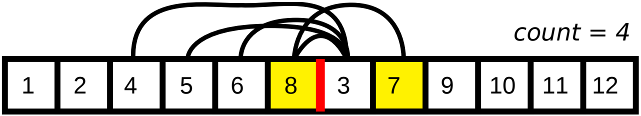
- Given lists $A = a_1, a_2, \ldots, a_m$ and $B = b_1, b_2, \ldots b_m$, compute the number of pairs a_i and b_j such $a_i > b_j$.
- Key idea: problem is much easier if A and B are sorted!
- MERGE-AND-COUNT procedure:
  1. Maintain a *current* pointer for each list.
  2. Maintain a variable *count* initialised to 0.
  3. Initialise each pointer to the front of the list.
  4. While both lists are nonempty:
     - 4.1 Let a_i and b_j be the elements pointed to by the *current* pointers.
     - 4.2 Append the smaller of the two to the output list.
     - 4.3 If $b_j < a_i$, increment *count* by the number of elements remaining in A.
     - 4.4 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



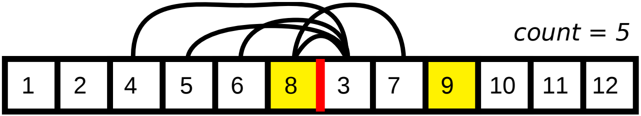
- Given lists $A = a_1, a_2, \ldots, a_m$ and $B = b_1, b_2, \ldots b_m$, compute the number of pairs a_i and b_j such $a_i > b_j$.
- Key idea: problem is much easier if A and B are sorted!
- MERGE-AND-COUNT procedure:
  1. Maintain a *current* pointer for each list.
  2. Maintain a variable *count* initialised to 0.
  3. Initialise each pointer to the front of the list.
  4. While both lists are nonempty:
     4.1 Let a_i and b_j be the elements pointed to by the *current* pointers.
     4.2 Append the smaller of the two to the output list.
     4.3 If $b_j < a_i$, increment *count* by the number of elements remaining in A.
     4.4 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



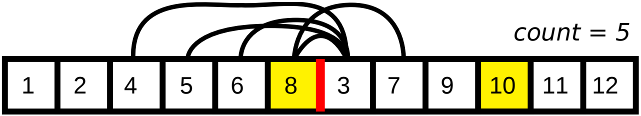
- Given lists $A = a_1, a_2, \ldots, a_m$ and $B = b_1, b_2, \ldots b_m$, compute the number of pairs a_i and b_j such $a_i > b_j$.
- Key idea: problem is much easier if A and B are sorted!
- MERGE-AND-COUNT procedure:
  1. Maintain a *current* pointer for each list.
  2. Maintain a variable *count* initialised to 0.
  3. Initialise each pointer to the front of the list.
  4. While both lists are nonempty:
     - 4.1 Let a_i and b_j be the elements pointed to by the *current* pointers.
     - 4.2 Append the smaller of the two to the output list.
     - 4.3 If $b_j < a_i$, increment *count* by the number of elements remaining in A.
     - 4.4 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



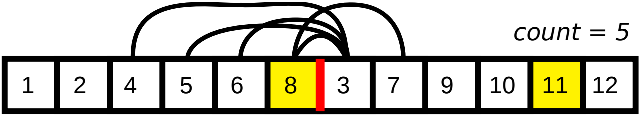
- Given lists $A = a_1, a_2, \ldots, a_m$ and $B = b_1, b_2, \ldots b_m$, compute the number of pairs a_i and b_j such $a_i > b_j$.
- Key idea: problem is much easier if A and B are sorted!
- MERGE-AND-COUNT procedure:
  1. Maintain a *current* pointer for each list.
  2. Maintain a variable *count* initialised to 0.
  3. Initialise each pointer to the front of the list.
  4. While both lists are nonempty:
     - 4.1 Let a_i and b_j be the elements pointed to by the *current* pointers.
     - 4.2 Append the smaller of the two to the output list.
     - 4.3 If $b_j < a_i$, increment *count* by the number of elements remaining in A.
     - 4.4 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



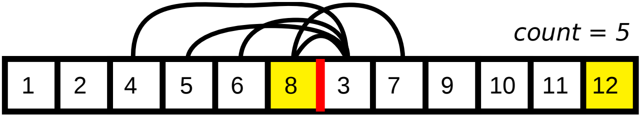
- Given lists $A = a_1, a_2, \ldots, a_m$ and $B = b_1, b_2, \ldots b_m$, compute the number of pairs a_i and b_j such $a_i > b_j$.
- Key idea: problem is much easier if A and B are sorted!
- Merge-and-Count procedure:
  1. Maintain a *current* pointer for each list.
  2. Maintain a variable *count* initialised to 0.
  3. Initialise each pointer to the front of the list.
  4. While both lists are nonempty:
     4.1 Let a_i and b_j be the elements pointed to by the *current* pointers.
     4.2 Append the smaller of the two to the output list.
     4.3 If $b_j < a_i$, increment *count* by the number of elements remaining in A.
     4.4 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:
     4.1 Let a_i and b_j be the elements pointed to by the *current* pointers.
     4.2 Append the smaller of the two to the output list.
     4.3 If $b_j < a_i$, increment *count* by the number of elements remaining in A.
     4.4 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



count = 5

- 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:
     - 4.1 Let a_i and b_j be the elements pointed to by the *current* pointers.
     - 4.2 Append the smaller of the two to the output list.
     - 4.3 If $b_j < a_i$, increment *count* by the number of elements remaining in A.
     - 4.4 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

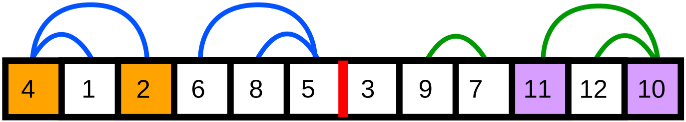
- Prove by induction. Strategy: every inversion in the data is counted exactly once.

Counting Inversions: Correctness of Sort-and-Count

- Prove by induction. Strategy: every inversion in the data is counted exactly once.
- Base case: $n = 1$.
- Inductive hypothesis: Algorithm counts number of inversions correctly for all sets of $n - 1$ or fewer numbers.
- Inductive step: Pick an arbitrary k and l such that $k < l$ but $x_k > x_l$. When is this inversion counted by the algorithm?
  - $k, l \leq \lfloor n/2 \rfloor$:
  - $k, l \geq \lceil n/2 \rceil$:
  - $k \leq \lfloor n/2 \rfloor, l \geq \lceil n/2 \rceil$:

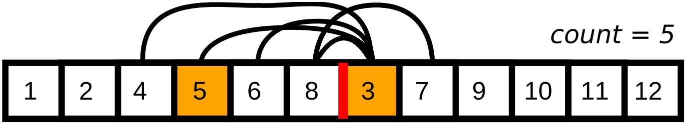
Counting Inversions: Correctness of Sort-and-Count

- Prove by induction. Strategy: every inversion in the data is counted exactly once.
- Base case: $n = 1$.
- Inductive hypothesis: Algorithm counts number of inversions correctly for all sets of $n - 1$ or fewer numbers.
- Inductive step: Pick an arbitrary 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.
  - $k, l \geq \lceil n/2 \rceil$: $x_k, x_l \in B$, counted in r_B.
  - $k \leq \lfloor n/2 \rfloor, l \geq \lceil n/2 \rceil$:



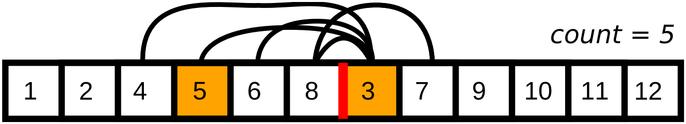
Counting Inversions: Correctness of Sort-and-Count

- Prove by induction. Strategy: every inversion in the data is counted exactly once.
- Base case: $n = 1$.
- Inductive hypothesis: Algorithm counts number of inversions correctly for all sets of $n-1$ or fewer numbers.
- Inductive step: Pick an arbitrary 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.
  - $k, l \geq \lceil n/2 \rceil$: $x_k, x_l \in B$, counted in r_B.
  - $k \leq \lfloor n/2 \rfloor, l \geq \lceil n/2 \rceil$: $x_k \in A, x_l \in B$. Is this inversion counted by MERGE-AND-COUNT?



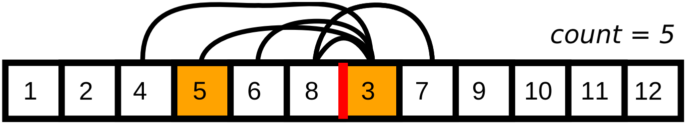
Counting Inversions: Correctness of Sort-and-Count

- Prove by induction. Strategy: every inversion in the data is counted exactly once.
- Base case: $n = 1$.
- Inductive hypothesis: Algorithm counts number of inversions correctly for all sets of $n - 1$ or fewer numbers.
- Inductive step: Pick an arbitrary 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.
  - $k, l \geq \lceil n/2 \rceil$: $x_k, x_l \in B$, counted in r_B.
  - $k \leq \lfloor n/2 \rfloor, l \geq \lceil n/2 \rceil$: $x_k \in A, x_l \in B$. Is this inversion counted by MERGE-AND-COUNT? Yes, when x_l is output.



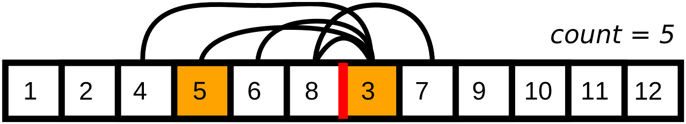
Counting Inversions: Correctness of Sort-and-Count

- Prove by induction. Strategy: every inversion in the data is counted exactly once.
- Base case: $n = 1$.
- Inductive hypothesis: Algorithm counts number of inversions correctly for all sets of $n-1$ or fewer numbers.
- Inductive step: Pick an arbitrary 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.
  - $k, l \geq \lceil n/2 \rceil$: $x_k, x_l \in B$, counted in r_B.
  - $k \leq \lfloor n/2 \rfloor, l \geq \lceil n/2 \rceil$: $x_k \in A, x_l \in B$. 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?



Counting Inversions: Correctness of Sort-and-Count

- Prove by induction. Strategy: every inversion in the data is counted exactly once.
- Base case: $n = 1$.
- Inductive hypothesis: Algorithm counts number of inversions correctly for all sets of $n-1$ or fewer numbers.
- Inductive step: Pick an arbitrary 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.
  - $k, l \geq \lceil n/2 \rceil$: $x_k, x_l \in B$, counted in r_B.
  - $k \leq \lfloor n/2 \rfloor, l \geq \lceil n/2 \rceil$: $x_k \in A, x_l \in B$. 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.



Integer Multiplication

MULTIPLY INTEGERS
INSTANCE: Two n-digit binary integers x and y
SOLUTION: The product xy

Integer Multiplication

MULTIPLY INTEGERS
INSTANCE: Two n-digit binary integers x and y
SOLUTION: The product xy

- Multiply two n-digit integers.

Integer Multiplication

MULTIPLY INTEGERS
INSTANCE: Two n-digit binary integers x and y
SOLUTION: The product xy

- Multiply two n-digit integers.
- Result has at most $2n$ digits.

Integer Multiplication

MULTIPLY INTEGERS
INSTANCE: Two n-digit binary integers x and y
SOLUTION: The product xy

- Multiply two n-digit integers.
- Result has at most $2n$ digits.
- Algorithm we learnt in school takes

```
                  1100
                × 1101
                ------
                  1100
   12             0000
 × 13            1100
 ----           1100
   36          --------
  12           10011100
 ----
  156
 (a)              (b)
```

Figure 5.8 The elementary-school algorithm for multiplying two integers, in (a) decimal and (b) binary representation.

Integer Multiplication

MULTIPLY INTEGERS
INSTANCE: Two n-digit binary integers x and y
SOLUTION: The product xy

- Multiply two n-digit integers.
- Result has at most $2n$ digits.
- Algorithm we learnt in school takes $O(n^2)$ operations. Size of the input is not 2 but $2n$,

```
                 1100
               × 1101
               ------
                 1100
   12           0000
 × 13          1100
 ----         1100
   36        --------
  12         10011100
 ----
  156

  (a)           (b)
```

Figure 5.8 The elementary-school algorithm for multiplying two integers, in (a) decimal and (b) binary representation.

Divide-and-Conquer Algorithm

- Assume integers are binary.
- Let us use divide and conquer

Divide-and-Conquer Algorithm

- Assume integers are binary.
- Let us use divide and conquer by splitting each number into first $n/2$ bits and last $n/2$ bits.
- Let x be split into x_0 (lower-order bits) and x_1 (higher-order bits) and y into y_0 (lower-order bits) and y_1 (higher-order bits).

$$xy =$$

Divide-and-Conquer Algorithm

- Assume integers are binary.
- Let us use divide and conquer by splitting each number into first $n/2$ bits and last $n/2$ bits.
- Let x be split into x_0 (lower-order bits) and x_1 (higher-order bits) and y into y_0 (lower-order bits) and y_1 (higher-order bits).

$$\begin{aligned} xy &= (x_1 2^{n/2} + x_0)(y_1 2^{n/2} + y_0) \\ &= x_1 y_1 2^n + (x_1 y_0 + x_0 y_1) 2^{n/2} + x_0 y_0. \end{aligned}$$

Divide-and-Conquer Algorithm

- Assume integers are binary.
- Let us use divide and conquer by splitting each number into first $n/2$ bits and last $n/2$ bits.
- Let x be split into x_0 (lower-order bits) and x_1 (higher-order bits) and y into y_0 (lower-order bits) and y_1 (higher-order bits).

$$\begin{aligned} xy &= (x_1 2^{n/2} + x_0)(y_1 2^{n/2} + y_0) \\ &= x_1 y_1 2^n + (x_1 y_0 + x_0 y_1) 2^{n/2} + x_0 y_0. \end{aligned}$$

- Algorithm: each of x_1, x_0, y_1, y_0 has $n/2$ bits, so we can compute $x_1 y_1$, $x_1 y_0$, $x_0 y_1$, and $x_0 y_0$ recursively, and merge the answers in $O(n)$ time.

Divide-and-Conquer Algorithm

- Assume integers are binary.
- Let us use divide and conquer by splitting each number into first $n/2$ bits and last $n/2$ bits.
- Let x be split into x_0 (lower-order bits) and x_1 (higher-order bits) and y into y_0 (lower-order bits) and y_1 (higher-order bits).

$$\begin{aligned} xy &= (x_1 2^{n/2} + x_0)(y_1 2^{n/2} + y_0) \\ &= x_1 y_1 2^n + (x_1 y_0 + x_0 y_1) 2^{n/2} + x_0 y_0. \end{aligned}$$

- Algorithm: each of x_1, x_0, y_1, y_0 has $n/2$ bits, so we can compute $x_1 y_1$, $x_1 y_0$, $x_0 y_1$, and $x_0 y_0$ recursively, and merge the answers in $O(n)$ time.
- What is the running time $T(n)$?

Divide-and-Conquer Algorithm

- Assume integers are binary.
- Let us use divide and conquer by splitting each number into first $n/2$ bits and last $n/2$ bits.
- Let x be split into x_0 (lower-order bits) and x_1 (higher-order bits) and y into y_0 (lower-order bits) and y_1 (higher-order bits).

$$\begin{aligned} xy &= (x_1 2^{n/2} + x_0)(y_1 2^{n/2} + y_0) \\ &= x_1 y_1 2^n + (x_1 y_0 + x_0 y_1) 2^{n/2} + x_0 y_0. \end{aligned}$$

- Algorithm: each of x_1, x_0, y_1, y_0 has $n/2$ bits, so we can compute $x_1 y_1$, $x_1 y_0$, $x_0 y_1$, and $x_0 y_0$ recursively, and merge the answers in $O(n)$ time.
- What is the running time $T(n)$?

$$T(n) \leq 4T(n/2) + cn$$

Divide-and-Conquer Algorithm

- Assume integers are binary.
- Let us use divide and conquer by splitting each number into first $n/2$ bits and last $n/2$ bits.
- Let x be split into x_0 (lower-order bits) and x_1 (higher-order bits) and y into y_0 (lower-order bits) and y_1 (higher-order bits).

$$\begin{aligned} xy &= (x_1 2^{n/2} + x_0)(y_1 2^{n/2} + y_0) \\ &= x_1 y_1 2^n + (x_1 y_0 + x_0 y_1) 2^{n/2} + x_0 y_0. \end{aligned}$$

- Algorithm: each of x_1, x_0, y_1, y_0 has $n/2$ bits, so we can compute $x_1 y_1$, $x_1 y_0$, $x_0 y_1$, and $x_0 y_0$ recursively, and merge the answers in $O(n)$ time.
- What is the running time $T(n)$?

$$T(n) \le 4T(n/2) + cn \le O(n^2)$$

Improving the Algorithm

- Four sub-problems lead to an $O(n^2)$ algorithm.
- How can we reduce the number of sub-problems?

Improving the Algorithm

- Four sub-problems lead to an $O(n^2)$ algorithm.
- How can we reduce the number of sub-problems?
  - We do not need to compute $x_1 y_0$ and $x_0 y_1$ independently; we just need their sum.

Improving the Algorithm

- Four sub-problems lead to an $O(n^2)$ algorithm.
- How can we reduce the number of sub-problems?
  - We do not need to compute $x_1 y_0$ and $x_0 y_1$ independently; we just need their sum.
  - $x_1 y_1 + (x_1 y_0 + x_0 y_1) + x_0 y_0 = (x_0 + x_1)(y_0 + y_1)$
  - Compute $x_1 y_1$, $x_0 y_0$ and $(x_0 + x_1)(y_0 + y_1)$ recursively and then compute $(x_1 y_0 + x_0 y_1)$ by subtraction.
  - We have three sub-problems of size $n/2$.
  - Strategy: simple arithmetic manipulations.
- What is the running time $T(n)$?

Improving the Algorithm

- Four sub-problems lead to an $O(n^2)$ algorithm.
- How can we reduce the number of sub-problems?
  - We do not need to compute $x_1 y_0$ and $x_0 y_1$ independently; we just need their sum.
  - $x_1 y_1 + (x_1 y_0 + x_0 y_1) + x_0 y_0 = (x_0 + x_1)(y_0 + y_1)$
  - Compute $x_1 y_1$, $x_0 y_0$ and $(x_0 + x_1)(y_0 + y_1)$ recursively and then compute $(x_1 y_0 + x_0 y_1)$ by subtraction.
  - We have three sub-problems of size $n/2$.
  - Strategy: simple arithmetic manipulations.
- What is the running time $T(n)$?

$$T(n) \leq 3T(n/2) + cn$$

Improving the Algorithm

- Four sub-problems lead to an $O(n^2)$ algorithm.
- How can we reduce the number of sub-problems?
  - We do not need to compute x_1y_0 and x_0y_1 independently; we just need their sum.
  - $x_1y_1 + (x_1y_0 + x_0y_1) + x_0y_0 = (x_0 + x_1)(y_0 + y_1)$
  - Compute x_1y_1, x_0y_0 and $(x_0 + x_1)(y_0 + y_1)$ recursively and then compute $(x_1y_0 + x_0y_1)$ by subtraction.
  - We have three sub-problems of size $n/2$.
  - Strategy: simple arithmetic manipulations.
- What is the running time $T(n)$?

$$\begin{aligned} T(n) &\leq 3T(n/2) + cn \\ &\leq O(n^{\log_2 3}) = O(n^{1.59}) \end{aligned}$$

Final Algorithm

```
Recursive-Multiply(x,y):
  Write x = x_1 · 2^{n/2} + x_0
        y = y_1 · 2^{n/2} + y_0
  Compute x_1 + x_0 and y_1 + y_0
  p = Recursive-Multiply(x_1 + x_0, y_1 + y_0)
  x_1y_1 = Recursive-Multiply(x_1, y_1)
  x_0y_0 = Recursive-Multiply(x_0, y_0)
  Return x_1y_1 · 2^n + (p − x_1y_1 − x_0y_0) · 2^{n/2} + x_0y_0
```

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

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, 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

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

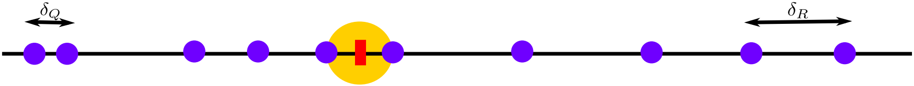
- Let $P = \{p_1, p_2, \ldots, p_n\}$ with $p_i = (x_i, y_i)$.
- Use $d(p_i, p_j)$ to denote the Euclidean distance between p_i and p_j. For a specific pair of points, can compute $d(p_i, p_j)$ in $O(1)$ time.
- Goal: find the pair of points p_i and p_j that minimise $d(p_i, p_j)$.
- How do we solve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

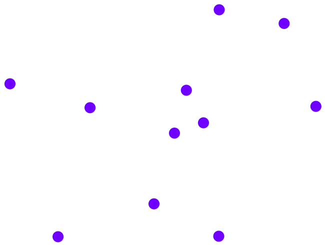
Closest Pair: Set-up

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



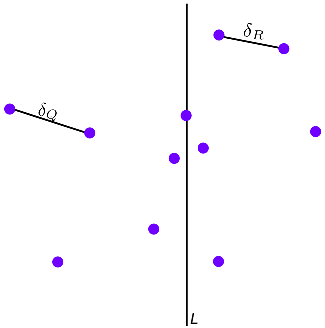
Closest Pair: Algorithm Skeleton

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.



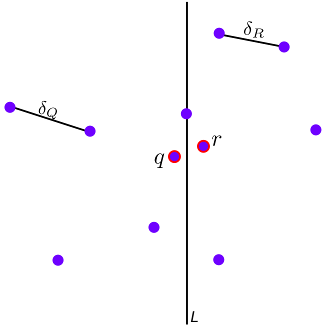
Closest Pair: Algorithm Skeleton

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



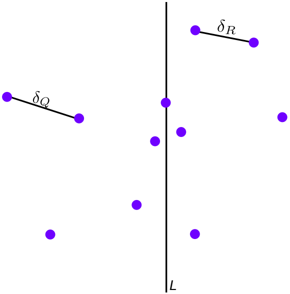
Closest Pair: Algorithm Skeleton

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.



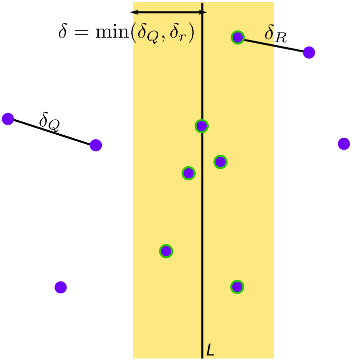
Closest Pair: Proof Sketch

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



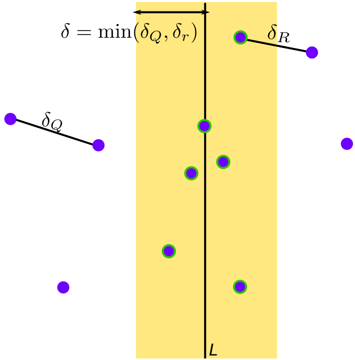
Closest Pair: Conquer Step

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



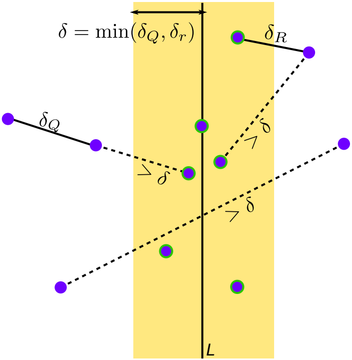
Closest Pair: Conquer Step

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

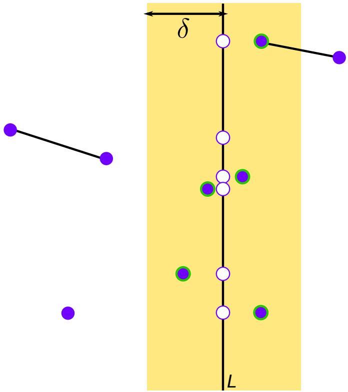


Closest Pair: Packing Argument

- 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.

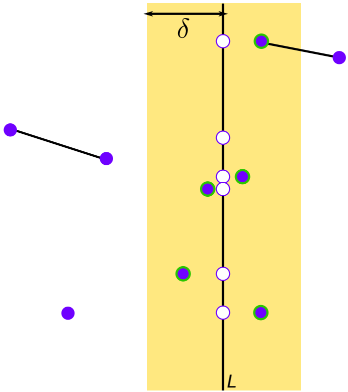
Closest Pair: Packing Argument

- 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 $s \in S$.



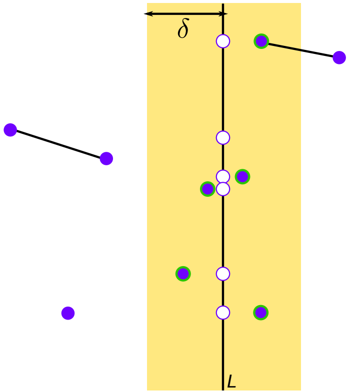
Closest Pair: Packing Argument

- 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.



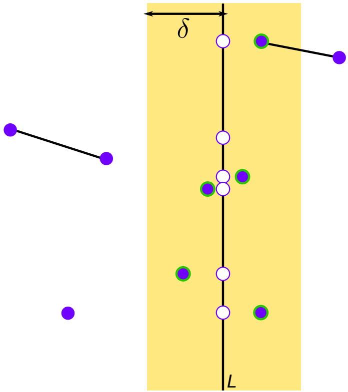
Closest Pair: Packing Argument

- 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 $s'_y - s_y \geq \delta$.



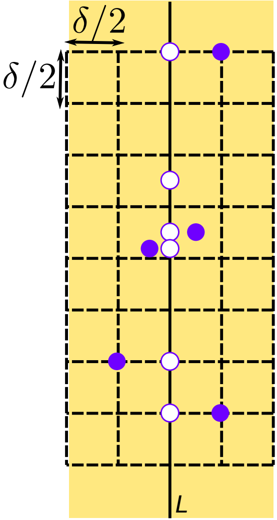
Closest Pair: Packing Argument

- 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.



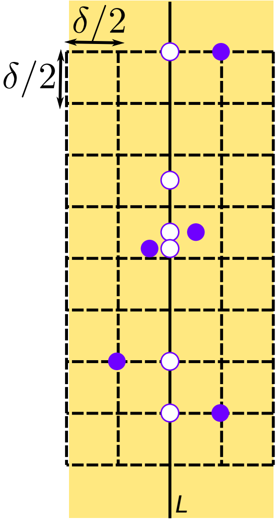
Closest Pair: Proof of Packing Argument

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



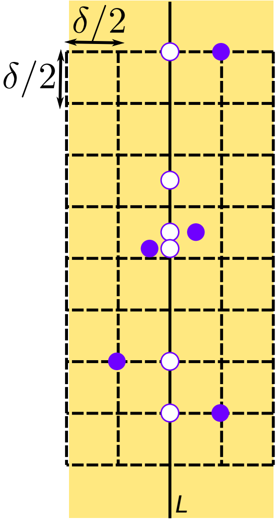
Closest Pair: Proof of Packing Argument

- 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 $\delta/2$.



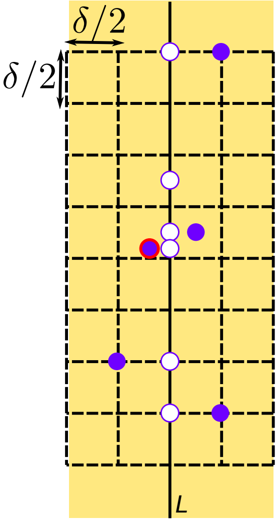
Closest Pair: Proof of Packing Argument

- 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.



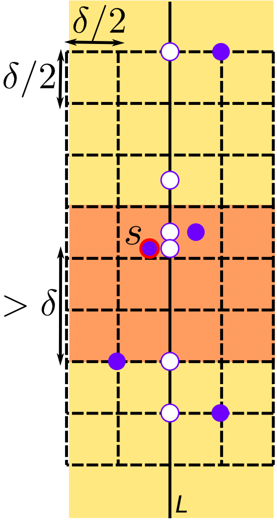
Closest Pair: Proof of Packing Argument

- 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.



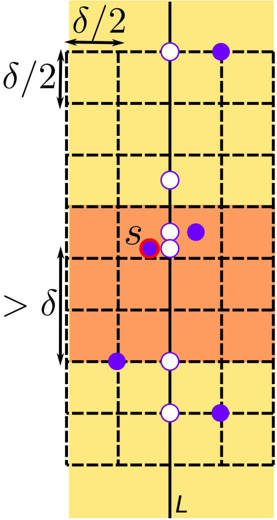
Closest Pair: Proof of Packing Argument

- 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 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 $\delta/2$.
- 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 $|P| \leq 3$ 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^*) = Closest-Pair-Rec(Q_x, Q_y)
  (r_0^*, r_1^*) = Closest-Pair-Rec(R_x, R_y)

  $\delta = \min(d(q_0^*, q_1^*), d(r_0^*, r_1^*))$
  x^* = maximum x-coordinate of a point 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 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(q0*,q1*) < d(r0*,r1*) then
   Return (q0*,q1*)
Else
   Return (r0*,r1*)
Endif
```