Design and Analysis of Algorithms, MTech-I (1^{st} semester)

Chapter 2: Divide and Conquer Design Approach

September 5, 2022

Devesh C Jinwala,

Professor in CSE, SVNIT, Surat and Adjunct Professor, IITJammu & Dean (R&C), SVNIT

Department of Computer Science and Engineering, SVNIT, Surat

The Basic D&C Paradigm

- Divide-and-conquer basic paradigm

The Basic D&C Paradigm

- Divide-and-conquer basic paradigm
  - Break up problem into several parts.

The Basic D&C Paradigm

- Divide-and-conquer basic paradigm
  - Break up problem into several parts.
  - Solve each part recursively.

The Basic D&C Paradigm

- Divide-and-conquer basic paradigm
  - Break up problem into several parts.
  - Solve each part recursively.
  - Combine solutions to sub-problems into overall solution.

The Basic D&C Paradigm

- Divide-and-conquer basic paradigm
  - Break up problem into several parts.
  - Solve each part recursively.
  - Combine solutions to sub-problems into overall solution.
- Most common usage.

The Basic D&C Paradigm

- Divide-and-conquer basic paradigm
  - Break up problem into several parts.
  - Solve each part recursively.
  - Combine solutions to sub-problems into overall solution.
- Most common usage.
  - Break up problem of size n into two equal parts of size $\frac{1}{2}n$.

The Basic D&C Paradigm

- Divide-and-conquer basic paradigm
  - Break up problem into several parts.
  - Solve each part recursively.
  - Combine solutions to sub-problems into overall solution.
- Most common usage.
  - Break up problem of size n into two equal parts of size $\frac{1}{2}n$.
  - Solve two parts recursively.

The Basic D&C Paradigm

- Divide-and-conquer basic paradigm
  - Break up problem into several parts.
  - Solve each part recursively.
  - Combine solutions to sub-problems into overall solution.
- Most common usage.
  - Break up problem of size n into two equal parts of size $\frac{1}{2}n$.
  - Solve two parts recursively.
  - Combine two solutions into overall solution in linear time.

The Basic D&C Paradigm

- Divide-and-conquer basic paradigm
  - Break up problem into several parts.
  - Solve each part recursively.
  - Combine solutions to sub-problems into overall solution.
- Most common usage.
  - Break up problem of size n into two equal parts of size $\frac{1}{2}$n.
  - Solve two parts recursively.
  - Combine two solutions into overall solution in linear time.
- Consequence.

The Basic D&C Paradigm

- Divide-and-conquer basic paradigm
  - Break up problem into several parts.
  - Solve each part recursively.
  - Combine solutions to sub-problems into overall solution.
- Most common usage.
  - Break up problem of size n into two equal parts of size $\frac{1}{2}$n.
  - Solve two parts recursively.
  - Combine two solutions into overall solution in linear time.
- Consequence.
  - Brute force: n^2.

The Basic D&C Paradigm

- Divide-and-conquer basic paradigm
  - Break up problem into several parts.
  - Solve each part recursively.
  - Combine solutions to sub-problems into overall solution.
- Most common usage.
  - Break up problem of size n into two equal parts of size $\frac{1}{2}$n.
  - Solve two parts recursively.
  - Combine two solutions into overall solution in linear time.
- Consequence.
  - Brute force: n^2.
  - Divide-and-conquer: $n\ logn$.

Motivation

DC algorithms often

- have lesser running times
- their running time can be determined by the standard tech to solve recurrences.

The Sorting Problem

Obvious sorting applications

- List files in a directory.
- Organize an MP3/MPEG library.
- List names in a phone book.
- Display Google PageRank results.

Not so obvious sorting applications

- Data compression.
- Computer graphics.
- Interval scheduling.
- Computational biology.
- Minimum spanning tree.
- Supply chain management.
- Simulate a system of particles.
- Book recommendations on Amazon.
- Load balancing on a parallel computer

The Sorting Problem...

The problem becomes easier if sorting applied

- Find the median.

The Sorting Problem...

The problem becomes easier if sorting applied

- Find the median.
- Find the closest pair.

The Sorting Problem...

The problem becomes easier if sorting applied

- Find the median.
- Find the closest pair.
- Binary search in a database.

The Sorting Problem...

The problem becomes easier if sorting applied

- Find the median.
- Find the closest pair.
- Binary search in a database.
- Identify statistical outliers.

The Sorting Problem...

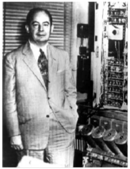
The problem becomes easier if sorting applied

- Find the median.
- Find the closest pair.
- Binary search in a database.
- Identify statistical outliers.
- Find duplicates in a mailing list.

Merge Sort: The Basic Logic

Overall Logic

- Divide array into two halves.



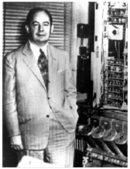
Jon von Neumann (1945)

A L G O R I T H M S			
A L G O R	I T H M S	divide	$O(1)$
A G L O R	H I M S T	sort	$2T(n/2)$
A G H I L M O R S T		merge	$O(n)$

Merge Sort: The Basic Logic

Overall Logic

- Divide array into two halves.
- Recursively sort each half.



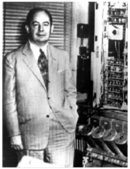
Jon von Neumann (1945)

A	L	G	O	R	I	T	H	M	S			
A	L	G	O	R	I	T	H	M	S	divide	$O(1)$	
A	G	L	O	R	H	I	M	S	T	sort	$2T(n/2)$	
A	G	H	I	L	M	O	R	S	T	merge	$O(n)$	

Merge Sort: The Basic Logic

Overall Logic

- Divide array into two halves.
- Recursively sort each half.
- Merge two halves to make sorted whole.



Jon von Neumann (1945)

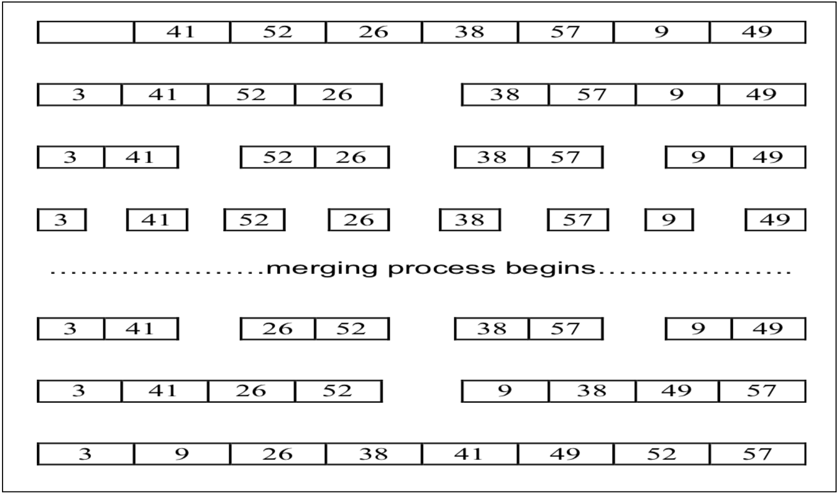
A L G O R I T H M S			
A L G O R	I T H M S	divide	$O(1)$
A G L O R	H I M S T	sort	$2T(n/2)$
A G H I L M O R S T		merge	$O(n)$

The Conquer Logic: Merging

Merging. Combine two pre-sorted lists into a sorted whole.

- How to merge efficiently?
- Linear number of comparisons.
- Use temporary array.
- The Challenge: In-place merge. [Kronrud, 1969]
- The Demo of Merge

Viewing Merge-sort in execution



The Merge-sort pseudocode

```
The Algorithm Merge-sort(A,p, r)
if (p < r)
        then q ← ⌊ (p + r)/2 ⌋
                MERGE-SORT(A, p, q)
                MERGE-SORT(A, q+1, r)
                MERGE(A, p, q, r)
```

The Merge procedure

```
Algorithm MERGE (A, p, q, r)
        let i = p and j = q+1 and k = 1
        while (i <= q) and (j <= r)
                do      if A[i] <= A[j]
                        then B[k] = A[i]
                                i = i + 1, k = k + 1
                        else B[k] = A[j]
                                j = j + 1, k = k + 1
// here one f the subarrays is in B
        if i > q then
        for index = j to r
                do B[k] = A[index]
                        k = k + 1
                else for index = i to q
                do B[k] = A[index]
                        k = k + 1
        for index = p to r
                do A[index] = B[index]
        return
```

The Merge Demo

Run the Demo MergeDemoFromPPT.mp4

The Merge-sort Analysis

- MERGE does $3n - 1 + 1$ comparisons i.e. time taken is $\theta(n)$ time . . . . a constant

The Merge-sort Analysis

- MERGE does $3n - 1 + 1$ comparisons i.e. time taken is $\theta(n)$ time .... a constant
- MERGE-SORT is a recursive procedure - i.e. a recurrence relation is to be framed.

The Merge-sort Analysis

- MERGE does $3n - 1 + 1$ comparisons i.e. time taken is $\theta(n)$ time . . . . a constant
- MERGE-SORT is a recursive procedure - i.e. a recurrence relation is to be framed.
    - an equation which defines a function over the natural numbers say $T(n)$, in terms of its own values at one or more integers smaller than n.

The Merge-sort Analysis

- MERGE does $3n - 1 + 1$ comparisons i.e. time taken is $\theta(n)$ time .... a constant
- MERGE-SORT is a recursive procedure - i.e. a recurrence relation is to be framed.
  - an equation which defines a function over the natural numbers say $T(n)$, in terms of its own values at one or more integers smaller than n.
  - expresses the resources used by the recursive procedures

Designing a recurrence relation

- IF $T(n)$ is the running time of the problem of size n, then

Designing a recurrence relation

- IF $T(n)$ is the running time of the problem of size n, then

$D(n)$ - time to divide into sub-problems

Designing a recurrence relation

- IF $T(n)$ is the running time of the problem of size n, then

$D(n)$ - time to divide into sub-problems

- divide the given problem into a subproblems each of size $1/b$ times the original problem.

Designing a recurrence relation

- IF $T(n)$ is the running time of the problem of size n, then

$D(n)$ - time to divide into sub-problems

- divide the given problem into a subproblems each of size $1/b$ times the original problem.
- Then, what is then the time to solve a sub-problem b?

Designing a recurrence relation

- IF $T(n)$ is the running time of the problem of size n, then

$D(n)$ - time to divide into sub-problems

- divide the given problem into a subproblems each of size $1/b$ times the original problem.
- Then, what is then the time to solve a sub-problem b?
- Then, what is then the time to solve a numbers of sub-problems of size b?

Designing a recurrence relation

- IF $T(n)$ is the running time of the problem of size n, then

$D(n)$ - time to divide into sub-problems

- divide the given problem into a subproblems each of size $1/b$ times the original problem.
- Then, what is then the time to solve a sub-problem b?
- Then, what is then the time to solve a numbers of sub-problems of size b?

C(n) - time to combine into solutions

$\theta(1)$ - time to solve the atomic subproblems - for small inputs such that $n \leq c$ for some c.

Designing a recurrence relation

- Then, the recurrence relation can be expressed as follows:

Designing a recurrence relation

- Then, the recurrence relation can be expressed as follows:

$T(n)$

$= \theta(1)$ for $n \leq= c$ for some c.
$= aT(n/b) + D(n) + C(n)$........ otherwise. recurrence relation using boundary conditions and without it

The Merge-sort recurrence relation

- T(n) = number of comparisons to mergesort an input of size n.

$$T(n) \leq \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{T(\lceil n/2 \rceil)}_{\text{solve left half}} + \underbrace{T(\lfloor n/2 \rfloor)}_{\text{solve right half}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

Figure: Mergesort Recurrence Relation

- Guess solution T(n) = O(n lg n)
- We now show number of ways to prove this result

Solving the recurrence : The Recursion Tree method

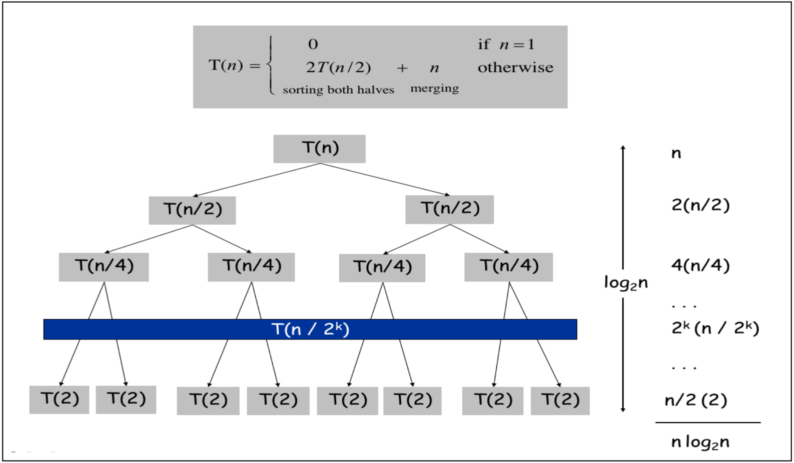


Figure: The Recursion Tree method

Solving the recurrence: By iterative substitution

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

Claim: If T(n) satisfies this recurrence, then T(n) = n log_2 n - assuming n is a power of 2

Proof : ... ...

Solving the recurrence : Proof by telescoping

$$T(n) = \begin{cases} 0 & \text{if } n=1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

Claim: If T(n) satisfies this recurrence, then T(n) $=$ n log_2 n - assuming n is a power of 2

Proof : ... ...

Solving the recurrence : Proof by Mathematical Induction

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

Claim: If T(n) satisfies this recurrence, then $T(n) = n\ log_2\ n$ - assuming n is a power of 2

- Base case: $n = 1$.
- Inductive hypothesis: $T(n) = n \lg n$.
- Goal: show that $T(2n) = 2n \lg (2n)$.

Proof : ... ...

... ...

Analysis of the Merge-sort recurrence

- What if n is not assumed to be a power of 2 ?

$$T(n) \le \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{T(\lceil n/2 \rceil)}_{\text{solve left half}} + \underbrace{T(\lfloor n/2 \rfloor)}_{\text{solve right half}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

Proof by induction on n

- Base case: n = 1.
- Define $n_1 = \lfloor n/2 \rfloor$, $n_2 = \lceil n/2 \rceil$
- Induction step: assume true for 1, 2, ... , n–1.
- Proof: ... ...

Analysis of the Merge-sort recurrence

- What if n is not assumed to be a power of 2 ?
- Then, we have to solve the following recurrence ...

$$T(n) \leq \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{T(\lceil n/2 \rceil)}_{\text{solve left half}} + \underbrace{T(\lfloor n/2 \rfloor)}_{\text{solve right half}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

Proof by induction on n

- Base case: n = 1.
- Define $n_1 = \lfloor n/2 \rfloor$, $n_2 = \lceil n/2 \rceil$
- Induction step: assume true for 1, 2, ... , n–1.
- Proof: ... ...

Solving Recurrences: Tutorial Problems

Solving the recurrences: By iterative substitution

Solve the following recurrences

1. Claim: If T(n) satisfies the recurrence $T(n) = 2T(n/2) + n$, then T(n) = n log_2 n - assuming n is a power of 2
2. x

Solving the recurrences: By Master's Theorem

Solve the following recurrences using Master's theorem

1. $T(n) = 2T(n/2) + n$
2. $T(n) = 3T(n/2) + n^2$
3. $T(n) = T(n/2) + 2^n$
4. $T(n) = 2^n T(n/2) + n^n$
5. $T(n) = 16T(n/4) + n$
6. $T(n) = 2T(n/2) + n \; log \; n$
7. $T(n) = 3T(n/4) + n^{0.51}$
8. $T(n) = 16T(n/4) + n!$

Solving the recurrences: By Master's Theorem

Solve the following recurrences using Master's theorem

1. $T(n) = \sqrt{2}T(n/2) + log\ n$
2. $T(n) = 3T(n/2) + n$
3. $T(n) = 3T(n/3) + \sqrt{n}$
4. $T(n) = 4T(n/2) + cn$
5. $T(n) = 3T(n/4) + n\ log\ n$
6. $T(n) = 3T(n/3) + n/2$
7. $T(n) = 7T(n/3) + n^2$
8. $T(n) = 4T(n/2) + log\ n$

MergeSort Applications: 1

Counting Inversion: Motivation

Collaborative Flitering

- A number of websites use collaborative filtering to "identify" people with similar taste and then push the related contents to the entire collection.
- The meta-search engines execute the same query on many different search engines and then try to synthesize the results by looking for similarities.

There are many such other applications where *finding similarity* is required. What could be the metric ?

Measuring Similarity

- How to measure how similar are two people's rankings?

Measuring Similarity

- How to measure how similar are two people's rankings?
- Find the items *out of order*?

Measuring Similarity

- How to measure how similar are two people's rankings?
- Find the items *out of order*?
  - my rank: 1, 2, . . . , n.

Measuring Similarity

- How to measure how similar are two people's rankings?
- Find the items *out of order*?
  - my rank: 1, 2, . . . , n.
  - Your rank: a_1, a_2,........, a_n.

Measuring Similarity

- How to measure how similar are two people's rankings?
- Find the items *out of order*?
  - my rank: 1, 2, . . . , n.
  - Your rank: a_1, a_2,........, a_n.
- How to find the items *out of order* between two lists

Measuring Similarity

- How to measure how similar are two people's rankings?
- Find the items *out of order*?
  - my rank: 1, 2, . . . , n.
  - Your rank: a_1, a_2,........, a_n.
- How to find the items *out of order* between two lists
- Songs i and j inverted if $i < j$, but $a_i > a_j$.

Measuring Similarity

- How to measure how similar are two people's rankings?
- Find the items *out of order*?
    - my rank: 1, 2, . . . , n.
    - Your rank: a_1, a_2,........, a_n.
- How to find the items *out of order* between two lists
- Songs i and j inverted if $i < j$, but $a_i > a_j$.
- Suppose your ranking of five songs is 5,4,3,2,1 and mine is in ascending order. How many are out of order ?

Counting the items out of order

Songs

	A	B	C	D	E
Me	1	2	3	4	5
You	1	3	4	2	5

Inversions

3-2, 4-2

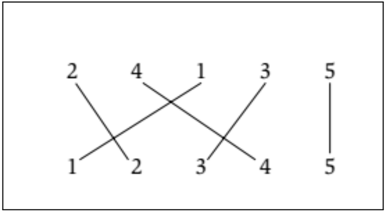


Figure: Similarity metric:Inversion:number of inversions between two rankings.

Issues in counting inversion

- How many inversions can be, let us say, given two arrrays to compare ?

Issues in counting inversion

- How many inversions can be, let us say, given two arrrays to compare ?
  - there can be a quadratic number of inversions.

Issues in counting inversion

- How many inversions can be, let us say, given two arrrays to compare ?
  - there can be a quadratic number of inversions.
- Hence, we need faster algorithms.....

Issues in counting inversion

- How many inversions can be, let us say, given two arrrays to compare ?
  - there can be a quadratic number of inversions.
- Hence, we need faster algorithms.....
- Asymptotically faster algorithm must compute total number without even looking at each inversion individually.

Counting Inversion: Applications

- Voting theory.

Counting Inversion: Applications

- Voting theory.
- Collaborative filtering - books, movies, songs, restaurants

Counting Inversion: Applications

- Voting theory.
- Collaborative filtering - books, movies, songs, restaurants
- Measuring the "sortedness" of an array.

Counting Inversion: Applications

- Voting theory.
- Collaborative filtering - books, movies, songs, restaurants
- Measuring the "sortedness" of an array.
- Sensitivity analysis of Google's ranking function.

Counting Inversion: Applications

- Voting theory.
- Collaborative filtering - books, movies, songs, restaurants
- Measuring the "sortedness" of an array.
- Sensitivity analysis of Google's ranking function.
- Rank aggregation for meta-searching on the Web.

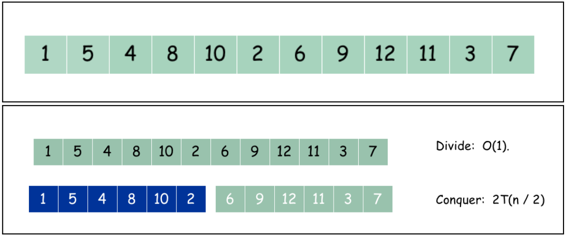
Counting Inversion: Applications

- Voting theory.
- Collaborative filtering - books, movies, songs, restaurants
- Measuring the "sortedness" of an array.
- Sensitivity analysis of Google's ranking function.
- Rank aggregation for meta-searching on the Web.
- Nonparametric statistics (e.g., Kendall's Tau distance).

Counting Inversions : Logic

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

Counting Inversions : Logic



Counting Inversions : Logic

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

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

Divide: $O(1)$.

1	5	4	8	10	2

6	9	12	11	3	7

Conquer: 2T(n / 2)

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

Divide: $O(1)$.

1	5	4	8	10	2

6	9	12	11	3	7

Conquer: 2T(n / 2)

5 blue-blue inversions

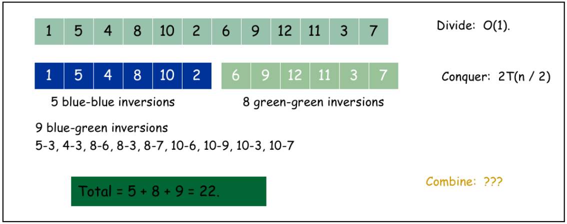
5-4, 5-2, 4-2, 8-2, 10-2

8 green-green inversions

6-3, 9-3, 9-7, 12-3, 12-7, 12-11, 11-3, 11-7

Counting Inversions: Divide and Conquer

- Divide: separate list into two pieces.
- Conquer: recursively count inversions in each half.
- Combine: count inversions where a_i and a_j are in different halves, and return sum of three quantities. Complexity ??

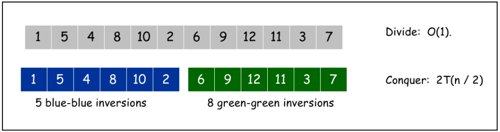


Dry running Merge-and-count(A,B)

Given the two sorted subhalves A and B

- Combine: count blue-green inversions
- Assume each half is sorted.
- Count inversions where ai and aj are in different halves.
- Merge two sorted halves into sorted whole.

Apply the logic to the given two subhalves as below

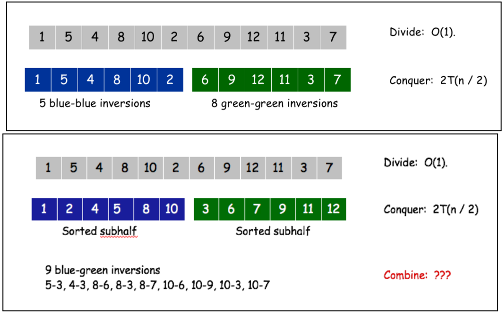


Dry running Merge-and-count(A,B)

Given the two sorted subhalves A and B

- Combine: count blue-green inversions
- Assume each half is sorted.
- Count inversions where ai and aj are in different halves.
- Merge two sorted halves into sorted whole.

Apply the logic to the given two subhalves as below



The Merge-Invert Demo

Run the Demo MergeInvertFromPPT

Counting Inversions pseudocode: MergeandCount(A,B)

```
Algorithm Merge-and-count(A, B)
Maintain a Current pointer into each list,
            initialized to point to the front elements
Maintain a variable Count for the number of inversions,
            initialized to 0
While both lists are nonempty {
      Let a_i and b_j be the elements pointed to
            by the Current pointer
      Append the smaller of these two to the output list
      If b_j < a_i then
            increment Count by the number of
                  elements remaining in A
            Advance the Current pointer in the list from
                  which the smaller element
                  was selected.
}
```

Complexity

3	7	10	14	18	19

2	11	16	17	23	25
6	3	2	2	0	0

13 blue-green inversions: 6 + 3 + 2 + 2 + 0 + 0

Count: O(n)

2	3	7	10	11	14	16	17	18	19	23	25

Merge: O(n)

$$T(n) \leq T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n) \quad \Rightarrow \mathrm{T}(n) = O(n \log n)$$

Counting Inversion: Implementation

```
Algorithm Sort-and-Count(L):
\\Pre-condition: [Merge-and-Count]  A and B are sorted.
\\ Post-condition. [Sort-and-Count]  L is sorted.

1.      if list L has one element
2.              return 0 and the list L
3.      Divide the list into two halves A and B
4.      (r_A, A) ← Sort-and-Count(A)
5.      (r_B, B) ← Sort-and-Count(B)
6.      (r, L) ← Merge-and-Count(A, B)
7.      return r = r_A + r_B + r and the sorted list L
```

MergeSort Applications: 2

Mergesort Applications: Closest Pairs of Points

Closest pair

Given n points in the plane, find a pair with smallest Euclidean distance between them.
What is the Euclidean distance between points $p_1(x_1, y_1)$ and $p_2(x_2, y_2)$?

Closest Pairs of Points: Applications

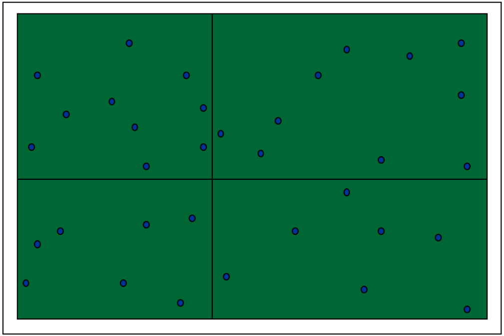
- Fundamental geometric primitive in
  - graphics, pattern recognition
  - computer vision, image processing, VLSI design
  - geographic information systems,
  - molecular modeling,
  - air traffic control.
  - Special case of nearest neighbor, Euclidean MST, Voronoi.

Finding Closest Pairs..: Approaches

- Brute force: What will be the time for a brute force approach ?
- How would it work ?
- 1-D version....What is a 1-D version ?
- Approach. . . ..
  - Divide into two parts and compute within each. . . .
  - O(n log n) easy if points are on a line.
  - Assumption: No two points have different x coordinate.
- What do learn from this exercise ?

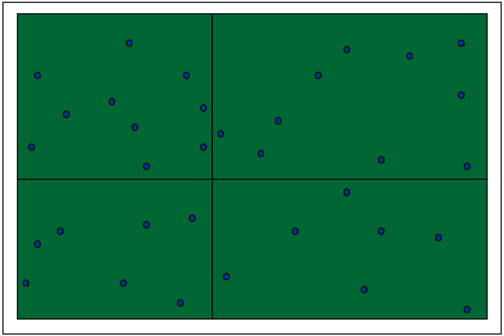
Closest Pairs...First attempt

- Divide. Sub-divide region into 4 quadrants.



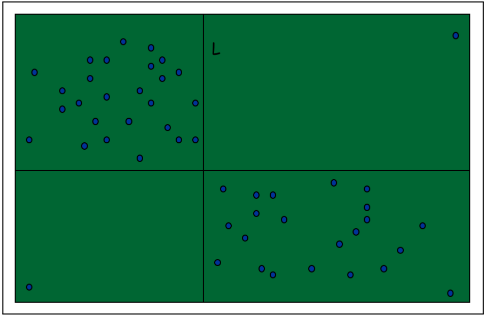
Closest Pairs...First attempt

- Divide. Sub-divide region into 4 quadrants.
- What is the obstacle in this case ?



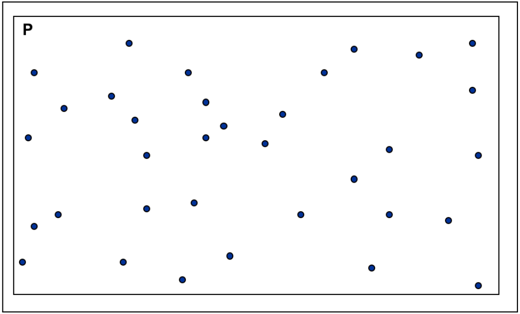
Closest Pairs...First attempt...

- The obstacle



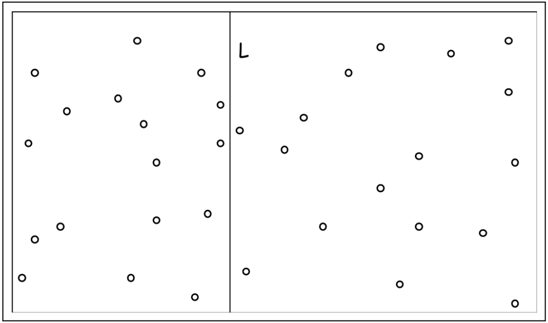
Closest Pairs: D & C Approach

- Suppose we are given a sample point set P as shown below:



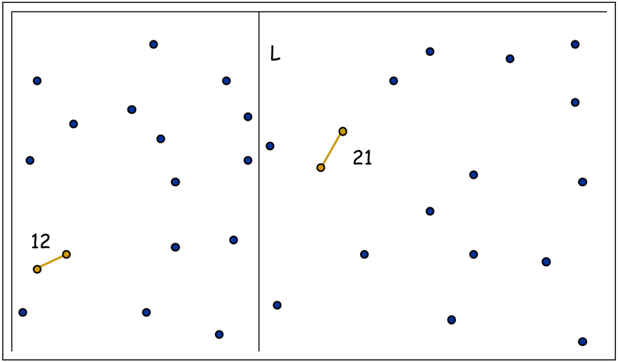
Closest Pairs: D & C Approach...

- Suppose we are given a sample point set P as shown below:
- Divide: Draw a vertical line L so that roughly $(1/2)n$ points are there on each side.



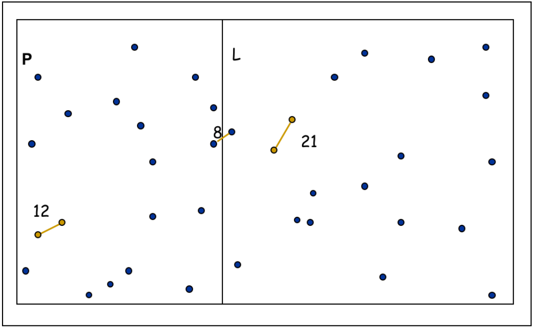
Closest Pairs: D & C Approach...

- Divide: Draw a vertical line L so that roughly $(1/2)n$ points are there on each side.
- Conquer: Find the closest pair in each side recursively.
- What is the partial recurrence relation ?



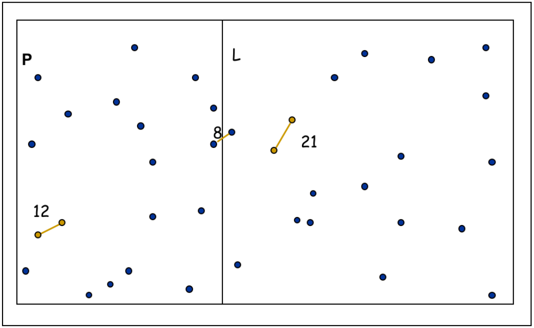
Closest Pairs: D & C Approach...

- Divide: Draw a vertical line L so that roughly $(1/2)n$ points are there on each side.
- Conquer: Find the closest pair in each side recursively.
- Combine: Find closest pair with one point in each side.
- Return the best of 3 solutions.



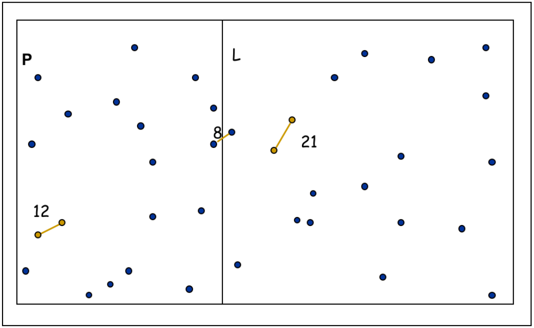
Closest Pairs: D & C Approach...

- How does the recurrence relation now take shape ?



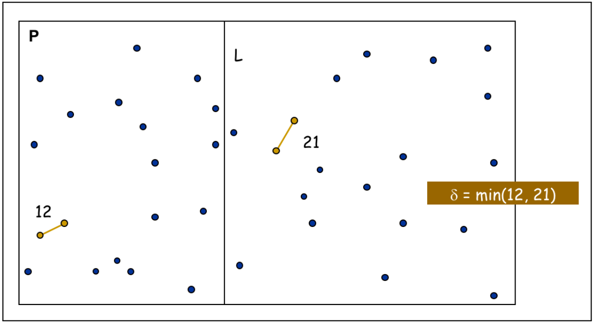
Closest Pairs: D & C Approach...

- What is the time required for combining ?



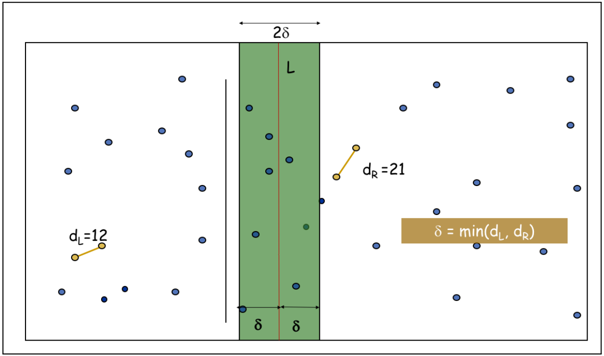
Closest Pairs: Second D & C Approach

- Find closest pair with one point in each side, assuming that distance $< \delta$ i.e. $\delta = \min(d_L, d_R)$



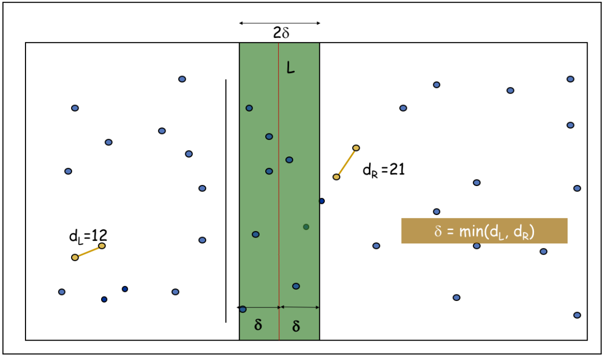
Closest Pairs: Second D & C Approach

- What is the primary observation and inference especially having computed dL and dR and δ=min(dL, dR) ?



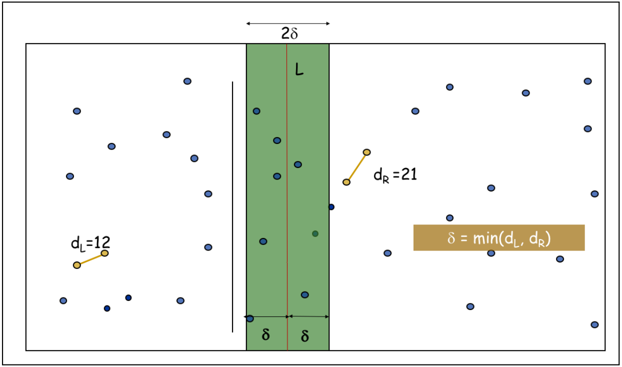
Closest Pairs: Second D & C Approach

- What is the primary observation and inference especially having computed dL and dR and δ=min(dL, dR) ?
- we need to compute dC i.e. the distance between two points splitting across the left and the right subhalves ONLY if the dC improves upon the dL and dR.



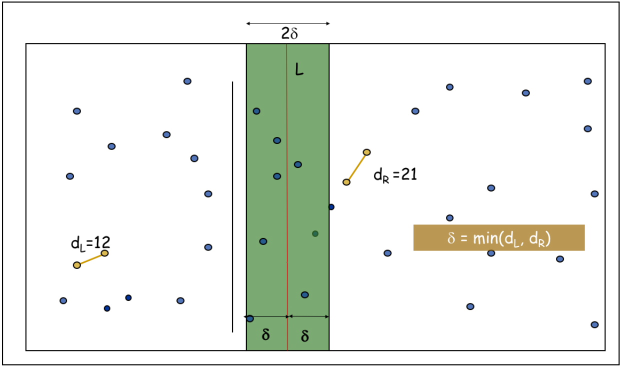
Closest Pairs: Second D & C Approach...

- How many points could be there in this strip ?



Closest Pairs: Second D & C Approach...

- Then, how to compute the distance $\min(\delta(dL, dR)), dC)$?



Closest Pairs: Second D & C Approach...

- Then, how to compute the distance $\min(\delta(d_L, d_R)), d_C)$?

```
Algorithm ComputeMin()
1. for i = 1 to Num_points_in_the_strip
2       for j = (i+1) to Num_points_in_the_strip
3.            if (dist(P_i, P_j) < δ)
4.                  δ = dist(P_i, P_j)
```

Not efficient

- With all the points located in the strip, the complexity is $\theta(n^2)$
- Thus, there is a need to improve upon this approach. . . . . .

Closest Pairs: Second D & C Approach...

Two approaches

- We assume that with n points in the entire plane, there are only $O(n^{0.5})$ points in the strip on an average. . . .
- We can do a brute force on the points lying in the strip.
- What will be the time taken by this approach for brute force then ?
- What will be the total run time ?
- But, then is it possible ?? Can it be assumed in the worst case ?

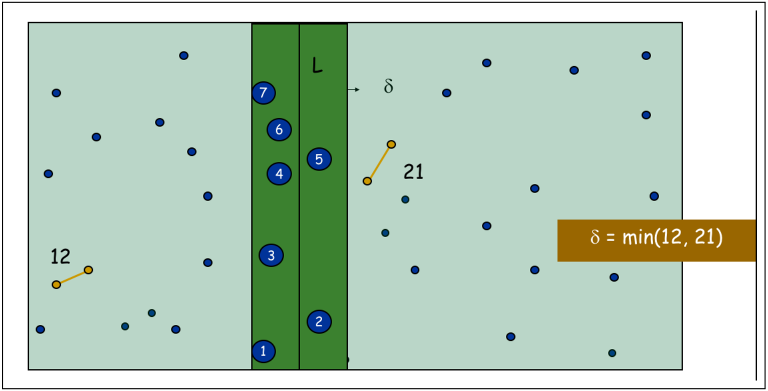
Closest Pairs: Second D & C Approach...

- Improved approach : Sort the points in the strip based on their y coordintaes....

```
Algorithm ComputeMinImproved()
1. for i = 1 to Num_points_in_the_strip
2.         for j = (i+1) to Num_points_in_the_strip
3.                 if (P_i and P_j 's y coordinates differ by
                            more than δ)
4.                          break; //Go to the next P
5.                 else
6.                 if (dist(P_i, P_j) < δ )
7.                          δ = dist(P_i, P_j)
```

Closest Pairs: Second D & C Approach...

- only need to consider points within δ of line L.
- Sort points in 2δ-strip by their y coordinate.



Closest Pairs: Second D & C Approach...

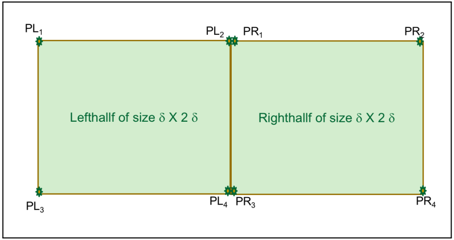
- only need to consider points within δ of line L.
- Sort points in 2δ-strip by their y coordinate.
- How many points to be considered in the worst case ?

Closest Pairs: Second D & C Approach...

- Let s_i be the point in the 2δ-strip, with the i_th smallest y-coordinate.
- Claim. If $|i - j| \geq 7$, then the distance between the distinct points s_i and s_j is at least δ
- Proof.

Proof

Proof to be worked out.



Closest Pairs: Second D & C Approach...

Time for computing d_C

Because only seven points are considered for each pi, the time for computing d_C that is better than δ is O(n).
Thus, we appear to have a O(n log n) solution to the closest pairs of points problem.

Closest Pairs algorithm

```
Algorithm ClosestPair(p1, ..., pn)
1.Compute separation line L such that half the points
                    are on one side and half on the other side
2.      δ1 = Closest−Pair(left half)
3.      δ2 = Closest−Pair(right half)
4.      δ = min(δ1, δ2)
5.      Delete all points further than δ from separation
              line L
6.      Sort remaining points by y−coordinate.
7. Scan points in y−order and compare distance between
              each point and next 11 neighbors.
              If any of these distances is less
              than δ, update δ.
8. return δ.
```

Algorithm Closest Pairs

- Running time
  - We have assumed that a list of points sorted by y coordinates is already available.

Algorithm Closest Pairs

- Running time
  - We have assumed that a list of points sorted by y coordinates is already available.
  - If we perform this sort in each recursive call, then we have $O(n\ log\ n)$ extra work

Algorithm Closest Pairs

- Running time
  - We have assumed that a list of points sorted by y coordinates is already available.
  - If we perform this sort in each recursive call, then we have $O(n\ log\ n)$ extra work
  - This gives an $O(n\ log^2\ n)$ algorithm.

Algorithm Closest Pairs

- Running time
    - We have assumed that a list of points sorted by y coordinates is already available.
    - If we perform this sort in each recursive call, then we have $O(n\ log\ n)$ extra work
    - This gives an $O(n\ log^2\ n)$ algorithm.
    - That is,

Algorithm Closest Pairs

- Running time
  - We have assumed that a list of points sorted by y coordinates is already available.
  - If we perform this sort in each recursive call, then we have $O(n\ log\ n)$ extra work
  - This gives an $O(n\ log^2\ n)$ algorithm.
  - That is,

Algorithm Closest Pairs

- Running time
  - We have assumed that a list of points sorted by y coordinates is already available.
  - If we perform this sort in each recursive call, then we have $O(n\ log\ n)$ extra work
  - This gives an $O(n\ log^2\ n)$ algorithm.
  - That is,

$$\boxed{T(n) \leq 2T(n/2) + O(n \log n) \Rightarrow T(n) = O(n \log^2 n)}$$

Algorithm Closest Pairs

- Running time
  - We have assumed that a list of points sorted by y coordinates is already available.
  - If we perform this sort in each recursive call, then we have $O(n\ log\ n)$ extra work
  - This gives an $O(n\ log^2\ n)$ algorithm.
  - That is,

$$\boxed{T(n) \leq 2T(n/2) + O(n \log n) \Rightarrow T(n) = O(n \log^2 n)}$$

This is certianly better than the brute force $O(n^2)$ work.

Algorithm Closest Pairs: Third D&C approach

- Can we achieve O(n log n)?

Algorithm Closest Pairs: Third D&C approach

- Can we achieve O(n log n)?
- We can, by not sorting points in strip from scratch each time - in each recursive call.

Algorithm Closest Pairs: Third D&C approach

- Can we achieve O(n log n)?
- We can, by not sorting points in strip from scratch each time - in each recursive call.
- instead,

Algorithm Closest Pairs: Third D&C approach

- Can we achieve O(n log n)?
- We can, by not sorting points in strip from scratch each time - in each recursive call.
- instead,
    - Let two lists be maintained - one sorted by the x coordinate and the other by the y coordinate.

Algorithm Closest Pairs: Third D&C approach

- Can we achieve O(n log n)?
- We can, by not sorting points in strip from scratch each time - in each recursive call.
- instead,
    - Let two lists be maintained - one sorted by the x coordinate and the other by the y coordinate.
    - This can be achieved by a pre-processing sorting step that costs $O(n\ log\ n)$

Algorithm Closest Pairs: Third D&C approach

- Can we achieve O(n log n)?
- We can, by not sorting points in strip from scratch each time - in each recursive call.
- instead,
  - Let two lists be maintained - one sorted by the x coordinate and the other by the y coordinate.
  - This can be achieved by a pre-processing sorting step that costs $O(n\ log\ n)$
- Each recursive returns two lists: all points sorted by y coordinate, and all points sorted by x coordinate.

Algorithm Closest Pairs: Third D&C approach

- Can we achieve O(n log n)?
- We can, by not sorting points in strip from scratch each time - in each recursive call.
- instead,
  - Let two lists be maintained - one sorted by the x coordinate and the other by the y coordinate.
  - This can be achieved by a pre-processing sorting step that costs $O(n\ log\ n)$
- Each recursive returns two lists: all points sorted by y coordinate, and all points sorted by x coordinate.
- Sort by merging two pre-sorted lists

Algorithm Closest Pairs: Third D&C approach

- Can we achieve O(n log n)?
- We can, by not sorting points in strip from scratch each time - in each recursive call.
- instead,
  - Let two lists be maintained - one sorted by the x coordinate and the other by the y coordinate.
  - This can be achieved by a pre-processing sorting step that costs $O(n\ log\ n)$
- Each recursive returns two lists: all points sorted by y coordinate, and all points sorted by x coordinate.
- Sort by merging two pre-sorted lists

Algorithm Closest Pairs: Third D&C approach

- Can we achieve O(n log n)?
- We can, by not sorting points in strip from scratch each time - in each recursive call.
- instead,
    - Let two lists be maintained - one sorted by the x coordinate and the other by the y coordinate.
    - This can be achieved by a pre-processing sorting step that costs $O(n\ log\ n)$
- Each recursive returns two lists: all points sorted by y coordinate, and all points sorted by x coordinate.
- Sort by merging two pre-sorted lists

$$\boxed{T(n) \leq 2T\left(n/2\right) + O(n) \quad \Rightarrow \quad \mathrm{T}(n) = O(n \log n)}$$

B L A N K   P A G E

B L A N K   P A G E