5 Divide and Conquer Algorithms

5.1 Translating algorithms into recurrence relations and deriving upper bounds

Key Feature Divide the input into several parts (usually of equal size), solve the problem for each part in a recursive fashion and, finally, combine the solutions for the parts into the overall solution.

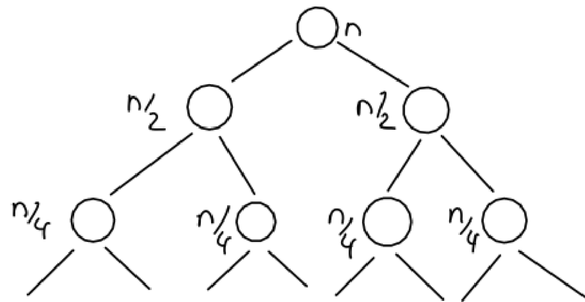
Key strategy for a run-time analysis:

Use a recurrence relation that bounds the run time recursively in terms of run times for smaller instances.

Examples of recurrence relations:

Example 1: Generic Algorithm Nr. 1

- divide the input of size n into two pieces of size $\frac{n}{2}$
  - solve the problem for each piece separately in a recursive fashion
  - combine the two results into an overall solution
  - need linear time to divide sets and combine solutions

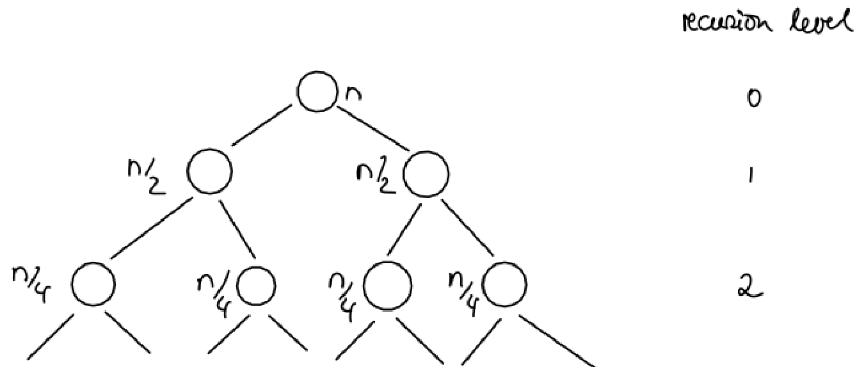


- define $T(n)$:= the worst-case time of the algorithm for an input size n (in the following, assume $n = 2^k$, $k \in \mathbb{N}$)
- then

$$T(n) \leq 2T(\frac{n}{2}) + cn \quad \text{(recurrence relation)(*)}$$

for some constant $c \in \mathbb{R}+$, if $n > 2$. And $T(2) \leq c$ for $n = 2$.

Example Mergesort algorithm



Group work:

- recurrence relation:

$$T(n) \le 2T(\frac{n}{2}) + cn$$

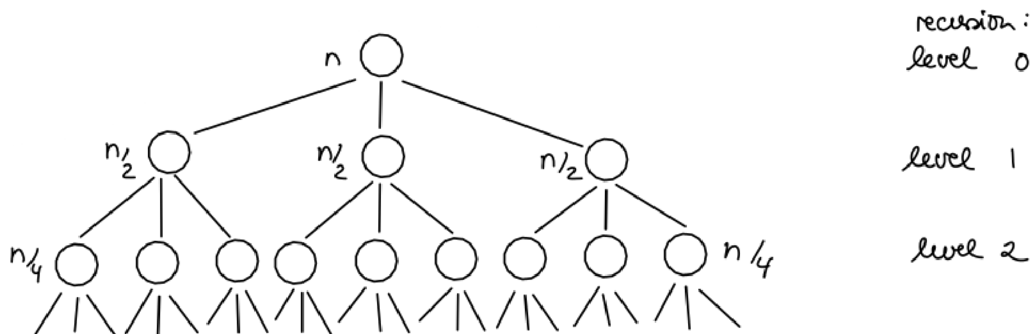
- how many levels k do we need to reduce the problem size from n to 1?
- What is the total size of the problem at each recursion level?

Answers: We require $k = log_2(n)$ levels and the problem size at every level is constant and equal to n.

Example 2: Generic Algorithm Nr. 2 (generalizes alg. Nr. 1)

- divide the input of size n into q pieces of size $\frac{n}{2}$ each, $q \in \mathbb{N}$
  - solve the problem for each piece separately in a recursive fashion
  - combine the q results into an overall solution
  - need linear time to divide sets and combine solutions

Example ($q = 3$):



Group Work: Write down the corresponding recurrence relation. The corresponding recurrence relation is:

$$T(n) \le qT(\frac{n}{2}) + cn \tag{1}$$

for $n > 2$ and some constant $c \in \mathbb{R}+$ for $n = 2$ have $T(2) \le c$.

Goal Solve recurrence relation (1) above. Solve cases $q = 1$ and $q = 2$ and $q > 2$ separately, where $q \in \mathbb{N}$.

Case 1: $q = 2$ (See earlier lecture)

- at each recursion level k the total amount of work to be done is bound by ($k = 0$ at start for input size n):

$$2^k \cdot c \cdot \frac{n}{2^k} = cn$$

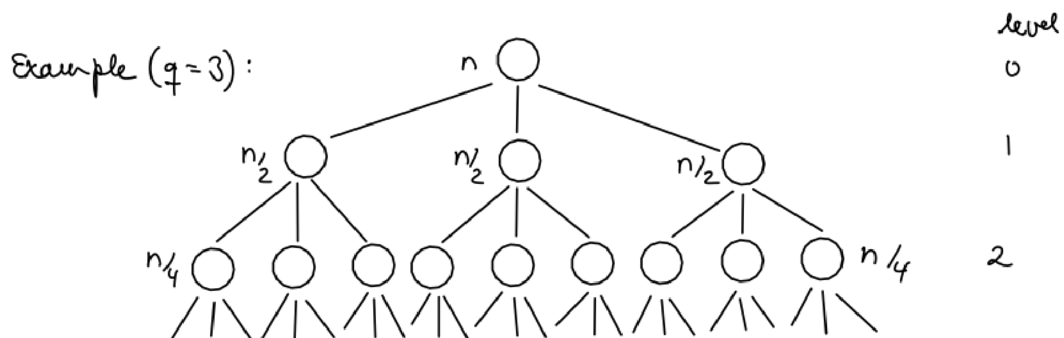
- we need k $log_2(n)$ levels or layers to reduce the input size n to 1

$\Rightarrow$ overall by summing over $log_2(n)$ levels that each require cn, we get a total run time of

$$O(nlog(n))$$

Case 2: $q > 2$, $q \in \mathbb{N}$

Group work What changes by going from $q = 2$ to $q > 2$?



- How many levels do we need?
- How much total work is done at level k?

Answers:

- at recursion k, we have q^k instances of size $\frac{n}{2^k}$ each
- the total work performed at level k is thus $c \cdot q^k \cdot \frac{n}{2^k} = c \cdot n \cdot (\frac{q}{2})^k$
- we need $log_2(n)$ levels of recursion until our problem size is reduced to 1

We solve the recurrence relation by summing over all recursion levels:

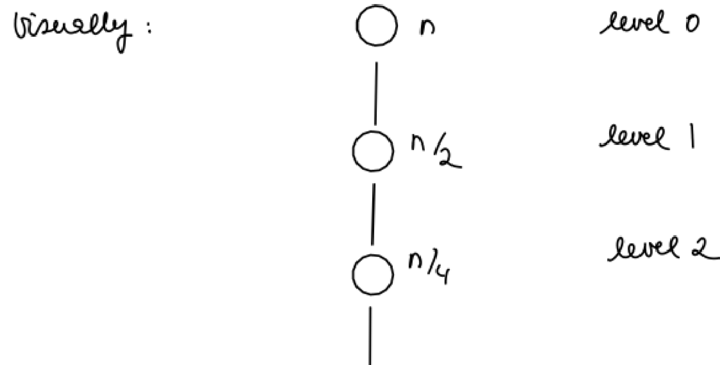
$$
\begin{aligned}
(1) \Leftrightarrow T(n) &\leq q \cdot T(\frac{n}{2}) + c \cdot n \qquad \text{(assume } n > 2\text{)} \\
&\leq \sum_{i=1}^{log_2(n)} (\frac{q}{2})^{i-1} n \cdot c \qquad \text{(as we are summing over all } log_2(n) \text{ recursion levels)} \\
&= c \cdot n \sum_{i=1}^{log_2(n)} (\frac{q}{2})^{i-1} \\
&= c \cdot n \sum_{i=1}^{log_2(n)-1} (\frac{q}{2})^{i} \\
&= c \cdot n (\frac{1 - (\frac{q}{2})^{log_2(n)}}{1 - (\frac{q}{2})}) \qquad \text{(use geometric sum: } \sum_{i=0}^{m} r^i = \begin{cases} \frac{1-r^{m+1}}{1-r} & r \neq 1 \\ m+1 & r = 1 \end{cases} \text{)} \\
&= c \cdot n (\frac{(\frac{q}{2})^{log_2(n)} - 1}{(\frac{q}{2}) - 1}) \\
&\leq c \cdot n (\frac{(\frac{q}{2})^{log_2(n)}}{(\frac{q}{2}) - 1}) \qquad \text{(use } a^{logb} = e^{logb \cdot loga} = b^{loga}\text{)} \\
&= cn (\frac{n^{log_2 * (\frac{q}{2})}}{(\frac{q}{2}) - 1}) \qquad \text{(use } log_2(\frac{q}{2}) = log_2(q) - log_2(2) \text{(where } \log_2(2) = 1\text{))} \\
&= \frac{c \cdot n \cdot n^{log_2(q)-1}}{\frac{q}{2} - 1} \qquad \text{(view } \frac{c}{\frac{q}{2} - 1} \text{ as constant)} \\
&= \frac{cn^{log_2}(q)}{\frac{q}{2} - 1} \qquad \text{(note : } \frac{q}{2} > 1 \text{ because } q > 2\text{)} \\
&= O(n^{log_2(q)})
\end{aligned}
$$

To summarize:

Any function $T(n)$ satisfying the recurrence relation (1) for $q > 2$ is bounded by $O(n^{log_2(q)})$.

Note:

As $q > 2$ and thus $log_2(q) > 1$, the running time is more than linear, but polynomial in n.



Case 3: $q = 1$

Group Work: What changes with respect to $q = 2$ and $q > 2$?

Answer:
- at recursion level k, we have one instance of size $\frac{n}{2^k}$ each
- the total work performed at level k is thus $c \cdot \frac{n}{2^k}$
- we need $log_2(n)$ levels of recursion until our problem size is reduced to 1

Group Work:

As for case B ($q > 2$), try to solve (1) by summing over all recursion levels.

$$\begin{aligned}
T(n) &\leq q \cdot T(\frac{n}{2}) + c \cdot n \\
&\leq \sum_{i=1}^{log_2(n)} \frac{c \cdot n}{2^{i-1}} && \text{(sum over all } log_2(n) \text{ recursion levels)} \\
&= c \cdot n \sum_{i=0}^{log_2(n)-1} (\frac{1}{2})^i && \text{(use geometric sum)} \\
&= c \cdot n (\frac{1 - (\frac{1}{2})^{log_2(n)}}{1 - \frac{1}{2}}) && \text{(use } a^{logb} = b^{loga}\text{)} \\
&= c \cdot n (\frac{1 - n^{log_2(\frac{1}{2})}}{1 - \frac{1}{2}}) \\
&= 2cn(1 - n^{-1}) && (log_2(1/2) = -1) \\
&= 2cn - 2c \\
&\leq 2 \cdot cn
\end{aligned}$$

To Summarize:

Any function $T(n)$ satisfying the recurrence relation (1) with $q = 1$ is bounded by $O(n)$.

Note:

Even though we need $log_2(n)$ layers, the overall run time is linear in n and half the work performed by the algorithm is done at the top level of the recursion.

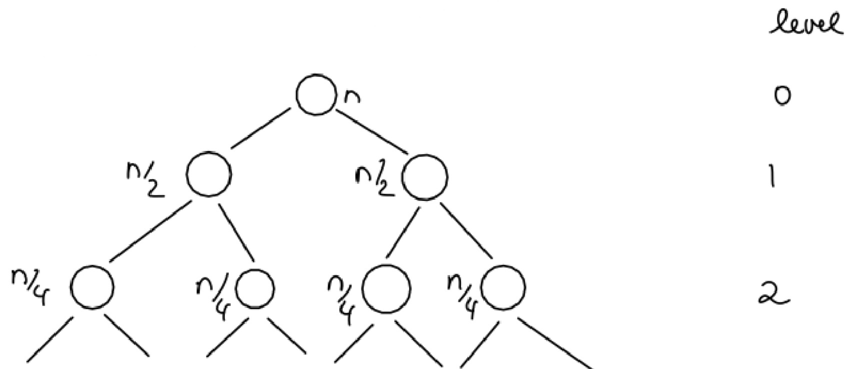
Example 3: Generic Algorithm Nr. 3 (Compare to Alg. Nr. 1)

- divide the input of size n into 2 pieces of size $\frac{n}{2}$ each
  - solve the problem for each piece separately in a recursive function
  - combine the *two* results into an overall solution
  - require *quadratic* time for dividing and recombining solutions

The corresponding recurrence relation is

$$T(n) \leq 2 \cdot T(\frac{n}{2}) + cn^2 \qquad \text{for } n > 2 \text{ and } T(2) \leq c. \tag{2}$$

Solving the recurrence relation:



Group Work: At any given level k, what is the total amount of work required?

Answer:
- require $log_2(n)$ levels of recursion until we have reduced the problem size to 1
- at a given level k, we are dealing with 2^k problems of size $\frac{n}{2^k}$ each which each require at most $c \cdot (\frac{n}{2^k})^2$ time, i.e. a total time of up to $2^k \cdot c \cdot (\frac{n}{2^k})^2 = c\frac{n^2}{2^k}$

Solving (2) by explicitly summing over all recursion levels we obtain:

$$
\begin{aligned}
T(n) &\leq 2 \cdot T(\frac{n}{2}) + cn^2 \\
&\leq \sum_{i=1}^{log_2(n)} (\frac{cn^2}{2^{i-1}}) \\
&= cn^2 \sum_{i=1}^{log_2(n)} (\frac{1}{2})^{i-1} \\
&= cn^2 \sum_{i=0}^{log_2(n)-1} (\frac{1}{2})^{i} && \text{(use geometric sum)} \\
&= cn^2 (\frac{1-(\frac{1}{2})^{log_2(n)}}{1-(\frac{1}{2})}) && \text{(use } a^{logb} = b^{loga} \text{ and } log_2(1/2) = -1\text{)} \\
&= cn^2 2(1-n^{-1}) \\
&= 2cn^2 - 2cn \leq 2cn^2 = O(n^2)
\end{aligned}
$$

To summarize:

Any function $T(n)$ satisfying the recurrence relation (2) is bounded by $O(n^2)$.

5.2 Translating algorithms into recurrence relations and deriving upper bounds

5.2.1 Example 1: Counting Inversions

Situation Given two rankings, $A = (a_1, a_2, \ldots, a_n)$ and $B = (b_1, b_2, \ldots, b_n)$ of numbers a_i and b_i, where

$$\{a_1, a_2, \ldots, a_n\} = \{b_1, b_2, \ldots, b_n\} = \mathbb{N}_n = \{1, 2, \ldots, n\}$$

Assumption In all of the following, assume that $B = (1, 2, 3, \ldots, n)$, i.e. that B denotes the reference ranking.

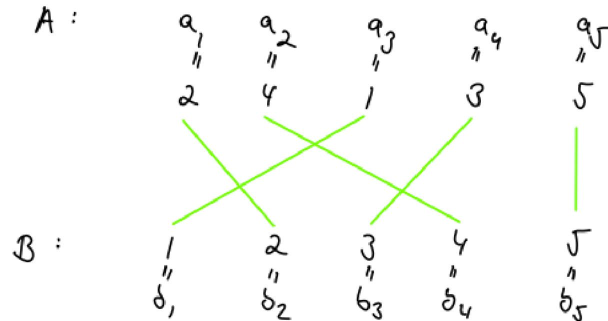
Example:

$$
\begin{aligned}
A &= (2, 4, 1, 3, 5) \\
B &= (1, 2, 3, 4, 5) \qquad \text{, i.e. } n = 5 \text{ in this case.}
\end{aligned}
$$

Goal: Come up with a quantitative way of measuring how similar the two rankings are. If the two rankings are identical, the measure should be 0. The measure should increase as the difference of two rankings increases.

Group Work: Propose a quantitative measure for comparing two rankings.

Idea: Visualize the two rankings in the following way:

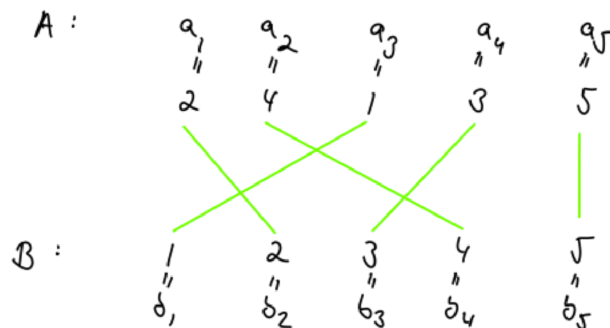


- connect a_i and b_j if $a_i = b_j$.
- count the total number of inversions

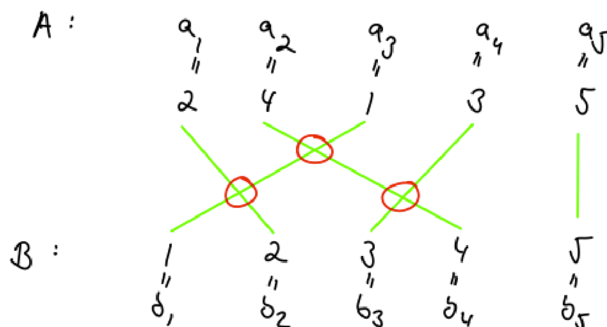
Definition An inversion corresponds to one pair of numbers (a_i, a_j) such that $a_i > a_j$ and $i < j$.

In the above example: $(2, 1)$ is an inversion as $(a_1 = 2, a_3 = 1)$ and $1 < 3$.

Group Work Determine all inversions in the above example. How do you identify them?



Answer Identify all crossing pairs of lines (O) in the above diagram:



- The inversions are $(2,1)$, $(4,1)$, and $(4,3)$.
- number of inversions is 0 (as desired) if the rankings are the same and increases (as desired) as the rankings become more dissimilar.

Group Work: What is the largest number of inversions for two given lists A and B of length n?

Answer If we are dealing with two rankings of length n, we have n corresponding lines linking an a_i to a b_j if $a_i = b_j$.

These n lines can have at most

$$\binom{n}{2} = \frac{n \cdot (n-1)}{2} \qquad \text{pairwise categories}$$

as any of the lines can be crossed with any of the other $(n-1)$ lines and as we count each such crossing only once (not twice), namely only the case (a_i, a_j) with $a_i > a_j$ and $i < j$, i.e. the corresponding inversion.

(Group Work) We get $\binom{n}{2}$ inversions if A and B are in opposite orders, i.e. A is in descending order, whereas B is in ascending order.

Goal: Devise an efficient algorithm to count the number of inversions in a ranking A of length n.

Idea 1 Check all pairs (a_i, a_j) if $a_i > a_j$ and $i < j$

Group Work How efficient would this be?

Answer Need to check all $\binom{n}{2}$ pairs, i.e. require $O(n^2)$ time.

Motivation Is there a more efficient algorithm? (think about this)

We already know that the maximum number of inversions is $\binom{n}{2}$, i.e. $O(n^2)$. So, if there is a more efficient algorithm which requires less than $O(n^2)$ time, it *cannot* look at the max number if inversions individually.

Idea 2 Use a recursive algorithm

Assume in the following that $n = 2^k$ for some $k \in \mathbb{N}$.

(1.) partition $A = (a_i, \ldots, a_n)$ into two lists of length $\frac{n}{2}$ each. $A_1 = (a_1, \ldots, a_{\frac{n}{2}})$ and $A_2 = (a_{\frac{n}{2}+1}, \ldots, a_n)$

(2.) Count the number of inversions within each half separately

(3.) Count the number of inversions where one number belongs to one half, and the other number belongs to the other half.

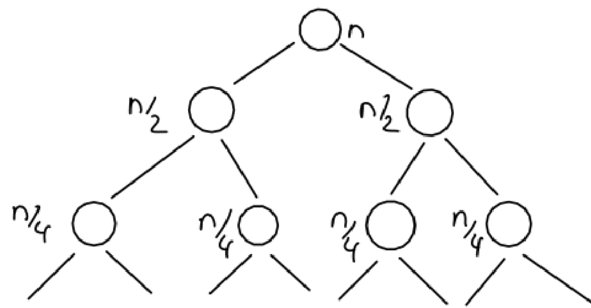
(this corresponds to the step where we combine two solutions into one, i.e. we need to determine how efficiently we can do this)

Group Work: Which kind of generic algorithm that we introduced in section 5.1 does the above recursive algorithm fit?

Answer

- we are dealing with a binary tree as we partition each problem size into two disjoint problems of half the original size at each recursion
  i.e. $q = 2$ and $n \to \frac{n}{2}$ as we go from one instance at level k to one instance at level $k + 1$
- we know that our binary tree requires $log_2(n)$ levels
- yet unclear how much time it requires to combine solutions into one (use symbol ?)

$\Rightarrow$ we know that we are dealing with an algorithm of the following kind:



and a recurrence relation: $T(n) \leq 2 \cdot T(\frac{n}{2}) + ?$

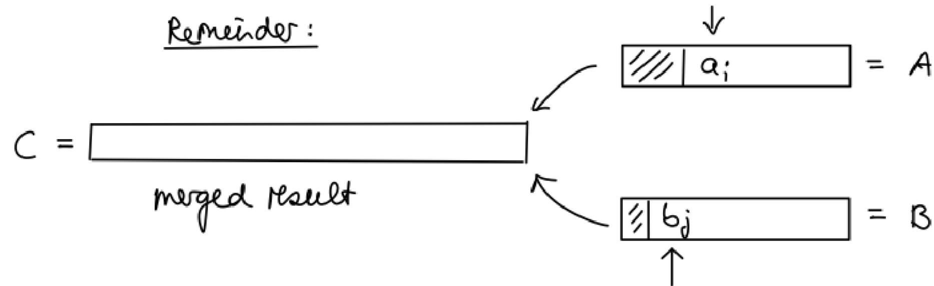
Goal: Find an efficient algorithm for combining the solutions of two subproblems into one, i.e. find out what ? should be.

The situation is as follows:

- *have* 2 sorted lists A_1 and A_2 (for which we know the solution, i.e. the number of inversions in A_1 and in A_2, separately). Call $A = A_1$ and $B = A_2$ in the following, two make the notation easier.
- *want* to produce a simple sorted list $\tilde{A}$ *while* counting the number of inversions between $A = A$, and $B = A_2$.
  For this we need to count the number of pairs (a, b) with $a \in A, b \in B$ and $a > b$.

Group Work Remember the merge sort algorithm that we introduced earlier to combine two sorted lists A and B into a single sorted list C.

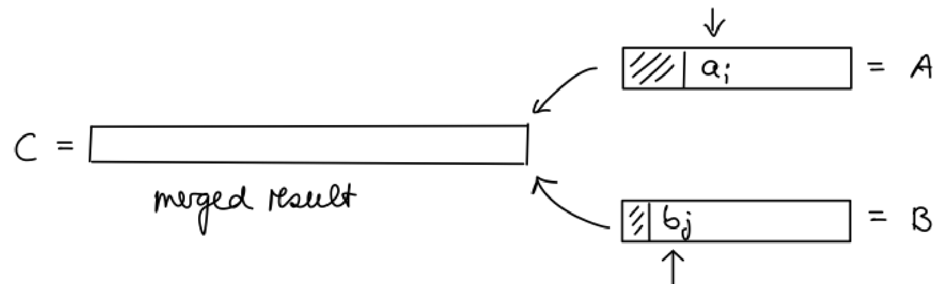
Can we adapt this algorithm to address the problem above?



We need to count the number of pairs (a, b) with $a \in A, b \in B$ and $a > b$.

Answer: Is is clear that we can use the old merge sort algorithm to get the sorting aspect done as before in $O(n)$, see section 26.1 before.

The big question is whether we can also count the inversions while making the merged list C in the old way.



- if $a_i < b_j$, i.e. we are moving a_i to the merged list, we also know that we are not dealing with an inversion as everything left in list B is also larger than a_i
- else, i.e. if $a_i > b_j$ (so, if we would move b_j to the merged list), we also know that b_j is smaller than *all* elements in A as these elements are already sorted. So, if we encounter $a_i > b_j$, we know that this corresponds to $|A|$ inversions, where $|A|$ is the number of elements in A at that point.

Conclusion: By adapting the old merge sort algorithm such that we keep track of the inversions, we can solve the problem in $O(n)$ time.

The new algorithm is called the

Definition Merge-and-Count (A, B) algorithm:

- for each list A and B separately keep a pointer and initialize it to point to the respective first element
- keep track of the total number of inversions using variable count and set count = 0 to start with
- while both lists (A and B) are not empty {

– let a_i and b_j be the elements currently pointed to
– append $\min\{a_i, b_j\}$ to new, combined list C
– if $(a_i > b_j)\{$
count $+=$ remaining number of elements in A
$\}$
– advance the current pointer in the list from which $\min\{a_i, b_j\}$ was selected

$\}$

Group Work Convince yourself, e.g. by reminding you of the merge sort algorithm discussed in section 2.6.1, that the above algorithm also requires $O(n)$ time.

We can now specify a recursive algorithm that simultaneously sorts a list A and count the number of inversions w.r.t. the default ranking B.

Definition Sort-and-Count (A) where n denotes the length of A

- if $(n = 1)\{$
  return 0 as number of inversions and A itself as list
  $\}$
- else $\{$
  – divide A into two halts A_1 and A_2 of equal size
  A_1 contains the first $\lceil \frac{n}{2} \rceil$ elements
  A_2 contains the remaining $\lfloor \frac{n}{2} \rfloor$ elements
  – $(r_{A_1}, A_1) =$ sort-and-count(A_1)
  – $(r_{A_2}, A_2) =$ sort-and-count(A_2)
  – $(r, L) =$ merge-and-count(A_1, A_2)
  – return $r + r_{A_1} + r_{A_2}$ as number of inversions and list L

  $\}$

Conclusion As merge-and-count takes $O(n)$ time (see previous ?) in last recurrence relation, the above sort-and-count procedure fits the generic algorithm Nr. 1 (see 5.1) and the corresponding recurrence relation

$$T(n) \leq 2 \cdot T(\frac{n}{2}) + c \cdot n$$

We thus know from our previous proof in 5.1, that the algorithm requires $O(nlog(n))$ time for a list of n elements.

5.2.2 Example 2: Finding the closest pair of points

Goal Given n points in the plane, identify the pair of points that is closest together.

Group Work How much time would the straightforward algorithm take?

Answer Consider all $\binom{n}{2}$ pairs of points and keep track of pair with smallest distance. This requires $O(n^2)$ time.

Goal Come up with a clever algorithm that requires less time.

History M.I. Sharnos and D. Hoey (early 1970) found an algorithm that solve the problem in $O(n \log(n))$ time.

Idea 1: Let us first consider the one-dimensional case, i.e. given n points along a line ($\mathbb{R}$), find the closest pair of points.

Group Work Come up with an efficient algorithm that solve the problem.

Answer
- First, sort all n points. we know that this requires $O(n \log(n))$ time.
- Second, walk through the sorted list, calculate the distance between adjacent points and keep track of the minimum distance we encounter. We can do this in $O(n)$ time.
- $\Rightarrow$ overall: require $O(n \log(n))$ time to solve the one-dimensional problem

Idea 2: Try to retain some of the ideas behind the algorithm for the one-dimensional case to tackle the two-dimensional case.

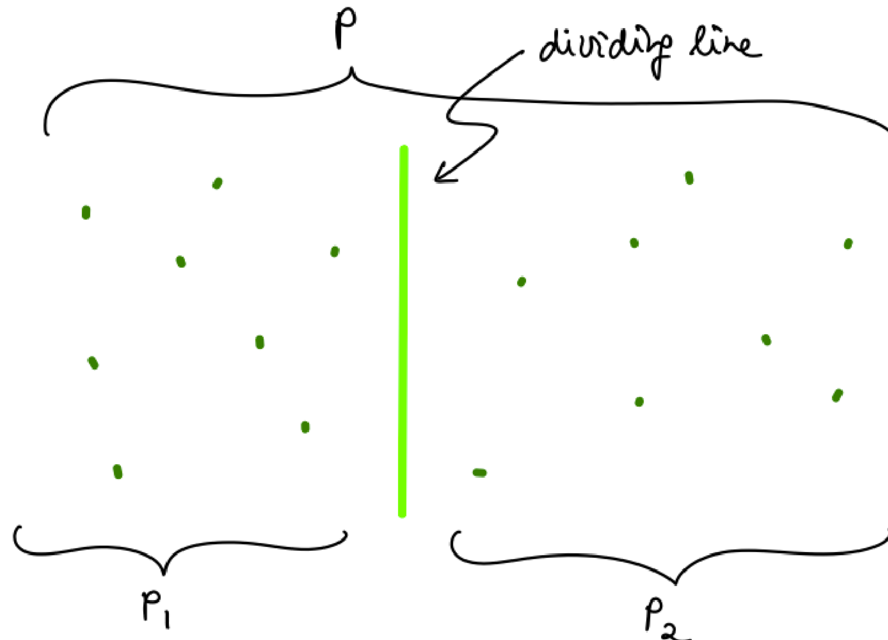
- every one of the n points can be denoted as $p = \binom{x}{y} \in \mathbb{R}^2$
- the distance between two points p_1 and p_2 is their euclidean distance which is defined as

$$d(p_1, p_2) := \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

- *assume in the following:* there is no pair of points that have the same x- or y-coordinate.
- key idea 1: use a divide and conquer approach

(1.) (level k)

- divide the set of n points into two halves, those on the left side of a dividing line in the plane and those on the right side of that line (|).



We know that we can sort the x- (or, alternatively y-) coordinates of all n points in $O(n\log(n))$ time, i.e. the partitioning of the big set into one left subset P_1 and one right subset P_2 of the same size can be done in $O(n\log(n))$ time. If P is given set points, denote:

P_x := points in P sorted by x-coordinate

P_y := points in P sorted by y-coordinate

We remember, for each point $p \in P$, its position in both lists, so we can refer to them throughout the algorithm. These numbers remain unchanged.

(2.) (level $k+1$)

- next, find the closest pair of points within set P_1, and, independently, within set P_2. Let

  p_0^1 and p_1^1 denote the closet pair of points in P_1 and

  p_0^2 and p_1^2 denote the closet pair of points in P_2

  Compile sorted lists P_{1x} and P_{1y} and P_{2x} and P_{2y} by looping over the entries in P_x and P_y in $O(n)$ time and extracting the relevant entries.

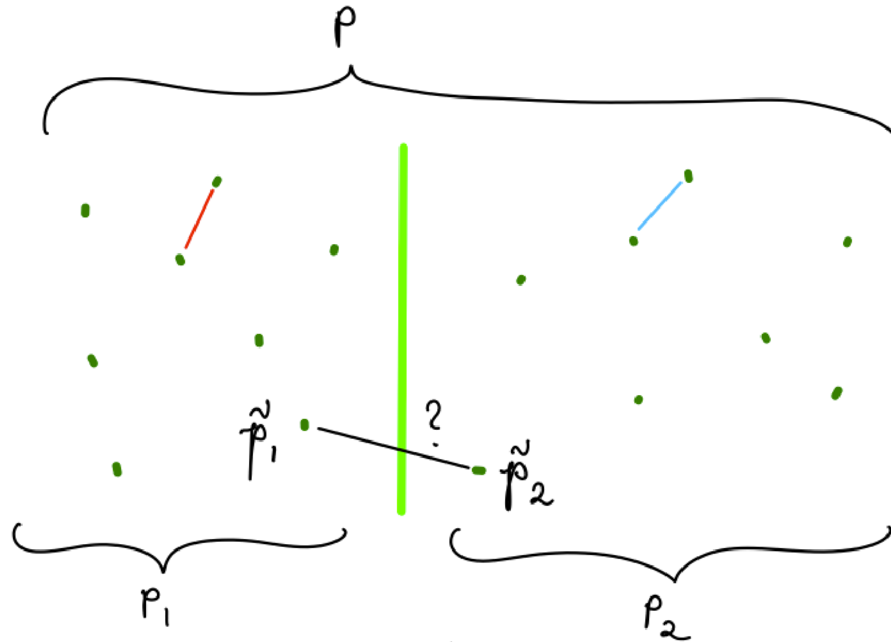
  As before, we remember for each point in P_1 and each point in P_2 its respective entries in the two x- and y- sorted lists.

(3.) merging solutions

- When we already have the solutions for subsets P_1 and P_2, we still need to check whether any distance between a point in P_1 and a point in P_2 is smaller than $d(p_0^1, p_1^1)$ and $d(p_0^2, p_1^2)$.

$$\delta := \min\{d(p_0^1, p_1^1), d(p_0^2, p_1^2)\}$$

the minimum distance between the shortest distance of any smaller points within P_1 and P_2



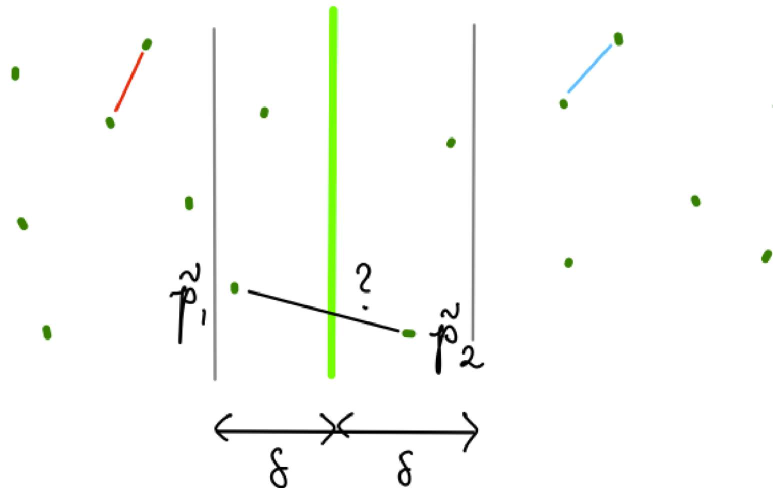
What we need to check is whether there is a pair of points $\tilde{p_1} \in P_1$ and $\tilde{p_2} \in P_2$ such that

$$d(\tilde{p_1}, \tilde{p_2}) < \delta$$

If this is *not* the case, δ is already the desired solution.

Group Work How do we check in the most efficient way if such a pair $\tilde{p_1} \in P_1$ and $\tilde{p_2} \in P_2$ exists?

Answer If $d(\tilde{p_1}, \tilde{p_2}) < \delta$, then $\tilde{p_1}$ and $\tilde{p_2}$ must lie within a distance of δ from the dividing line.



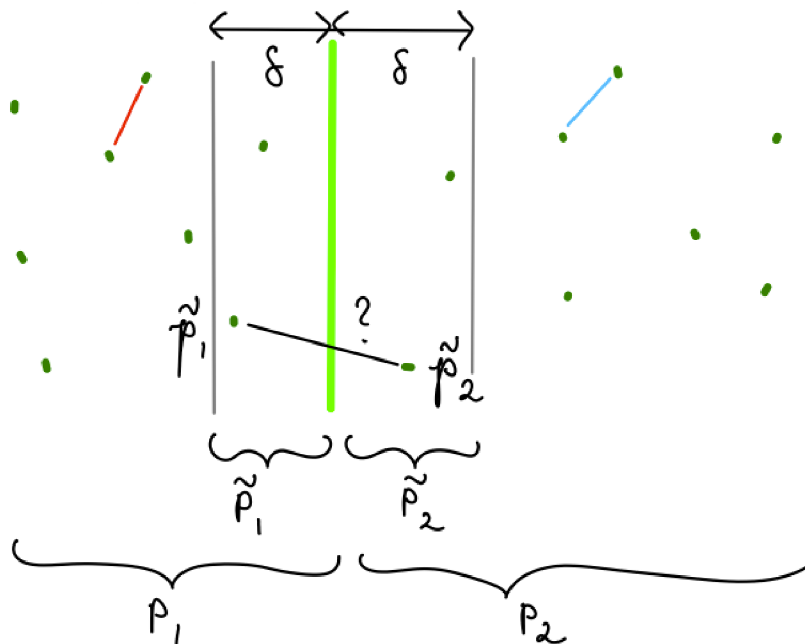
Proof: Let x' denote the x-coordinate of the dividing line (i.e. the dividing line is parallel to the y-axis). Then

$$\tilde{p_{2x}} - x^1 \leq \tilde{p_{2x}} - \tilde{p_{1x}} \leq \sqrt{(\tilde{p_{2x}} - \tilde{p_{1x}})^2 + (\tilde{p_{2y}} - \tilde{p_{1y}})^2} \leq d(\tilde{p_1}, \tilde{p_2}) < \delta$$

$$\text{and } x^1 - \tilde{p_{1x}} \leq \tilde{p_{2x}} - \tilde{p_{1x}} \leq \sqrt{(\tilde{p_{2x}} - \tilde{p_{1x}})^2 + (\tilde{p_{2y}} - \tilde{p_{1y}})^2} \leq d(\tilde{p_1}, \tilde{p_2}) < \delta$$

This shows that both points $\tilde{p_1}$ and $\tilde{p_2}$ have to lie within a narrow band within the distance of δ to the dividing line $\Box$.

Conclusion In order to find candidate pairs of points $\tilde{p_1} \in P_1$ and $\tilde{p_2} \in P_2$ for which $d(\tilde{p_1}, \tilde{p_2}) < \delta$, we only need to consider a subset of points $\tilde{P_1} \subset P_1$ and $\tilde{P_2} \subset P_2$ that lie within a distance of δ to the dividing line.

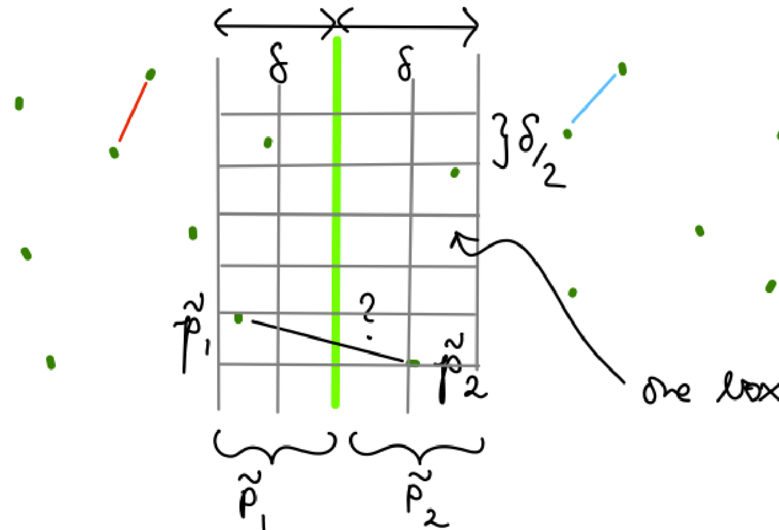


Note: $\tilde{P_1}$ may be equal to P_1 if all points in P_1 happen to be close to the line. Same for $\tilde{P_2}$ and P_2.

- Try to learn more about the candidate pairs of $\tilde{p_1} \in P_1$ and $\tilde{p_2} \in P_2$ with $d(\tilde{p_1}, \tilde{p_2}) < \delta$.

  1. Any two points in $\tilde{P_1}$ are at least a distance of δ apart as we would otherwise contradict the definition of δ as the shortest distance between any pair in P_1 and any pair on P_2.

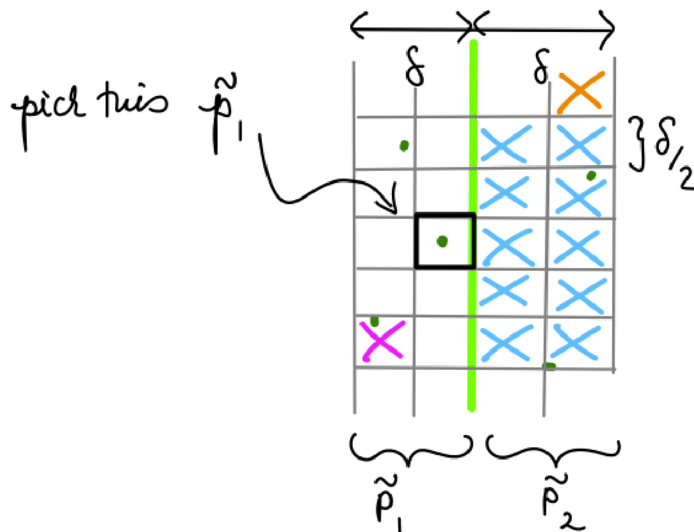
     If we draw a grid around the dividing line with a grid-distance of $\frac{\delta}{2}$ in both directions, every box contains at most one point in $\tilde{P_1}$ or $\tilde{P_2}$.



This is because any two points within the same box would have a distance $\leq \frac{\delta}{\sqrt{2}} < \delta$.

2. Given one particular $\tilde{p_1} \in P_1$, in how many neighbouring boxes do we need to look to find a $\tilde{p_2} \in P_2$ with $d(\tilde{p_1}, \tilde{p_2}) < \delta$?

   (We get the same answer if we start by considering a $\tilde{p_2} \in P_2$ first.)



$\Rightarrow$ for a given $\tilde{p_1} \in P_1$, we need to consider only up to 10 boxes X in $\tilde{P}_2$. As each box contains at most one $\tilde{p_2} \in P_2$, need to calculate only distance of $\tilde{p_1}$ and up to 10 points in $\tilde{P}_2$

Group Work For $\tilde{p_1} \in P$, why don't we have to consider points in X and X?

Answer We don't consider X because we need to find a box with a point in $\tilde{P}_2$, not $\tilde{P}_1$.

We don't consider X because any point in $\tilde{p_2} \in P_2$ would have a distance $d(\tilde{p_1}, \tilde{p_2}) > \delta$ which would not be any improvement w.r.t. δ.

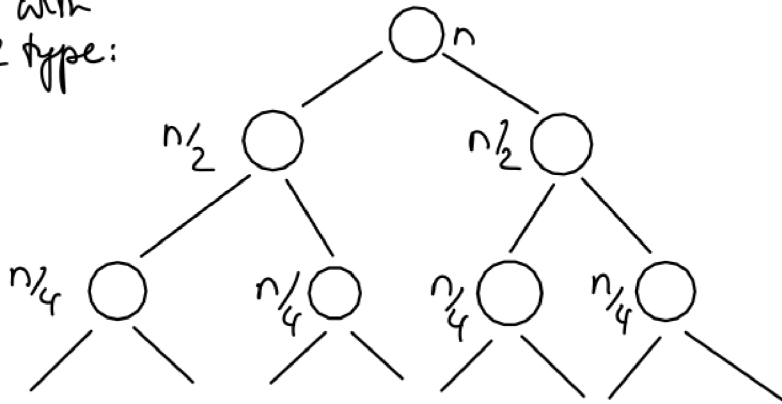
Remember: We are hunting for $\tilde{p_1} \in P_1$ and $\tilde{p_2} \in P_2$ with $d(\tilde{p_1}, \tilde{p_2}) < \delta$.

Note: If $\tilde{p_1} \in P_1$ was in a box further away from the line, i.e. one box to the left w.r.t. the box of the above figure, we only need to consider 5 boxes, i.e. up to 5 points in $\tilde{P}_2$.

Conclusion In order to find a pair $\tilde{p_1} \in \tilde{P}_1$ and $\tilde{p_2} \in \tilde{P}_2$ with $d(\tilde{p_1}, \tilde{p_2}) < \delta$, we

- loop over the y-sorted list of points in $\tilde{P}_1$
- for a given $\tilde{p_1} \in \tilde{P}_1$, compute the distance to the closest 10 points in the y-sorted list of $\tilde{P}_2$.
- if we find a pair $\tilde{p_1}$ and $\tilde{p_2}$ with $d(\tilde{p_1}, \tilde{p_2}) < \delta$, keep track of the new minimum distance and report it once we have finished looping over the elements in $\tilde{P}_1$.
- as all point in P_1 could also be in $\tilde{P}_1$, we require $O(|P_1|)$ to complete the calculation, i.e. it requires linear time.

Conclusion from (1.) (2.) and (3.) : The overall algorithm fits generic algorithm Nr. 1 from section 5.1, hence we know an input set of n points in two dimensions requires $O(n \log(n))$ time to identify the closest pair of points.



and a recurrence relation: $T(n) \leq 2 \cdot T(\frac{n}{2}) + c \cdot n$ because combining solutions takes linear time.

Reference: The above already spells out all the details of the algorithm. you can find the pseudo-code of the algorithm on page 230 in section 5.4 in the course book.