6 Randomized algorithms

Key Idea Modify divide-and-conquer algorithms (see previous section) to reduce their worst-case running time by making the divide-step random.

Implication As we are no longer using a deterministic way to divide a given problem size into sub-problems, we now have to consider the expectation values of random variables in order to analyze the expected time spent on recursive calls.

6.1 Example 1: Finding the median

Goal Given a set S of $n \in \mathbb{N}$ numbers, $S = \{a_1, a_2, \cdots, a_n\}$, determine their median.

Assumption In the following, assume for simplicity sake that all n numbers in S are distinct.

Definition The median of a set of $n \in \mathbb{N}$ numbers $S = \{a_1, a_2, \cdots, a_n\}$ is equal to the kth largest element in S, where

$$k = \begin{cases} \frac{(n+1)}{2} & \text{if } |S| = n \text{ is odd} \\ \frac{n}{2} & \text{else (i.e. if } n \text{ is even)} \end{cases}$$

Group Work Given the algorithms that we already encountered, come up with an efficient algorithm to find the median of a given set S. Also specify a corresponding asymptotic upper bound for the worst-case running time, i,e, use O-notation.

Answer Sort the n numbers in S, requiring $O(n \log(n))$ time, and then pick the kth elements as defined above to identify the median of S.

Goal Try to design an algorithm that requires less than $O(n \log(n))$ time.

Idea 1

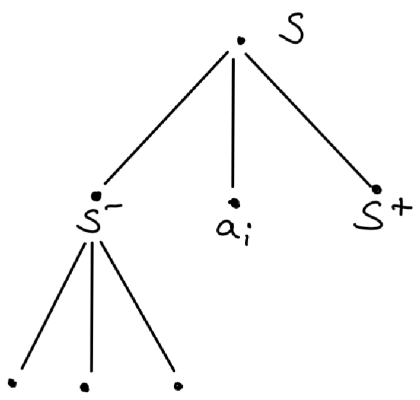
Find a slightly more general algorithm, called Select(S, k), that takes as input a set S of n numbers and a number $k \in \{1, \cdots, n\}$ and returns as output the kth largest value in S.

Note:

- using this algorithms, we can determine the median of s using either

$$\text{Select}\left(S, \frac{(n+1)}{2}\right) \text{ (if } |S| = n \text{ is odd), or}$$
$$\text{Select}\left(S, \frac{n}{2}\right) \text{ (if } |S| = n \text{ is even)}$$

- the basic structure of algorithm Select(S, k) is to
    1. choose an element $a_i \in S$ (the splitter)
    2. form sets S^- $:= \{a_j \in S | a_j < a_i\}$ and S^+ $:= \{a_j \in S | a_j > a_i\}$



    3. determine if S^- or S^+ contains the kth largest element in S and iterate only on either S^- or S^+ (but not both)

- we still need to specify an efficient way for choosing the splitter
- Our overall goal is to implement the Select algorithm so it runs in $O(n)$ expected time.
- Given the above details about the Select-algorithm, we can specify it as follows:

Definition Select(S, k):

```
• choose a splitter $a_i \in S$        $(*)$
• for each element $a_j \in S$ {
    – if $(a_j < a_i)$ { put $a_j$ in $S^-$ }
    – else if $(a_j > a_i)$ { put $a_j$ in $S^+$ }
  }
• if $(|S^-| = k - 1)$ { $a_i$ is the desired $k$th element }
  else (if $|S^-| \geq k$) {
    – the $k$th largest element in $S$ must lie in $S^-$
    – recursively call Select($S^-$, $k$)
  }
  else {        (i.e. if $|S^-| =: l < (k - 1)$)
    – the $k$th largest element in $S$ must lie in $S^+$
    – recursively call Select($S^+$, $k - (l + 1)$)
  }
```

Group Work

- What does Select(S, 1) correspond to?
- What does Select(S, n) correspond to?
- Why do we make a call to Select(S^+, $k - (l+1)$) in the above algorithm rather than to, say, Select(S^+, k) or Select (S^+, $k-1$)?

Answers

- Select(S, 1) = min$\{S\}$
- Select(S, n) = max$\{S\}$
- S^- contains l elements a_j with $a_j < a_i$

  S^+ contains $n-(l+1)$ elements a_j with $a_i > a_j$

  Asking for the kth largest element in S is thus the same as asking for the $k-(l+1)$th largest S^+, if we know that $|S^-| = l < k-1$, i.e. if we know that it is not contained in S^-, nor equal to a_i

Observations

- The above algorithm Select(S,k) terminates as it makes recursive calls on increasingly stricter smaller sets
- If we choose the splitter element wisely, sets S^- and S^+ have roughly the same size and we halve the problem size in each iteration.

Group Work Which recurrence relation do we obtain if we always choose the median value as splitter?

Answer In that case, we are halving the problem size at every recursion, i.e. we have

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

As this fits the description of generic algorithm Nr. 1 from section 5 before (for $q = 1$), we know that the select algorithm would require (see case 3 in section 5)

$$O(n) \qquad \text{time}$$

Conclusion If we can come up with a clever way of setting the splitter element to the median value, the overall algorithm runs in linear time. As we want to use the Select algorithm to identify the median of a given set S, this is logic-wise circular. Hence, we should rephrase the goal as follows:

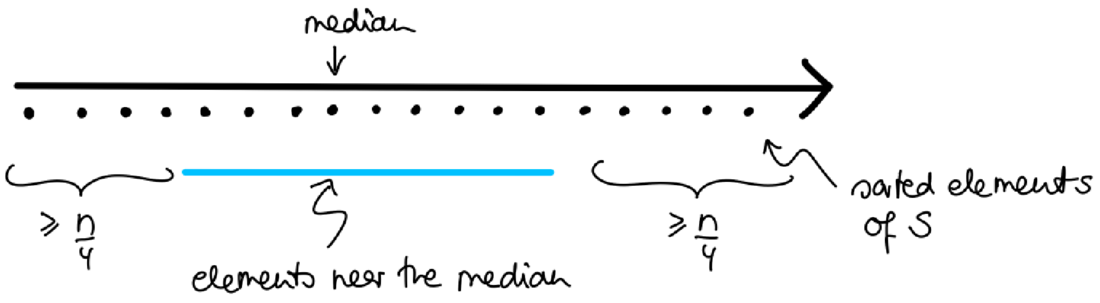
Goal Choose splitter element in the Select algorithm using a random algorithm in such a way that we can (on average) expect the splitter element to divide S into two approximately equally large sets S^- and S^+

Idea 2

In the above Select algorithm, choose splitter element a_i uniformly at random from S

Group Work Why is this a decent idea, i.e. an idea that is in line with our previously stated goal?

Answer Using idea 2, every element of S has the same probability of being selected as splitter element. As there are more well-centered elements in S than at the edges, we have a decent chance of picking an element near the median element.



Definition Given a set S of n distinct numbers, i.e. $S = \{a_1, a_2, \cdots, a_n\}$. We say that an element $a_i \in S$ is central, if

- at least one quarter of the elements of S are smaller than a_i, and
- at least one quarter of the elements of S are larger than a_i

Definition (Final Select algorithm Select(S, k))

- use Select(S, k) as defined before, but
- choose a splitter element (see ($*$) in previously explained algorithm) uniformly at random

Note As the Select algorithm is no longer deterministic, we can no longer talk about its running time, but only about its *expected* running time.

Running Time of Select(S,k)

Defintions • We will say that the algorithm is in phase j, if the problem size we are dealing with is

$$\leq n \cdot \left(\frac{3}{4}\right)^j \qquad and > n \cdot \left(\frac{3}{4}\right)^{j+1}$$

Note: This is a meaningful definition as we expect the problem size to shrink by a factor of $\frac{3}{4}$ or more if the splitter element can be expected to be central.

- Let X be a random variable that is equal to the number of recursions in the algorithm, i.e.

$$X = X_0 + X_1 + X_2 + \cdots$$

Where X_i denotes the expected number of steps that the algorithm spends in phase i

Observations

1. at any phase j, the probability of choosing a central element by making a uniformly random choice is $\frac{1}{2}$
   $\Rightarrow$ the expected number of iterations before the randomly chosen element in phase j is a central element is 2 (which is independent of the value of j)
2. When the algorithm is in phase j the problem size is at most

$$n \cdot \left(\frac{3}{4}\right)^j$$

$\Rightarrow$ require at most $c \cdot n \cdot \left(\frac{3}{4}\right)^j$ steps in phase j for some constant $c \in \mathbb{R}+$
1. and 2. $\Rightarrow$ the expected number of steps of the algorithm in phase j is

$$E[X_j] \leq 2 \cdot c \cdot n \left(\frac{3}{4}\right)^j$$

The expected total number of steps of the algorithm is (sum over phases):

$$\begin{aligned}\Rightarrow E[X] &= \sum_{j=0}^{m} E[X_j] \leq \sum_{j=0}^{m} 2 \cdot c \cdot n \left(\frac{3}{4}\right)^j \\ &\leq \sum_{j=0}^{\infty} 2 \cdot c \cdot n \left(\frac{3}{4}\right)^j = 2 \cdot c \cdot n \sum_{j=0}^{\infty} \left(\frac{3}{4}\right)^j \\ &\leq 2 \cdot c \cdot n \cdot 4 = 8 \cdot c \cdot n \qquad \text{using geometric series}\end{aligned}$$

Conclusion The expected running time of Select(S, k) for $|S| = n$ and $k \in \{1, \cdots, n\}$ is $O(n)$.

6.2 Example: Sorting Numbers

Goal

- Given a set S of $n \in \mathbb{N}$ distinct numbers, sort them
- Devise a divide-and-conquer algorithm to address the above goal

Idea Use the same idea as in the example before (see 6.1), i.e. divide a given input set S according to a splitter element into two sets, those with elements smaller than the splitter element and those that are larger.

The difference w.r.t. the example in 6.1 is that rather than discarding one of the two sets, we keep both and sort them recursively before merging them, with the splitter element in between, into one sorted list.

As in the previous example (6.1), we will choose splitter elements in a uniformly random way such that each chosen splitter element is expected to be central.

Definition Quicksort(S)

```
• if ($|S| \leq 3$) { sort $S$ and return sorted list }
• else {
• while no central splitter element has been found {
      – choose a splitter at random
      – for each element $a_j \in S${
        if ($a_j < a_i$) { move $a_j$ to $S^-$ }
        else if ($a_j > a_i$) { move $a_j$ to $S^+$ }
        }
      – if $\left(|S^-| \geq \frac{|S|}{4} \text{ and } |S^+| \geq \frac{|S|}{4}\right)$ {
        $a_i$ is a central splitter
        }
  }
  • recursively call Quicksort($S^-$)
  • recursively call Quicksort($S^+$)
• return (the sorted set $S^-$, $a_i$, the sorted set $S^+$) as sorted return list
  }
```

Note The above algorithm only proceeds to the next recursion once it has checked that the recursively chosen splitter is indeed central.

Group Work How much time do we expect the *while-loop* to take, i.e. how long does it take to propose a splitter a_i and check whether it is central?

Answer $O(|S|)$

Conclusion Because we expect to pick a central splitter after two iterations of the while loop, the above algorithm requires $O(|S|)$ time, *excluding* the time spent on *recursive calls*.

Definition We will say that a subproblem of the above algorithm is of type j if the size of the set under consideration is

- $\leq n \left(\frac{3}{4}\right)^j$ and

- $> n\left(\frac{3}{4}\right)^{j+1}$

Observation 1 The expected time spent on a subproblem of type j, excluding recursive calls, is $O\left(n \cdot \left(\frac{3}{4}\right)^j\right)$

Observation 2 Splitting one subproblem of type j via a central splitter creates two *disjoint* subproblems of type $j+1$

Group Work What is the total number of *different* subproblems of type j that we may encounter? Call this quantity x.

Answer
1. We have a problem of size n to start with
2. We now that a problem of type j has a minimum size of $n\left(\frac{3}{4}\right)^{j+1}$
3. We know that the total problem size remains n for different values of j

$$\Rightarrow \qquad n = x \cdot n\left(\frac{3}{4}\right)^{j+1}$$
$$x = \left(\frac{4}{3}\right)^{j+1}$$

$\Rightarrow$ Conclusion: We have at most $\left(\frac{4}{3}\right)^{j+1}$ subproblems of type j

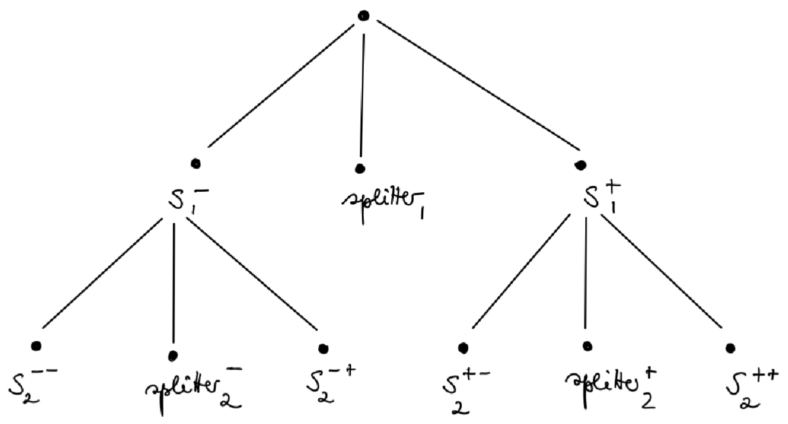
Group Work What is the expected amount of time required to sort out all sub problems of type j?

Answer We have at most $\left(\frac{4}{3}\right)^{j+1}$ subproblems of type j which are disjoint and each have a size of at most $n \cdot \left(\frac{3}{4}\right)^j$. As the expected running time of the algorithm (excluding recursive calls) is linear, we require $O(n \cdot \left(\frac{3}{4}\right)^j)$ for each subproblem of type j.

Overall, we therefore require $O\left(\left(\frac{4}{3}\right)^{j+1} \cdot n \cdot \left(\frac{3}{4}\right)^j\right) = O\left(\frac{4}{3} \cdot n\right) = O(n)$ time to sort out all sub-problems of type j

Group Work What is the maximum number of different types that we have to consider before subproblems have a minimum size?

Answer Tackle the question visually:



type	expected size x of each sub-problem S^+, S^-
0	$(x = n)$
1	$x \in]n\cdot(3/4)^2, n(3/4)]$
2	$x \in]n(3/4)^3, n(3/4)^2]$

- The total problem size remains n for every type
  $\Rightarrow$ Require a total number of $O(\log(n))$ types

Overall Conclusion The total *expected* running time of the Quicksort algorithm is $O(n \log(n))$.

Proof (summary of what we already showed):

- Have $O(\log(n))$ different types of consider
- All subproblems of a given type require $O(n)$ expected running time
  $\Rightarrow$ Overall, need $O(n \log(n))$ expected running time.

6.3 Example: Finding a closest pair of points in the plane

Reminder We already present an efficient algorithm for solving this problem, see section 5.2.2.

That algorithm required $O(n \log(n))$ time to find the closest pair of n given points.

Question Can we do this even more efficiently using a randomized algorithm?

Answer Yes! (as we will see in the following)

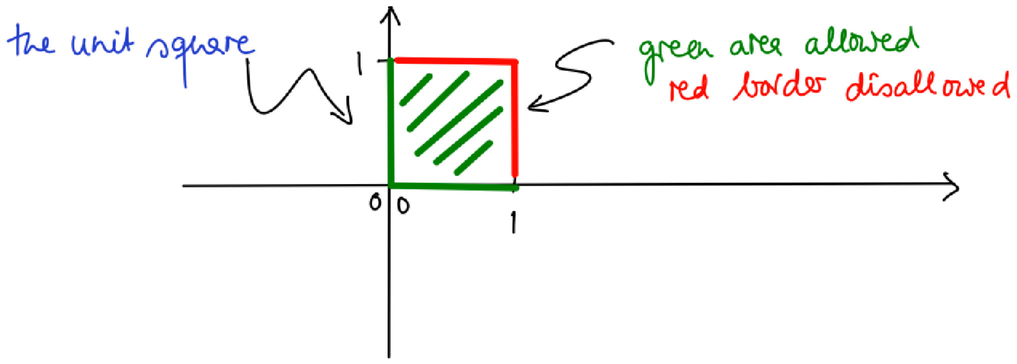
Reminder

- Given n points in the plane, $\{p_1, p_2, \ldots, p_n\}$, find a pair of points $i \neq j$ such that the distance $d(p_i, p_j)$ is minimal
- 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

Additional Assumption All n points lie in the unit square, i.e. $x_i \in [0,1]$ and $y_i \in [0,1]$ for all $i \in \mathbb{N}_n$



Group Work Is the above assumption restrictive in any way?

Answer No, we can assume without loss of generality (WLOG) that it is always fulfilled as we can rescale the x and y coordinates of any finite set of points so they end up in the unit-square.

Plan for the following:

- rather than coming up with a randomized algorithm of our old algorithm from section 5.2.2,
- devise a *new* algorithm which employs an underlying so-called dictionary data structure.
- the "randomized" part of this new algorithm involves randomized operations on the dictionary data structure

Key Idea 1 (new algorithm)

1. suppose we only have $n = 2$ points, then the answer is $\delta = (p_1, p_2)$
2. now: assume we know the answer δ for a given set of n randomly ordered points $p_1, \ldots, p_n$
   How may δ change if we add a new point p_{n+1}?
   ```
   loop over all $d(p_i, p_{n+1})$ for $i \in \{1, \ldots, n\}$ {
   if $(d(p_i, p_{n+1}) < S)$ { update $\delta$ }
   }
   ```
   $\Rightarrow \delta$ is then the new answer for $n+1$ points $p_1, p_2, \ldots, p_{n+1}$

Definition Step 2 in the algorithm above is called a stage

Group Work What is the minimum and maximum number of stages required to find the solution δ for a given set of n points?

Answer
- minimum is 1 as we may be lucky and $\delta = d(p_1, p_2)$
- maximum is $(n-2)$ if adding a new point always decreased δ further

Question How can we efficiently test if the currently closest pair of points at distance δ remains the closest pair of points when we add a new point and, if required, to identify the new closest pair of points?

Group Work Any ideas how to tackle this problem? Think about 5.2.2 and the grid and boxes we introduced.

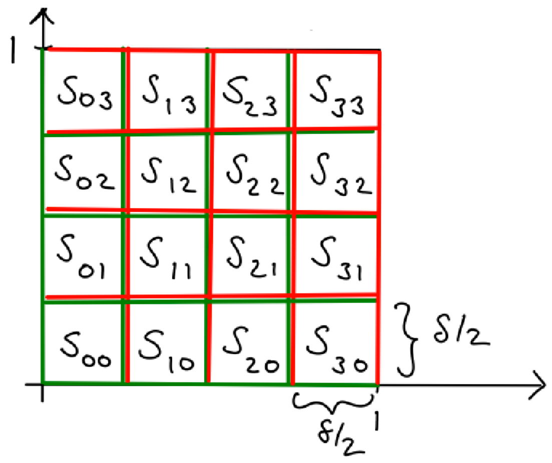
Key Idea 2 (new algorithm)

Subdivide the unit square into sub-squares whose sides are $\frac{\delta}{2}$ long

$\Rightarrow$ for a given value of δ, we obtain N^2 sub-squares, where $N = \frac{2}{\delta}$ (not $\frac{1}{2\delta}$ as the book says)

Definition subsquare $a, b \in \{0, \ldots, N-1\}$

$$S_{ab} := \{ \begin{pmatrix} x \\ y \end{pmatrix} \in R^2 | x \in [a\frac{\delta}{2}, (a+1)\frac{delta}{2}],$$
$$y \in [b\frac{\delta}{2}, (b+1)\frac{\delta}{2}]\}$$ (different from definition in book!)



Note The grid (and the squares) change as δ changes. In particular, we get a finer grid as δ decreases with more squares.

Conclusion 1 For a given δ, if two points p and q belong to the same sub square S_{ab}, then $d(p,q) < \delta$, i.e. any sub square contains at most one point.

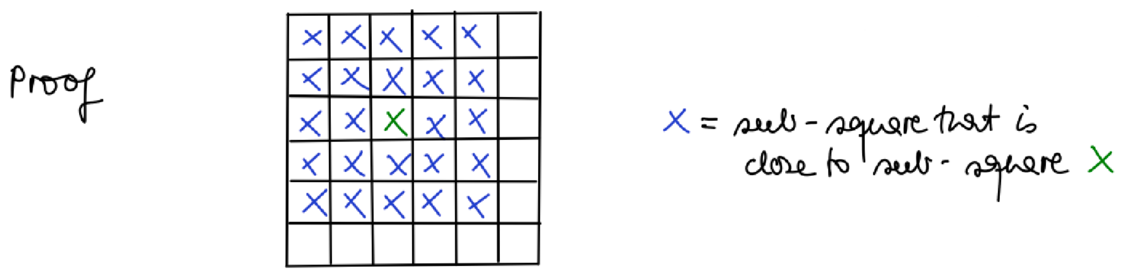
Proof (Group Work)

If p and q belong to the same sub-square, their x-coordinates differ by $< \frac{\delta}{2}$, same for their y-coordinates, we therefore have:

$$d(p,q) < \sqrt{\left(\frac{\delta}{2}\right)^2 + \left(\frac{\delta}{2}\right)^2} = \frac{\delta}{\sqrt{2}} < \delta \qquad \square$$

Conclusion 2 For a given δ, if two points p and q have $d(p,q) < \delta$, their respective sub squares are the *same* or close.

Group Work Proof



Proof Suppose the two points do not belong to the same adjacent sub squares, then WLOG

$$p \in S_{ab} \text{ and } q \in S_{a'b'} \text{ where } |a - a'| > 2 \text{ or } |b - b'| > 2 \text{ (or both)}$$

This implies that either their x-coordinates or their y-coordinates differ by δ or more. This implies

$$d(p,q) \geq \delta \text{ which is a contradiction to } d(p,q) < \delta \square$$

Conclusion 3 For a given δ and point $p \in S_{ab}$, we only need to look at other points in at most 25 sub-squares close to S_{ab} in order to find points q which may have $d(p,q) < \delta$.

Implication for Key Idea 1 (new algorithm)

- In previous step (2.), when we assume we know the answer δ for n points we remember (for that value of δ the sub-square S_{ab} to which each of the n points belongs.
- When we add the $(n+1)$th point p, we
  1. determine which sub-square S_{ab} it belongs to (per the δ values that we determined for n points)
  2. check up to 25 sub-squares that are close to S_{ab} in order to identify points q with $d(p,q) < \delta$

Note: As we have ≤ 25 sub squares to consider which each contain at most one of the n previously considered points, we can complete 2. in constant time.

Key Idea 3 (new data structure)

- *Motivation* In the new algorithm, we need a quick way of
  - finding the sub square for a given point p
  - finding all points contained in a given sub square S_{ab}
- given a set of n already considered points $p_1, p_2, \cdots, p_n$ create a dictionary which
  - comprises the sub-squares that contain any of the n points
  - for each sub square remembers the index i of the corresponding point p_i

  Note: As any sub square contains at most one of the n points, the total number of sub squares (N^2) is typically much larger than the interesting sub squares we want to store in the dictionary.
- Have 4 operations on the dictionary:

Lookup(S_{ab}): returns the point in a given sub square S_{ab}, if such a point exists.

Insert(S_{ab}, p): insert a new sub square S_{ab} into the dictionary which contains new point p

MakeDictionary(p, δ): create a new dictionary for a set of points p and a new grid with the sub squares of $(\frac{\delta}{2})^2$ size. For each point $p \in P$, we determine the sub square S_{ab} containing it and insert the corresponding sub square into the dictionary using the Insert operation.

DeleteDictionary: deletes the dictionary

Definition RandomizedClosestPair(P)

```
• order the n input points of $P$ in a random sequence $p_1, p_2, \ldots, p_n$
• $\delta = d(p_1, p_2)$
• $\delta' = \delta$
• MakeDictionary($P, \delta$)
• for ($i = 1, \ldots, n$) {
    – determine the sub square $S_{ab}$ that contains $p_i$
    – loop over $\leq 25$ subsquares $S_{a'b'}$ close to $S_{ab}${
        * $p_j$ = Lookup($S_{a'b'}$)
        * if ($p_j$ exists and $j < i$) {
            · if ($\delta' = d(p_j, p_i) < \delta$) {
              delete the current dictionary
              MakeDictionary($P_i, \delta'$) where $P_i := \{p_1, p_2, \ldots, p_i\}$
              break (i.e. stop for loop)
              }
              }
        }
      – if($\delta' == \delta$) {
          * insert $p_i$ into current dictionary
        }
    }
```

Group Work In the above algorithm, when dealing with one point p_i

- how many Lookup operations are required?
- how many distance computations are required?
- how many MakeDictionary operations are required?

Answer We need at most 25 of each of these operations

Conclusion : As the algorithm loops over n points p_i from p_1 to p_n, it overall requires at most

- $O(n)$ distance computations
- $O(n)$ Lookup operations and
- $O(n)$ MakeDictionary operations

Question How many Insert-operations does this algorithm require, including those by any MakeDictionary operation?

Group Work In iteration i in the algorithm, i.e. when dealing with p_i, how many Insert operations do we expect at most?

Answer
- we either have to make MakeDictionary call involving set P_i and δ' or a single call to Insert (if $\delta' == \delta$)
- if we make a call to MakeDictionary for P_i and δ, we have to make up to i insert calls

$\Rightarrow$ we require up to i Insert-operations

$\Rightarrow$ throughout the algorithm, the max number of Insert-operations seems to grow from $i = 1$ to $i = n$.

However In the following, we will show that:

- as we loop over i from 1 to n, the chance of having to make a MakeDictionary call decreases
- the random order of points in P (chosen at the start of the algorithm) affects the chance of having to make MakeDictionary calls in the for loop of the algorithm

Definition : X := random variable specifying the total number of Insert operations in the algorithm

$$X_i := \begin{cases} 1 & \text{if iteration } i \text{ causes a change of } \delta \\ 0 & \text{else} \end{cases}$$

Claim 1 The total number of Insert operations performed by the algorithm is

$$n + \sum_i i \cdot X_i$$

Group Work Proof

Proof Each point in p is inserted once when it is first encountered (hence the n) and, in each iteration i, i points need to be inserted if the minimum distance δ changes (hence each term iX_i). $\square$

Claim 2 The probability that considering the ith point $p - I$ in the algorithm causes the minimum distance δ to change is

$$P[X - i = 1] \leq \frac{2}{i}$$

Proof Suppose that we are in iteration i and that the minimum distance among points $p_1, p_2, \ldots, p_i$ is $d(p, q)$, i.e. $p, q \in P_i$

When considering p_i, we only need to update the minimum distance if either $p_i = p$ or $p_i = q$

Group Work Why is this correct?

Answer If $p_i \notin \{p, q\}$, both p and q are part of P_i and the minimum distance is already $d(p, q)$ by the time we start considering p_i.

The chance that $p_i = p$ or $p_i = q$ is $\frac{2}{i}$ as we have i points to choose from in P_i and these points are in a randomly chosen order. $\square$

Claim 3 The expected total number of Insert operations of the algorithm is $O(n)$.

Group Work Proof

Proof Based on Claim 1, we have

$$E[X] = n + \sum_i i \cdot E[X_i] \leq n + sum_i i \frac{2}{i} = n + 2 \cdot n = 3n \square$$

Overall Conclusion The expected number of dictionary operations of the Randomized Closest Pair algorithm is $O(n)$

Finally In order to be able to conclude that the algorithm has an expected running time of $O(n)$, we need to specify a data structure for the dictionary where every Lookup and Insert operation has an expected running time of $O(1)$.

One such data structure is a so-called universal hashing-scheme (see section 13.6 of the book) which we will not introduce here.

Overall, we obtain the following:

Conclusion The expected running time of the Randomized Closest Pair algorithm is $O(n)$