2 Introductory examples

In this section we will use the techniques from the introduction to investigate various simple examples: a randomized identity test, randomized text searching, randomized Quicksort, randomized search trees, a randomized algorithm to compute the smallest enclosing circle, and a randomized distributed algorithm for synchronization. More advanced data structures and algorithms will be considered in the following sections.

2.1 A randomized identity test

We are given three $n \times n$-matrices A, B and C, and the task is to determine whether $A \cdot B = C$. Suppose that the entries of the matrices are from some finite field $\mathbb{F}$ with neutral elements 0 and 1 w.r.t. addition and multiplication.

The task can certainly be solved by computing $A \cdot B$ and comparing the result with C. For that one would need $\Theta(n^{2.376\cdots})$ time when using the best known algorithm for matrix multiplication [2]. We will consider instead a probabilistic method by Freivalds [4] that can deliver the right answer in time $O(n^2)$ with probability at least $1/2$ (see Figure 1). Instead of checking whether $A \cdot B = C$, Freivalds' algorithm just checks whether $A(B \cdot r) = C \cdot r$ for a randomly chosen vector r. The values $y = A(B \cdot r)$ and $z = C \cdot r$ will be used as so-called fingerprints of $A \cdot B$ and C.

```
r := ⟨vector of n independent random bits⟩
x := B · r
y := A · x
z := C · r
if y ≠ z then
        return NO
else
        return YES
```

Figure 1: Freivalds' algorithm.

The runtime of Freivalds' algorithm is certainly $O(n^2)$. Hence, it remains to prove its correctness.

Lemma 2.1 *Let A, B and C be three $n \times n$-matrices with $A \cdot B \neq C$ and r be a vector of n bits that are chosen independently and uniformly at random. Then* $\Pr[A(B \cdot r) = C \cdot r] \leq 1/2$.

Proof. For $D = A \cdot B - C$ it holds that $D \neq 0$. Let $y = A(B \cdot r)$ and $z = C \cdot r$. Then $y = z$ if and only if $D \cdot r = 0$.

Let d be the first row of D. W.l.o.g., d is non-zero and the non-zero entries of d are the first k entries $d_1, \ldots, d_k$. (The only purpose of this assumption is to simplify the proof. One can easily check that the proof can also be done without that assumption.) The first entry of the product $D \cdot r$ is equal to $d \cdot r$. Certainly, $d \cdot r \neq 0$ implies that also $D \cdot r \neq 0$. Hence, $\Pr[d \cdot r \neq 0] \leq \Pr[D \cdot r \neq 0]$.

We are looking for an upper bound of the probability that $d \cdot r = 0$. This is the case if and only if $\sum_{i=1}^{k} d_i r_i = 0$, which is true if and only if $r_k = (-\sum_{i=1}^{k-1} d_i r_i)/d_k$. For an arbitrary choice of the elements $r_1, \ldots, r_{k-1}$ there is at most one value for r_k so that the inequality is true, which implies that $\Pr[d \cdot r = 0] \leq 1/2$. Hence, $\Pr[d \cdot r \neq 0] \geq 1/2$ and therefore $\Pr[D \cdot r \neq 0] \geq 1/2$. □

Corollary 2.2 *If $A \cdot B = C$, then Freivalds' algorithm always gives the right answer. Otherwise, Freivalds' algorithm gives the right answer with probability at least* $1/2$.

An error probability of $1/2$ is still quite large. Can this be decreased? One way of doing that is not to choose r as a random bit vector but a random vector from $\mathbb{F}^n$. Then it follows from the condition that $r_k = (-\sum_{i=1}^{k-1} d_i r_i)\ /d_k$ in the proof of Lemma 2.1 that $\Pr[d \cdot r = 0] \leq 1/|\mathbb{F}|$ and therefore $\Pr[A(B \cdot r) = C \cdot r] \leq 1/|\mathbb{F}|$. Hence, the error probability decreases to $1/|\mathbb{F}|$.

Another way of reducing the error probability is to repeat Freivalds' algorithm k times independently at random (instead of just once). If at least one of the executions outputs NO, we output NO as well, and otherwise we output YES.

For each $i \in \{1, \ldots, k\}$ let the event E_i be true if and only if the i-th execution of Freivalds' algorithm outputs "YES". Suppose that $A \cdot B \neq C$. Since Freivalds' algorithm is executed *independently* at random,

$$\Pr[E_1 \cap E_2 \cap \ldots \cap E_k] = \prod_{i=1}^{k} \Pr[E_i] \leq (1/2)^k$$

i.e., the probability that the final output for $A \cdot B \neq C$ is YES is at most $(1/2)^k$. If k is set to $\log n$, then the error probability is at most $1/n$.

2.2 Text search

Next we consider two problems about comparing text strings. To simplify notation, we will focus on bit sequences. The reader may verify that the algorithms and their analysis can easily be extended to alphabets of larger size.

First of all, we consider the problem of determining whether two bit sequences are equal, i.e., given two bit sequences $(a_1 \ldots a_n)$ and $(b_1 \ldots b_n)$, the problem is to determine whether $(a_1 \ldots a_n) = (b_1 \ldots b_n)$. If one interprets the bit sequences as binary encodings of numbers, then $a = \sum_{i=1}^{n} a_i 2^{i-1}$ and $b = \sum_{i=1}^{n} b_i 2^{i-1}$, and the bit sequences are equal if and only if $a = b$ for these encodings (under the assumption that the bit sequences have the same length).

Now, imagine that a and b are stored in two different locations A and B, and the task is to determine their inequality by exchanging as few bits as possible. Certainly, one can solve this problem deterministically by transmitting n bits. In the following, we present a randomized algorithm that just needs to transmit $O(\log n)$ bits and that solves the problem with an error probability of at most $1/n$ (see Figure 2).

$p :=$prime number that is randomly chosen from $[1, 2n^2 \ln(n^2)]$
$a := \sum_{i=1}^{n} a_i 2^{i-1}$
$b := \sum_{i=1}^{n} b_i 2^{i-1}$
if $a \bmod p = b \bmod p$ **then**
    return YES
else
    return NO

Figure 2: Algorithm for comparison of two bit sequences a and b.

We just need to transmit $O(\log n)$ bits because it is sufficient to send either $a \bmod p$ or $b \bmod p$, and sending a number modulo p just needs $\log(2n^2 \ln(n^2)) = O(\log n)$ bits.

For a prime number p let $F_p(x) : \mathbb{Z} \to \mathbb{Z}_p$ be the mapping $x \to x \bmod p$. The algorithm in Figure 2 checks whether $F_p(a) = F_p(b)$, and outputs a wrong answer if this is the case even though $a \neq b$. This is the case if and only if p is a divisor of $c = |a - b|$, which is the starting point for the following theorem.

Theorem 2.3 *Let $n \geq n_0$ for some sufficiently large constant n_0. When choosing a random prime number $p \leq 2n^2 \ln(n^2)$ then*

$$\Pr[F_p(a) = F_p(b) \mid a \neq b] \leq \frac{1}{n}$$

Proof. We need two lemmas to show the theorem. The first one is difficult to show, so we skip its proof.

Lemma 2.4 (Chebychev) *Let $n \geq n_0$ for some sufficiently large constant n_0. Then the number $\pi(n)$ of prime numbers in $\{1, \ldots, n\}$ satisfies*

$$\frac{7}{8} \cdot \frac{n}{\ln n} \leq \pi(n) \leq \frac{9}{8} \cdot \frac{n}{\ln n}$$

Lemma 2.5 *The number of prime divisors of a number at most 2^n is at most n.*

Proof. Every prime divisor is at least 2. Therefore, the product of $n + 1$ prime divisors is more than 2^n. □

Let $c = |a - b|$. The algorithm in Figure 2 gives a wrong answer if and only if $c \neq 0$ and c is divided by p. Since $c \leq 2^n$, c has at most n prime divisors according to Lemma 2.5. Moreover, according to Lemma 2.4, $[1, 2n^2 \ln(n^2)]$ contains at

least n^2 prime numbers, as long as $n \geq n_0$. Hence, the probability to choose a p so that p divides c is at most $n/n^2 = 1/n$. □

Now we look at the second problem. In the so-called *pattern matching problem* there is a text $x = x_1 \dots x_n$ and a shorter search string $y = y_1 \dots y_m$, and the task is to determine if there is a $j \in \{1, \dots, n-m+1\}$ so that $y_i = x_{j+i-1}$ for all $i \in \{1, \dots, m\}$. This problem can be solved deterministically in time $O(n+m)$ using the Knuth-Morris-Pratt [6] or Boyer-Moore [1] algorithm, but both algorithms are not easy to understand and to analyze. We will present a randomized algorithm that runs in the same time but is much easier to understand.

Let $x(j)$ be the substring $x_j \dots x_{j+m-1}$ of length m in x. The basic idea behind the Karp-Rabin algorithm is to avoid a direct comparison of y and $x(j)$ and instead just compare finger prints $F_p(y)$ and $F_p(x(j))$ of y and $x(j)$. An important observation for this is that

$$\begin{aligned}
x(j+1) &= \sum_{i=1}^{m} x_{(j+1)+i-1} 2^{i-1} \\
&= \frac{1}{2}(x_j - x_j) + \frac{1}{2} \sum_{i=1}^{m-1} x_{(j+1)+i-1} 2^i + x_{(j+1)+m-1} 2^{m-1} \\
&= \frac{1}{2} \left(\sum_{i=0}^{m-1} x_{j+i} 2^i - x_j \right) + x_{j+m} 2^{m-1} \\
&= \frac{1}{2} \left(\sum_{i=1}^{m} x_{j+i-1} 2^{i-1} - x_j \right) + x_{j+m} 2^{m-1} \\
&= \frac{1}{2}(x(j) - x_j) + x_{j+m} 2^{m-1}
\end{aligned}$$

Hence, one can compute $x(j+1)$ from $x(j)$, and therefore also $F_p(x(j+1))$ from $F_p(x(j))$ by just executing a constant number of operations. Therefore, the algorithm in Figure 3 needs at most $O(m+n)$ time.

```
p :=prime number that is randomly chosen from [1, 2n^2 m ln(n^2 m)]
y := Σ_{i=1}^{m} y_i 2^{i-1} mod p
z := Σ_{i=1}^{m} x_i 2^{i-1} mod p
for j := 1 to n − m do
    if y = z then
        return j    // first position with a match
    z := (z − x_j)/2 + x_{j+m} 2^{m-1} mod p
return -1    // y does not occur in x
```

Figure 3: Karp-Rabin algorithm

Theorem 2.6 *The Karp-Rabin algorithm outputs a wrong answer with probability at most* $1/n$.

Proof. Let $c = |y - x(j)|$. Similar to the algorithm in Figure 2, the Karp-Rabin algorithm delivers a wrong answer only if $y \neq x(j)$ but p is a divisor of c. c is at most 2^m and therefore has at most m prime divisors. On the other hand, for $n^2 m \geq n_0$ there are at least $n^2 m$ many prime divisors in $[1, 2n^2 m \ln(n^2 m)]$. Thus, the probability that p is a divisor of c is at most $m/(n^2 m) = 1/n^2$. Therefore, the probability that there is a j so that p is a divisor of $|y - x(j)|$ is at most $n/n^2 = 1/n$. □

2.3 Quicksort

The following algorithm is a simple randomized variant of the well-known Quicksort algorithm. For simplicity, we assume that all numbers are pairwise different.

The runtime of the randomized Quicksort algorithm is certainly within a constant factor of the number of comparisons performed by it. Hence, it suffices to prove the following theorem in order to show an expected runtime of $O(n \log n)$.

RandQSort($s_1 \dots s_n$):
i :=random element out of $\{1, \dots, n\}$
$y := s_i$
$j_1 := 1$; $j_2 := 1$
for $i := 1$ **to** n **do**
    if $s_i < y$ **then** $t_{j_1} := s_i$; $j_1 := j_1 + 1$
    if $s_i > y$ **then** $u_{j_2} := s_i$; $j_2 := j_2 + 1$
return RandQSort(t) $\circ$ y $\circ$ RandQSort(u) // $\circ$: *concatenation*

Figure 4: Randomized Quicksort algorithm

Theorem 2.7 *On expectation, RandQSort makes at most $2n \cdot H_n = O(n \log n)$ comparisons, where H_n is the n-th harmonic number.*

Proof. W.l.o.g. we assume that $S = (s_1, \dots, s_n)$ is a permutation of $\{1, \dots, n\}$. We define the binary random variables

$$X_{i,j} = \begin{cases} 1 & \text{if } i \text{ and } j \text{ are compared in the algorithm} \\ 0 & \text{otherwise} \end{cases}$$

Altogether, the number of comparisons is equal to $\sum_{i<j} X_{i,j}$. The reason for this is that no comparison is done twice because one of the two elements must be the pivot element y. Due to the linearity of expectation,

$$\mathbb{E}\left[\sum_{i<j} X_{i,j}\right] = \sum_{i<j} \mathbb{E}[X_{i,j}] = \sum_{i<j} p_{i,j}$$

where $p_{i,j} = \Pr[X_{i,j} = 1]$.

Consider a fixed pair (i, j) with $i < j$. i and j can only be compared with each other if i or j is the pivot element and none of the elements in $\{i, \dots, j\}$ has been chosen as a pivot element before. Hence, we can state the following two facts:

- i and j are compared with each other if and only if i or j is selected as the first pivot element in $\{i, \dots, j\}$.
- The probability that an element $k \in \{i, \dots, j\}$ is chosen as the first pivot element in $\{i, \dots, j\}$ is the same for all elements in $\{i, \dots, j\}$.

Hence, $p_{i,j} = 2/(j - i + 1)$. Therefore,

$$\mathbb{E}\left[\sum_{i<j} X_{i,j}\right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{2}{k} = \sum_{i=2}^{n} \sum_{k=2}^{i} \frac{2}{k} = 2 \sum_{i=1}^{n} (H_i - 1)$$

where $H_i = \sum_{k=1}^{i} 1/k$ is the i-th harmonic number. From $\ln i \leq H_i \leq \ln i + 1$ the theorem follows. □

2.4 Successor Search

Next we introduce randomized search trees. For simplicity we will only consider binary trees. A binary tree $T = (V, E)$ with a key mapping $f : V \to U$ is a *search tree*, if for each node u with a left subtree T_1 and a right subtree T_2 $\max_{v \in T_1} f(v) < f(u) \leq \min_{w \in T_2} f(w)$.

Randomly built search tree

At first we will evaluate the performance of search trees that are built by insertions of elements of a set $S \subseteq U$ in a random order. Let s_i be the ith smallest element in $S = \{s_1, \dots, s_n\}$ and let π be a permutation of $\{1, \dots, n\}$ chosen uniformly at random. Based on π, we build a simple binary search tree T (i.e., each node has up to two children). For an illustration see Fig. 5. An example is given in Fig. 6.

$\pi :=$ a permutation of $\{1, \ldots, n\}$ chosen uniformly at random
$T :=$ empty search tree
for $i := 1$ **to** n **do**
    Insert($s_{\pi(i)}, T$) *// inserts $s_{\pi(i)}$ as a leaf in T*

Figure 5: Randomly built search tree

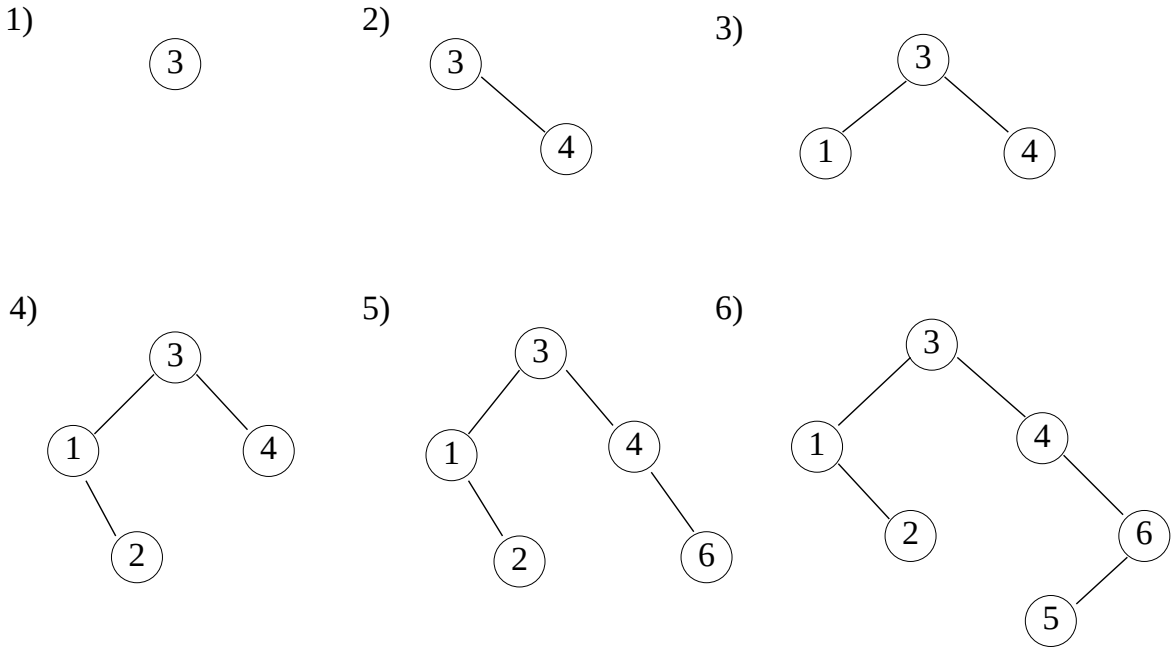

Figure 6: Insertion of 1,2,3,4,5,6 in the order 3,4,1,2,6,5.

We want to determine the time of a Lookup operation for a fixed element s_i. Let D_i be the random variable for the length of the path from s_i to the root of T, i.e., the number of ancestors of s_i in T. We further define the indicator variable $X_{i,j}$ with $X_{i,j} = 1$ iff (if and only if) s_j is an ancestor of s_i in T. It holds

$$D_i = \sum_{j=1}^{n} X_{i,j}$$

and

$$\begin{aligned} \mathbb{E}[D_i] &= \mathbb{E}[\sum_{j=1}^{n} X_{i,j}] = \sum_{j=1}^{n} \mathbb{E}[X_{i,j}] = \sum_{j=1}^{n} \Pr[X_{i,j} = 1] \\ &= \sum_{j=1}^{n} \Pr[s_j \text{ is ancestor of } s_i] \end{aligned}$$

To compute $\mathbb{E}[D_i]$ we introduce the following definition.

Definition 2.8 *Let $S = \{s_1, \ldots, s_n\}$ be a set of elements with pairwise distinct priorities $prio(s_k)$. A* treap *(Tree + Heap) for S is a binary tree T, such that:*

- *T is a search tree for S.*
- *T is a heap concerning the priorities of S, i.e. the priority for each node in T is smaller than the priorities of its children.*

The next lemma characterizes the relation between priorities and the tree structure of a treap.

Lemma 2.9 *Let $S = \{s_1, \ldots, s_n\}$ be a set with pairwise distinct priorities $prio(s_k)$. Then s_j is an ancestor of s_i in the treap T for S iff*

$$prio(s_j) = \min prio(S_{i,j})$$

where $\min prio(S_{i,j})$ denotes the smallest priority in $S_{i,j} = \{s_i, \ldots, s_j\}$

Proof. The proof is left as an exercise. □

Now we will consider treaps whose priorities are given by a random permutation.

Lemma 2.10 *Let π be chosen uniformly at random from the set of permutations of $\{1, \ldots, n\}$ and let $prio(s_k) = \pi(k)$. Then,*

$$\Pr[s_j \text{ is ancestor of } s_i] = \frac{1}{|j-i|+1}$$

Proof. The probability that for $k \in \{i, \ldots, j\}$ the priority $\pi(k)$ is the smallest priority of all elements in $S_{i,j}$ is the same for all k, due to symmetry, and thus,

$$\frac{1}{|S_{i,j}|} = \frac{1}{|j-i|+1}$$

□

Then it holds

$$\begin{aligned}
\mathbb{E}[D_i] &= \sum_{j=1}^{n} \Pr[s_j \text{ is ancestor of } s_i] \\
&= \frac{1}{i} + \frac{1}{i-1} + \ldots + \frac{1}{2} + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n-i+1} \\
&= H_i + H_{n+1-i} - 2 \\
&\leq 2 \ln n
\end{aligned}$$

This implies the following theorem:

Theorem 2.11 *If a set S of n elements is inserted in an random order in an initially empty binary search tree, then the expected depth of a fixed node s_i is at most $2 \ln n$.*

Randomized Search trees

So far we have seen that each element in a search tree has an expected depth of $O(\log n)$ if the elements are inserted in a random order. Now we want to develop efficient search trees with an expected depth of $O(\log n)$ for any order of insertions. We call these trees *randomized search trees*. We will construct the search tree so that the elements are in the same positions as if inserted in a random order, so that we can reuse the results of randomly built search trees. We define these search trees in the following way:

Definition 2.12 *A* randomized search tree *for S is a treap for S in which each element $s \in S$ has a priority $prio(s)$ that is chosen uniformly at random out of $[0, 1]$.*

Lemma 2.13 *Let $S = \{s_1, \ldots, s_n\}$ be a set of elements and let $prio(s_1), \ldots, prio(s_n)$ be chosen uniformly at random from $[0, 1]$. Then*

$$\Pr[prio(s_j) = \min prio(S_{i,j})] = \frac{1}{|j-i|+1}$$

Proof. We assume that there are no identical priorities, that is, the probability that elements have the same priority is 0 so that it does not have to be considered. Because each priority is chosen uniformly at random from the same set, the elements are symmetric and the probability for each element to have the smallest priority is

$$\Pr[prio(s_j) = \min prio(S_{i,j})] = \frac{1}{|j-i|+1}$$

□

This implies the following theorem:

Theorem 2.14 *Let S a set of n elements that are stored in a randomized search tree. Then the expected depth of a node s_i is*

$$H_i + H_{n+1-i} - 2 \leq 2 \ln n$$

It remains to show how to efficiently insert elements in a randomized search tree. The insertion is analogous to the insertion in a normal search tree. Additionally, we remember the search path and recover the heap invariant w.r.t. the priorities by rotations along the search path. (see Fig. 7).

```
TreapInsert(s, prio(s), v):
if v = NULL then v := new node(s, prio(s))
else
    if s < v then
        TreapInsert(s, prio(s), leftchild(v))
        if prio(leftchild(v)) < prio(v) then RotateRight(v)
    else
        if s > v then
            TreapInsert(s, prio(s), rightchild(v))
            if prio(rightchild(v)) < prio(v) then RotateLeft(v)
        // else s is already in the tree
```

Figure 7: The Insert operation

Next we show how to remove elements from a randomized search tree. Here, we simply invert the Insert operation. The element that needs to be removed is rotated downwards in the tree until it is a leaf. Then it can easily be removed (see Fig. 8).

```
TreapDelete(s):
v := node for s in the treap
RootDel(v)

RootDel(v):
if v is a leaf then v = NULL
else
    if prio(leftchild(v)) < prio(rightchild(v)) then
        RotateRight(v)
        RootDel(rightchild(v))
    else
        RotateLeft(v)
        RootDel(leftchild(v))
```

Figure 8: The Delete operation.

The time it takes to remove an element is proportional to the length of the path from the root to a leaf position. Thus,

Theorem 2.15 *In a randomized search tree with n elements the* TreapInsert *and* TreapDelete *take $O(\log n)$ time in expectation.*

2.5 Smallest enclosing circle

Next, we consider the following problem. Given a set P of n points $p_1, \ldots, p_n$ in a 2-dimensional Euclidean space, the task is to find the smallest enclosing circle of these points. In other words, we are looking for a point $s \in \mathbb{R}^2$ so that $r = \max_i ||s - p_i||$ is minimal, where $||s - p_i||$ is the Euclidean distance between s and p_i.

It is easy to see that the smallest enclosing circle always exists and that it is unique. Moreover, the radius r of the smallest enclosing circle satisfies one of the two cases:

- There are two points $p_i, p_j \in P$ with $||p_i - p_j|| = 2r$, or
- there are three points $p_i, p_j, p_k \in P$ that lie on the circle.

With that insight, one can design a simple deterministic algorithm that computes the smallest enclosing circle: For all triples (p_i, p_j, p_k) determine the smallest circle that contains p_i, p_j and p_k, and output the one with maximal radius among all of these circles. Unfortunately, this algorithm has a runtime of $O(n^3)$. An alternative is the randomized algorithm in Figure 9, also known as Clarkson's algorithm.

```
P := {p_1, ..., p_n}
while true
    S :=random subset of 13 points from P
    K :=smallest enclosing circle for S
    if K contains all points in P then return K
    Q :=multiset of all points in P that are not contained in K
    P := P ∪ Q    // ∪ extends multiset P by Q
```

Figure 9: Clarkson's algorithm for the smallest enclosing circle.

We will present a formal analysis of the algorithm later in this lecture. Here, we just want to give an intuition why the algorithm works and why it is efficient. Let $K(P)$ be the smallest enclosing circle of the point set P. As we have already mentioned, there is a set B of at most 3 points in P with $K(B) = K(P)$. Therefore, if for some subset R of P it holds that $K(R) \neq K(P)$, then at least one of the points in B must be outside of $K(R)$. This means that in Clarkson's algorithm at least one of the points in B doubles its number of copies in P. Hence, after k rounds at least one of these points has at least $2^{k/3} \approx 1.26^k$ copies in P. So the number of these copies grows exponentially with k. On the other hand, one can show that on expectation, not too many copies are added to P. Concretely, this number is at most $3z/13$, where z is the number of elements in P. This means that, on expectation, we will have at most $(1 + 3/13)z \approx 1.23z$ many points in P. (The 3 comes from the size of B and the 13 from the size of the sample that we pick.) Thus, on the one hand, the number of points in P grows by a factor of about 1.23, which means that there are about $n \cdot 1.23^k$ points in P after k rounds. On the other hand, after k rounds there is a point that has at least 1.26^k copies in P. No matter how large n, due to $1.26 > 1.23$ eventually there are more copies than there are points in P, a contradiction. Hence, the only way to resolve this paradox is that the algorithm terminates before getting to this point.

Therefore, on expectation, Clarkson's algorithm needs at most $O(\log n)$ rounds to terminate. With a clever implementation (store the copies of points by using counters), a single round just takes $O(n)$ time so that the overall runtime is $O(n \log n)$. This is much better than the runtime $O(n^3)$ obtained by the deterministic algorithm. One can even get a randomized (but more complex) algorithm with runtime $O(n)$. The details for that will be given later in this lecture.

2.6 Distributed synchronization

At the end of this chapter we will consider a problem in the area of distributed algorithms. Suppose that we have n continuously running processes, $P_1, \ldots, P_n$, that become active every t time steps. The problem is to synchronize the times at which these processes become active. There is no global time, but we assume that every process runs at the same speed and that it can execute the following operations:

- diff(P_j): outputs the time difference between the starting points of P_i and P_j (i.e., the time from the starting point of P_i till the next starting point of P_j).
- adjust(t): delays the starting point of P_i by t time units.

One possibility to solve the synchronization problem is to select a process that determines the activation times of all other processes. However, this method does not scale well because the chosen process could become a bottleneck, and if it fails, the whole procedure might have to start from scratch. An alternative strategy is shown in Figure 10.

```
choose two random processes, P_i and P_j
t_1 :=diff(P_i); t_2 :=diff(P_j)
if min(t_1, t − t_1) < min(t_2, t − t_2) then
        adjust(t_1)
else
        adjust(t_2)
```

Figure 10: Synchronization algorithm that is continuously executed by each process.

Surprisingly, this algorithm just needs $O(\log n)$ communication rounds most of the time until all processes are synchronized. This can be validated through experiments. Unfortunately, no formal analysis of this runtime bound exists so far. For some simpler cases upper and lower bounds are already known [3, 5], but we will not cover them in this lecture.

References

[1] R. Boyer and J. Moore. A fast string search algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[2] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9(3):251–280, 1990.

[3] B. Doerr, L. A. Goldberg, L. Minder, T. Sauerwald, and C. Scheideler. Stabilizing consensus with the power of two choices. In *Proc. of the 23rd ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 149–158, 2011.

[4] R. Freivalds. Probabilistic machines can use less running time. In *Information Processing 77, Proceedings of the IFIP Congress 1977*, pages 839–842, 1977.

[5] B. Haeupler, G. Pandurangan, D. Peleg, R. Rajaraman, and Z. Sun. Discovery through gossip. In *Proc. of the 24th ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 140–149, 2012.

[6] D. Knuth, J. Morris, and V. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.