Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institute of Theoretical Computer Science
Bernd Gärtner, Rasmus Kyng, Angelika Steger, David Steurer

Algorithms, Probability, and Computing | Special Assignment 1 | HS24

- Write an exposition of your solution using a computer, where we strongly recommend to use LaTeX. We do not grade hand-written solutions.
- You need to submit your solution via Moodle until October 29th by 2 pm. Late solutions will not be graded.
- For geometric drawings that can easily be integrated into LaTeX documents, we recommend the drawing editor IPE, retrievable at `http://ipe.otfried.org` in source code and as an executable for Windows.
- Write short, simple, and precise sentences.
- This is a theory course, which means: if an exercise does not explicitly say "you do not need to prove your answer" or "justify intuitively", then a formal proof is always required. You can of course refer in your solutions to the lecture notes and to the exercises, if a result you need has already been proved there.
- We would like to stress that the ETH Disciplinary Code applies to this special assignment as it constitutes part of your final grade. The only exception we make to the Code is that we encourage you to verbally discuss the tasks with your colleagues. However, you need to write down the names of all your collaborators at the beginning of the writeup. It is strictly prohibited to share any (hand)written or electronic (partial) solutions with any of your colleagues. We are obligated to inform the Rector of any violations of the Code.
- There will be two special assignments this semester. Both of them will be graded and the average grade will contribute 20% to your final grade.
- As with all exercises, the material of the special assignments is relevant for the (midterm and final) exams.

Exercise 1

10 **points**

(*Random Binary Search Trees*)

Let $n \in \mathbb{N}$, the keys $2, 4, 8, \ldots, 2^n$ are inserted in a uniformly random order in a binary search tree. Let $k_1, k_2, \ldots, k_\ell$ denote the value of the keys on the right spine of the tree, i.e. on the path from the root to the biggest key. In this exercise, we want to compute $X_n = \mathrm{E}[k_1 + k_2, \cdots + k_\ell]$. Note that ℓ is a random number, k_1 is the value in the root node, and $k_\ell = 2^n$.

(a) Write X_n as a function of n and X_i for $i = 1, 2, \ldots, n-1$;

(b) Write X_n as a function of only n and X_{n-1};

(c) Write X_n in a closed form. The result can contain sums and products but should not be written as a function of some X_i for $i \in \mathbb{N}$.

Answer to Exercise 1

(a) Working out the two cases for $n = 1$ yields $X_1 = 2$. As usual, we condition on the rank of the root node. Here, the key observation is that if the root has rank i, the right subtree of the root is a random tree with the keys $2, 4, \ldots, 2^{n-i}$ where all the keys are multiplied by 2^i. Further notice that if the root has rank n, it is the only node on the right spine of the tree and we get $k_1 = 2^n$ for our value. Hence, for $n \geq 2$, we can write the following recurrence.

$$\begin{aligned} X_n &= \frac{2^n}{n} + \frac{1}{n}\sum_{i=1}^{n-1}(2^i + 2^i X_{n-i}) \\ &= \frac{1}{n}\sum_{i=1}^{n} 2^i + \frac{1}{n}\sum_{i=1}^{n-1} 2^{n-i}X_i \\ &= \frac{2^{n+1}-2}{n} + \frac{1}{n}\sum_{i=1}^{n-1} 2^{n-i}X_i \end{aligned}$$

(b) For $n \geq 3$, we get that

$$nX_n - 2(n-1)X_{n-1} = 2 + 2X_{n-1}$$

Or equivalently,

$$X_n = \begin{cases} 2X_{n-1} + \frac{2}{n} & \text{if } n \geq 3 \\ 5 & \text{if } n = 2 \\ 2 & \text{if } n = 1 \end{cases}$$

(c) The solution is

$$X_n = \sum_{i=1}^{n} \frac{2^{n-i+1}}{i},$$

for all $n \geq 1$. Which can be shown by simple induction over n. The base case X_1 is easily checked. Let $m \geq 2$ be arbitrary but fixed, we get:

$$X_m = 2X_{m-1} + \frac{2}{m} = 2\sum_{i=1}^{m-1} \frac{2^{(m-1)-i+1}}{i} + \frac{2}{m} = \sum_{i=1}^{m-1} \frac{2^{m-i+1}}{i} + \frac{2}{m} = \sum_{i=1}^{m} \frac{2^{m-i+1}}{i},$$

where we used the induction hypothesis in the second step.

Exercise 2 <span></span>13 points

(*Number of spanning trees in the complete graph*)

The objective of the exercise is to find the number of distinct spanning trees in a complete graph with n vertices labelled with n distinct labels. An edge between vertices i and j is labelled $\{i, j\}$, and two spanning trees are considered different if they contain any differently labelled edges. For example, on the graph with $n = 3$ labelled vertices, we have 3 distinct spanning trees.
Let $K_n = (V, E)$ be the complete graph with n vertices and suppose that the vertices are labelled $1, 2, \ldots, n$. Let $R \subseteq V : |R| = k$ be a set of k vertices that are fixed (for simplicity, you can think of $R = \{1, 2, \ldots, k\}$). Denote with $T_{n,k}$ the number of (labelled) forests on $\{1, \ldots, n\}$ consisting of k trees whose roots are the vertices in R. Note that $T_{n,k}$ does not depend on R but only on its size and by fixing the roots, the number of possible forests decreases, eg. $T_{3,2} = 2$ as the vertices 1 and 2 cannot be in the same tree (using $R = \{1, 2\}$). For coherence, define also $T_{0,0} = 1$ and $T_{n,0} = 0$ for $n > 0$.

(a) Compute $T_{n,n}$ for $n \geq 1$.

(b) Show that for all $1 \leq k \leq n$,

$$T_{n,k} = \sum_{i=0}^{n-k} \binom{n-k}{i} T_{n-1,k-1+i}.$$

Hint: It might be helpful to use different sets R in recursive cases.

(c) Using the recursive expression above, prove that

$$T_{n,k} = kn^{n-k-1}.$$

(d) Use the previous result to deduce the number of different spanning trees in a complete graph with n labelled vertices.

Answer to Exercise 2

(a) If we have a forest with n vertices and n trees, the only possibility is that each tree consists of only one vertex. This proves that $T_{n,n} = 1$ for all $n \geq 1$.

(b) Suppose that we have a forest that consists of k trees. Let v be the root of one of the trees and suppose we remove v. v can be directly connected to i vertices for $i = 0, 1, \ldots, n-k$ (at least $k-1$ are not connected to v because they are the roots of the other trees). If v is directly connected to i vertices, after v has been removed, the tree with root v splits into i trees and there are $\binom{n-k}{i}$ possible combinations to connect i of the $n-k$ non-root nodes to v. Furthermore, we have obtained a forest with $(n-1)$ vertices and $(k-1+i)$ connected components. Note that we are not double-counting or not-counting any of the combinations here. *Showing this can be made explicit, but we do not expect the argument below to be stated in this much detail to get a full mark.*

Below follows an explicit exposition of this not double-counting and not-counting. Towards this goal, let's define the following sets of graphs

$$\mathcal{T}_{n,k} := \Big\{ \Big([n], E(F) \cup \{\{r,n\} : r \in R^*\} \Big) : T \in \mathcal{T}_{n-1,k-1+i}, R^* \in \binom{[n-k]}{i}, 0 \leq i \leq n-k \Big\}$$

$$\mathcal{F}_{n,k} := \Big\{ \text{ all rooted labelled forests on } n \text{ nodes and } k \text{ roots with } R = \{n-k+1, \ldots, n\} \Big\}$$

In the following, we compare $\mathcal{T}_{n,k}$ (the forests that we construct, i.e. $T_{n,k} = |\mathcal{T}_{n,k}|$) with $\mathcal{F}_{n,k}$ (the actual forests that we should obtain) and want to show that they are equal. We use $[n] = \{1, 2, \ldots, n\}$ and $\binom{[n]}{i}$:= are all i-element subsets of $[n]$ as in A&W.

1. No double-counting: Consider $T_1 \neq T_2 \in \mathcal{T}_{n,k}$. Note that the tree which contains the node n (it is the root of this tree) is different for every element as the edges that go out of n are different, i.e. the R^* (the i direct neighbours of n) are different.

2. No not-counting, i.e. $F \in \mathcal{F}_{n,k} \implies F \in \mathcal{T}_{n,k}$: Assume towards a contradiction that $\exists F : F \in \mathcal{F}_{n,k} \wedge F \notin \mathcal{T}_{n,k}$. Consider vertex n in F (n is a root as $n \in R$). It has to have $0 \leq \deg(n) \leq n-k$ (n can only be connected to non-root nodes, namely to elements in $[n-k]$). But we consider all of these possibilities in the definition of $\mathcal{T}_{n,k}$, namely all subsets of $[n-k]$ are in $\bigcup_{0 \leq i \leq n-k} \binom{[n-k]}{i}$. Now consider $F \backslash n$. It has to have $k-1+\deg(n)$ many trees. Assuming as our induction hypothesis that the above holds for $n' < n$, we know inductively that $F \backslash n \in \mathcal{F}_{n-1,k-1+\deg(n)} \implies F \backslash n \in \mathcal{T}_{n-1,k-1+\deg(n)}$.

3. All are valid, i.e. $F \in \mathcal{T}_{n,k} \implies F \in \mathcal{F}_{n,k}$: We can argue inductively that all our $F \in \mathcal{T}_{n,k}$ are valid. We can assume that this is the case inductively for $E(T)$, the edges of $T \in \mathcal{T}_{n-1,k-1+i}$. There, our $R' = R \backslash \{n\} \cup R^*$. The only edges that we now add are from root n to root nodes $r \in R^*$; because root nodes are not connected with each other, we don't form any cycles. Furthermore, we add 1 node and $i = |R'|$ edges, thus going from $k-1+i$ to $k-1+i+1-i = k$ connected components. Our $R = \{n-k+1, \ldots, n\}$, so because we connected n only to nodes in $[n-k]$, we indeed have rooted trees with roots in R. Note that in the induction step, we actually use a different R; being precise, we should actually define $\mathcal{T}_{n,k,R}$ and $\mathcal{F}_{n,k,R}$ and prove the properties for all $R : |R| = k$, but this is just a permutation of the indices and thus a trivial generalization of the above.

(c) We prove the result via induction. The Base Case $T_{1,k}$ is trivial with $T_{1,0} = 0$ by definition and $T_{1,1} = 1$ by (a) and the Base Case $T_{n,0} = 0$ by definition also satisfies our hypothesis. The Induction Hypothesis is given as in the exercise as $T_{a,b} = ab^{a-b-1} \; \forall \; b \leq a < n$.
So let's focus on the Induction Step:

$$\begin{aligned}
T_{n,k} &= \sum_{i=0}^{n-k} \binom{n-k}{i} T_{n-1,k-1+i} \\
&\overset{I.H.}{=} \sum_{i=0}^{n-k} \binom{n-k}{i} (k-1+i)(n-1)^{n-1-k-i} \\
&= \sum_{i'=0}^{n-k} \binom{n-k}{i'} (n-1-i')(n-1)^{i'-1} \\
&= \sum_{i'=0}^{n-k} \binom{n-k}{i'} (n-1)^{i'} - \sum_{i'=1}^{n-k} \binom{n-k}{i'} i'(n-1)^{i'-1} \\
&= (1+(n-1))^{n-k} - (n-k) \sum_{i'=1}^{n-k} \binom{n-k-1}{i'-1} (n-1)^{i'-1} \\
&= n^{n-k} - (n-k) \sum_{i''=0}^{n-k-1} \binom{n-k-1}{i''} (n-1)^{i''} \\
&= n^{n-k} - (n-k)(1+(n-1))^{n-1-k} \\
&= n^{n-k} - (n-k)n^{n-1-k} \\
&= kn^{n-1-k}
\end{aligned}$$

Where we used the substitutions $i' = n-k-i$, $i'' = i'-1$ and the fact that $\sum_{i=0}^{q} \binom{q}{i}(n-1)^i = n^q$ is a special case of the binomial theorem $(a+b)^q$ where $a = 1$ and $b = n-1$.

(d) Substitute $k = 1$ in the formula above and obtain $T_n = n^{n-2}$. This is known as Cayley's formula.

Exercise 3 12 points

(*Point Location*)
Given a collection S of n points in the plane and a constant $d > 0$, consider the problem of, given a query point q, finding all the points in S that are at most at distance d from q. Devise a data structure for this problem. In order to get the maximum score, the preprocessing time should be polynomial in the number of points n and if k is the number of points to report, the query should take $O(\log(n) + k)$ operations in expectation.

Answer to Exercise 3

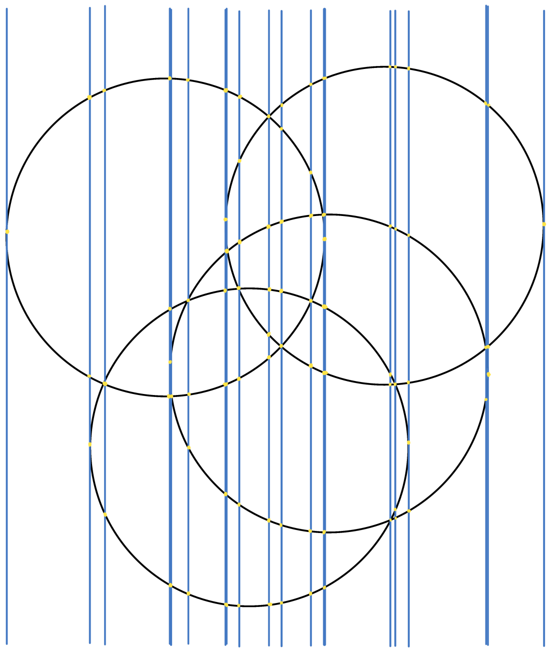

Figure 1: Vertical lines are added at the extreme points of circles and at their intersections with each other. The vertices of the final subdivision are in yellow.

Idea: Draw n circles of radius d, centred at the input points. The arcs of these circles and the intersection points define a planar graph that tiles the plane. Furthermore, for each tile, all points in the same tile share the same points at a distance of at most d.

At this point, we want to use a strategy somewhat similar to trapezoidal decomposition, where, upon receiving a query point q, we find the tile that contains q and report the precomputed answer. To make the subdivision more amenable to strategies like binary search, we'd like to add some structure. It is always possible to add vertical lines to divide the plane further, such that we can determine in $O(\log n)$ time which "vertical strip" contains our query point. Once we have found the correct strip, we still need to locate our point within one of the tiles in that strip. Therefore, we would like these strips to be, in some sense, "nice".

If we trace vertical lines such that they pass through each leftmost and rightmost point of a circle (i.e., its vertical tangents) and through each intersection point of a circle with another circle, we obtain a decomposition such that each strip (the region between two successive vertical lines) contains different arcs that have one endpoint on the left side of the strip and one endpoint on the right side. Furthermore, the interiors of the arcs do not intersect. In particular, for any x-coordinate in an interval associated with a strip, we encounter the arcs in the same order when moving along the y-axis. Thus, for each such interval/strip, we can maintain a sorted array of arcs, through which we can perform binary search to locate our query point (we assume that checking whether a point is below an arc can be done in constant time).
It remains to prove an upper bound on the number of vertices in the graph. Each vertex of the graph is either the intersection of a circle with a line or the point where a tangent line touches a circle. Each vertical line intersects at most $2n$ times with the circles. The number of vertical lines is $2n + 2\binom{n}{2}$ (the tangent ones and the ones corresponding to circle-circle intersections), giving an upper bound of $2n(2n + \binom{n}{2}) = O(n^3)$ on the number of vertices.

Exercise 4 15 points

(*Warm-up: Dynamic List Ranking*)
Suppose you have access to a *static* data structure which does the following: given a collection of n integer-valued elements, it takes $O(n \log n)$ preprocessing time; and after that, given a query element q, it returns the number of elements smaller than q in the initial input in $O(\log n)$ time[1]. The data structure is static in the sense that if we want to add a new element to the sequence, then we have to spend $O(n \log n)$ time and rebuild the data structure.
We want to use this data structure to solve the following *dynamic* problem. You are given a sequence of n elements $e_1, e_2, \ldots, e_n$ one at a time and each time, given a new element e_k, you have to tell what is the rank of this element, i.e. how many elements smaller than e are there in the sequence $e_1, e_2, \ldots, e_k$.

(a) Find an algorithm that solves the problem above using the data structure as a black box. After receiving n elements as input, the total amount of time required by the algorithm should be $O(n \log^2 n)$.

Note that you do not have access to an oracle that compares two elements without using the static data structure.
Hint: Use the data structures of geometrically increasing size.

(*Count points below a line dynamically*)
In this exercise, we devise a *dynamic* version of the data structure for reporting the number of points below a line with nearly the same total update time as the static version. We suppose that we start with an empty data structure. Then, at most n point insertions, at most n point removals, and an arbitrary amount of queries happen in some arbitrary order.

Our data structure should take $O(\log^2 n)$ time to process each query. Furthermore, the total time used for processing insertions and removals should be $O(n^2 \log n)$, and the total space occupied by the data structure should be $O(n^2)$.

(b) Reduce the problem above to the problem of counting the number of lines below a query point when only insertions and queries occur. Formally, you are required to demonstrate that, given a data structure capable of counting the number of lines below a query point—supporting both queries and insertions—it is possible to construct a data structure that addresses the problem described above while supporting insertions, deletions, and queries. Additionally, for any insertion of n points into both data structures, the total construction time of the two data structures must be asymptotically equivalent, as should the query and total insertion/deletion times. For a full score, your algorithm must work correctly even when asked to remove points that were not previously inserted.

(c) Devise an algorithm that solves the problem of counting the number of lines below a query point when only insertions and queries occur. In order to get a full score, your algorithm should take $O(\log^2 n)$ time to process each query, the total time used for processing insertions should be $O(n^2 \log n)$, and the total space occupied by the data structure should be $O(n^2)$.
Hint: Use the strategy from part (a).

[1] This can be achieved by sorting the elements and then performing binary search

Answer to Exercise 4

(a) When we get a new element, we create a data structure that contains only the new element. Afterwards, if there are 2 data structures containing the same number 2^i of elements, we build a new data structure that contains the 2^{i+1} elements and discard the two old data structures. Since the size of the data structures increases geometrically, at any moment, we can have at most $\log n$ data structures. Therefore, when a new element comes, we can query all the data structures and report the sum of the rank of the new element in each data structure; this requires $O(\log^2 n)$ time. For $j = 1, 2, \ldots, \log n$, a data structure of size 2^j is built after every 2^j insertions. So during the insertion of n elements, we build $n/2^j$ times a data structure of size 2^j. The total time needed by the algorithm is

$$\sum_{j=1}^{\log n} \frac{n}{2^j} 2^j j + \sum_{i=1}^{n} \log^2 i = O(n \log^2 n)$$

.

(b) We can reduce to the problem of counting the number of lines above a point using duality, as seen in the lecture note, it is clear that duality holds also in the dynamic setting.
Then, build two identical data structures which don't handle deletions. When you get an insertion, insert the point in the first, when you get a deletion, first check that the point has already been inserted and, if it has, insert the point in the second data structure[2]. When you get a query, query both data structures and report the difference. The time needed for an insertion is $T(\#\text{ insertion})$, the time needed for a deletion is $O(\log(k) + T(\#\text{ deletion}))$, while the time for a query is $T_{\text{query}}(\#\text{ insertion}) + T_{\text{query}}(\#\text{ deletion})$, where $T(\cdot)$ and $T_{\text{query}}(\cdot)$ are the insertion and query time of the two data structures, and k is the number of inserted points.

(c) We use Theorem 3.7 from the lectures as a black-box data structure and we apply the same strategy as in the warm-up.
When we get a new line, we build a data structure with only that line. When we have two data structures with the same size, let's say they contain 2^i lines, we build a new data structure that contains 2^{i+1} lines and discard the two old data structures. When we get a query, we query all the data structures and report the sum of the values we got. The time needed to process a query is bounded by the number of data structures that we have at each moment times the time to query the biggest data structure, which is $O(\log^2 n)$. The total time for the update after the insertions is

$$\sum_{j=1}^{\log n} \frac{n}{2^j} 2^{2j} \leq O(n^2 \log n).$$

And the total space occupied is

$$\sum_{j=1}^{\log n} 2^{2j} = O(n^2).$$

[2] To check existing insertions, we can use an auxiliary data structure that supports efficient insertion and lookup to record the inserted points. Examples include an AVL tree or a hash table.