CSE202 2020-2021 – FINAL EXAM

The 3 exercises are independent and can be treated in arbitrary order. Within an exercise, the answer to a question can be used in the next ones even if a proof has not been found.

The number of points indicated in front of each question is an indication of the relative difficulties of the questions. It is not necessary to solve all the questions to obtain the maximal possible grade.

Exercise 1: Randomized binary search trees

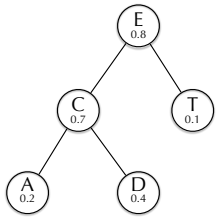
A *treap* is simultaneously a binary search tree and a heap. Each node contains a *key* and a *value*. The *keys* are organized as in a binary search tree with smaller keys in the left subtree and larger ones in the right subtree. As in a heap, the *value* of each node is larger than those of its children. The data structure thus relies on a class Node with 4 fields: `key, value, left, right`. (It is the same as for BST in Lecture 10, with an extra field `value`.)

Question 1. (1 pt) Give a picture of the treap formed from the nodes

$$(A, 0.2), (C, 0.7), (D, 0.4), (E, 0.8), (T, 0.1),$$

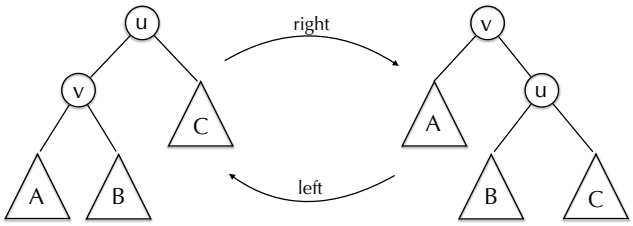
where the letters are the keys that should be sorted alphabetically and the real numbers are the values.

Solution.



□

Question 2. (2 pts) Searching is as in a binary search tree. Insertion and deletion rely on two rotation routines performing the operations indicated in the following picture:



Give the Python code for the right rotation (the left one is similar). Its first line should be

`def rotateright(self,node):`

All subtrees required by the rotation are assumed to exist. The tree is modified and no new node should be created during the rotation. The function returns the new root node `v`.

Solution. Here is the code, using an intermediate variable to avoid losing access to a node

```
def rotateright(self,node):
    tmp = node.left
    node.left = tmp.right
    tmp.right = node
    return tmp
```
□

Question 3. (4pts) Using these rotation operations, insertion proceeds by first inserting a new node at the bottom of the tree as in a binary search tree and then fixing the tree recursively so that it remains a heap. Give the Python code for inserting a node, modeled after the `_insert` routine for BSTs in Lecture 10. Its first line should be

`def _insert(self,node,key,value):`

It returns the root of the treap obtained by adding the node `(key,value)` to the treap whose root is `n`ode.

Solution.

```
def _insert(self,node,key,value):
    if node is None: return Node(key,value)
    if node.key > key:
        node.left = self._insert(node.left,key,value)
        if node.value<node.left.value: node = self.rotateright(node)
    elif node.key < key:
        node.right = self._insert(node.right,key,value)
        if node.value<node.right.value: node = self.rotateleft(node)
    return node
```
□

Question 4. (3 pts) If $\mathrm{depth}(v)$ denotes the number of edges in the path from the node v to the root, while v^+ and v^- denote the predecessor and the successor of the node v for the order on keys, what are the complexities of finding a node v and of inserting a node v?

Solution. Finding a node is as in binary search trees: the complexity is proportional to the depth of the node.

For insertion, the first phase is as in the insertion in binary search trees and stops at depth

$$\delta := \max(\mathrm{depth}(v^+), \mathrm{depth}(v^-)).$$

Next, the tree is fixed by a constant number of assignments performed during the rotations along the branch from the node to the root, thus in number $O(\delta)$. □

We now consider a treap where the values are drawn uniformly and independently in the interval $(0, 1)$. Without loss of generality, we assume that no two values are equal. The aim of the rest of this exercise is to show that the expected depth of a node in such a treap with n nodes is $O(\log n)$, so that all operations (search, insert)

have expected complexity $O(\log n)$. (Delete can also be done similarly and is not treated here.)

Denote by $N_1, \dots, N_n$ the nodes sorted by increasing keys. The node N_i is an *ancestor* of N_j if it is different from N_j and on the path from the root to N_j.

Question 5. (2pts) Show that $\mathbb{E}(\text{depth}(N_j)) = \sum_{i=1}^{n} \Pr(N_i \text{ is an ancestor of } N_j)$.

Solution. Let X_i be the random variable which is 1 if N_i is an ancestor of N_j and 0 otherwise. Then

$$\text{depth}(N_j) = \sum_{i=1}^{n} X_i$$

and

$$\mathbb{E}(\text{depth}(N_j)) = \sum_{i=1}^{n} \mathbb{E}(X_i) = \sum_{i=1}^{n} \Pr(X_i),$$

where the first equality follows from linearity of expectation and the second one from the fact that X_i is 0 or 1. □

Question 6. (2 pts) Show that N_i is an ancestor of N_j if and only if the value of N_i is the largest of the values of $\{N_i, \dots, N_j\}$ if $j > i$ and of $\{N_j, \dots, N_i\}$ otherwise.

Solution. As in a BST, the nodes with keys between N_i and N_j all belong to a subtree having one of these N_k at its root, making it a common ancestor. By definition of a treap, the value of that root N_k is larger than those of its descendants. This is the case in particular when $N_k = N_i$. □

Question 7. (2 pts) Show that

$$\Pr(N_i \text{ is an ancestor of } N_j) = \begin{cases} \frac{1}{j-i+1}, & \text{if } i < j, \\ 0 & \text{if } i = j, \\ \frac{1}{i-j+1} & \text{if } i > j. \end{cases}$$

Solution. The case $i = j$ is by definition. The other cases are symmetric, it is therefore sufficient to consider the first one. Since the values are drawn at random, each of the $j - i + 1$ nodes in $\{N_i, \dots, N_j\}$ has the same probability of having the maximal value. Therefore that probability is $1/(j - i + 1)$. □

Question 8. (2 pts) Conclude that $\mathbb{E}(\text{depth}(N_j)) < 2 \log n$.

Solution. Using the previous questions shows that

$$\begin{aligned} \mathbb{E}(\text{depth}(N_j)) &= \left(\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{j}\right) + \left(\frac{1}{2} + \cdots + \frac{1}{n-j+1}\right) \\ &< 2\left(\frac{1}{2} + \cdots + \frac{1}{n}\right) < 2\int_1^n \frac{dx}{x} = 2\log n. \end{aligned}$$

□

Exercise 2: Linear recurrent sequences

Given a linear recurrence of the form

$$(\mathcal{R}) \qquad u_{n+k} = a_0 u_n + \cdots + a_{k-1} u_{n+k-1},$$

with constant $a_0, \dots, a_{k-1}$ and initial conditions $(u_0, \dots, u_{k-1})$ all in a field $\mathbb{K}$, the aim of this exercise is to study the complexity of computing u_N for large N in terms of number of arithmetic operations $(+, \times)$ in $\mathbb{K}$.

Question 1. (1 pt) If the recurrence is used to write a loop that computes each u_n from the previous ones, express the asymptotic complexity of computing u_N as a function of N and k.

Solution. The loop would look like

```
for i in range(k,n): u[i] = a[k-1]*u[i-1]+...+a[0]*u[i-k]
```

The computation of u_N thus uses $N-k+1$ evaluations of this loop, each of which is obtained with at most k multiplications and $k-1$ additions. Thus the complexity to compute u_N this way is bounded by

$$(N-k+1)(2k-1) = O(kN). \qquad \square$$

Question 2. (2 pts) A method was given in Lecture 2 to compute the Nth Fibonacci number in $O(\log N)$ arithmetic operations in $\mathbb{K}$. Give the corresponding matrix $M_{\mathcal{R}}$ (of size $k \times k$) for the recurrence $(\mathcal{R})$ and express the asymptotic complexity of that method as a function of N and k, using a fast matrix product.

Solution. If U_n denotes the column vector $(u_n, u_{n-1}, \ldots, u_{n-k+1})$, then the recurrence $(\mathcal{R})$ implies $U_n = M_{\mathcal{R}} U_{n-1}$ with

$$M_{\mathcal{R}} = \begin{pmatrix} a_{k-1} & \ldots & a_1 & a_0 \\ 1 & \ldots & 0 & 0 \\ \vdots & \ddots & & \vdots \\ 0 & \ldots & 1 & 0 \end{pmatrix},$$

a matrix formed of a first line that encodes the recurrence, below which the identity matrix is completed by a column of 0.

Using the matrix, the computation uses

$$U_N = M_{\mathcal{R}}^{N-k+1} U_{k-1},$$

and the $N-k+1$st power of $M_{\mathcal{R}}$ is computed by binary powering in $O(\log N)$ multiplications of $k \times k$ matrices, each in $O(k^{2.3728})$ operations in $\mathbb{K}$, by a result of the same lecture. Thus the complexity of this algorithm is $O(k^{2.3728} \log N)$ operations in $\mathbb{K}$. $\square$

Question 3. (2 pts) Let P be the polynomial

$$P(X) = X^k - a_{k-1}X^{k-1} - \cdots - a_0 \in \mathbb{K}[X].$$

Give a divide-and-conquer algorithm that computes $X^N \bmod P$ in $O(\log N)$ operations in $\mathbb{K}[X]_{<k}$, the set of polynomials of degree less than k. Using a fast polynomial multiplication algorithm in $\mathsf{Mul}(k)$ operations for polynomials in $\mathbb{K}[X]_{<k}$, express its asymptotic complexity in terms of N and k.

Solution. The algorithm is binary powering modulo P:

$$X^n \bmod P = \begin{cases} (X^{n/2} \bmod P)^2 \bmod P, & \text{for even } n, \\ ((X^{(n-1)/2} \bmod P)^2 \bmod P) X \bmod P, & \text{for odd } n. \end{cases}$$

Each multiplication modulo P can be achieved by first multiplying two polynomials of degree $< k$ and then computing the remainder in the Euclidean division of the result by P. As seen in Lecture 4, both operations have complexity $O(\mathsf{Mul}(k))$, whence a total complexity in $O(\log_N \mathsf{Mul}(k))$ operations in $\mathbb{K}$ for the computation of $X^N \bmod P$. $\square$

Question 4. (2 pt) In the vector space $\mathbb{K}[X]_{<k}$ of polynomials of degree at most k, the map $Q \mapsto XQ \bmod P$ is linear. Show that, in the basis $(X^{k-1}, X^{k-2}, \ldots, 1)$, its matrix is exactly the transpose of the matrix $M_{\mathcal{R}}$ from question (2).

Solution. For $0 \leq i < k-1$, $X \cdot X^i \bmod P = X^{i+1} \bmod P$ which is an element of the basis. This shows that the columns $2, \ldots, k$ of the matrix of the map are given by the part of the matrix $M_{\mathcal{R}}$ below its first row. Next

$$X^k \bmod P = X^k - P = a_{k-1}X^{k-1} + \cdots + a_0,$$

giving the first column of the map, which is the transpose of the first row of the matrix $\mathcal{M}_{\mathcal{R}}$. □

Question 5. (2 pts) Show that for any $m \geq k$, the entries in the first row of $M_{\mathcal{R}}^{m-k+1}$ are the coefficients of $X^m \bmod P$ in the basis $(X^{k-1}, X^{k-2}, \ldots, 1)$.

Solution. From the previous question, it follows that ${}^tM_{\mathcal{R}}^i$ is the matrix of multiplication by $X^i \bmod P$ for $i \geq 0$. Thus its first column contains the coefficients of $X^{k-1+i} \bmod P$. Therefore the first column of ${}^tM_{\mathcal{R}}^{m-k+1}$ contains the coefficients of $X^m \bmod P$, which are thus the entries in the first row of $M_{\mathcal{R}}^{m-k+1}$. □

Question 6. (2 pts) Conclude by giving an algorithm computing u_N in asymptotic complexity logarithmic in N and linear in k up to logarithmic factors, assuming that primitive roots of unity are available in $\mathbb{K}$.

Solution. The algorithm consists in two steps:

1. *Compute $X^{N-k+1} \bmod P$ by binary powering modulo P, let V be the row vector of its coefficients;*
2. *Compute the product $V \cdot (u_{k-1}, \ldots, u_0)$.*

The second step costs k operations and is therefore negligible compared to the first one, which gives the complexity $O(\log_N \mathsf{Mul}(k))$. Using Fast Fourier Transform, this last estimate becomes $(k \log k \log N)$ when a primitive root of unity is available. □

Exercise 3: Load balancing

Given a set $L = (x_1, \ldots, x_n) \in \mathbb{N}^n$ that represents durations of n tasks and two machines that can work in parallel, the aim of load balancing is to split L into two disjoint subsets S_1 and S_2 (ie, $L = S_1 \cup S_2$ and $S_1 \cap S_2 = \varnothing$), attribute each set to one of the machines, so that the last task is completed as early as possible, ie, so that

$$M := \max(T_1, T_2) \quad \text{with} \quad T_i := \sum_{x \in S_i} x \text{ for } i = 1, 2$$

is minimal. Denote by M_{opt} this minimal value.

Question 1. (2 pts) Show that this problem is NP-hard. [Hint: use 2-Partition.]

Solution. The 2-Partition problem has a solution if and only if the load balancing problem with the same integers returns two sets whose sums are equal. This reduction shows that 2-Partition reduces to the load balancing problem. Since 2-Partition is NP-complete, it follows that load balancing is NP-hard. □

Question 2. (2 pts) Show that any strategy results in a splitting where

$$\max(T_1, T_2) \leq 2M_{\text{opt}}.$$

Solution. The total time $T_1 + T_2$ depends only on L and not on the algorithm. At best, M_{opt} is exactly its half, so that $M_{\text{opt}} \geq (T_1 + T_2)/2$, whence the result. □

We now consider the following greedy assignment method, where the list is first sorted by decreasing order and the tasks are assigned in order to the first machine that is available:

```
def greedypart(L):
    L = sorted(L, reverse=True)
    S1 = []
    S2 = []
    T1 = T2 = 0
    for i in range(0,len(L)):
        if T1<=T2: S1 += [i]; T1 += L[i]
        else: S2 += [i]; T2 += L[i]
    return S1,S2,T1,T2
```

Question 3. (1 pt) Show that this method is optimal for $n \leq 3$.

Solution. For $n = 1$ and $n = 2$, $\max(T_1, T_2) = \max(L)$ which is optimal. For $n = 3$, if $x_1 \geq x_2 \geq x_3$ (without loss of generality), then in the optimal solution, the machine assigned x_1 will not run another task, since $x_1 + x_3 \geq x_2 + x_3$. So the solution found by the algorithm, namely $(x_1), (x_2, x_3)$, is optimal. □

Question 4. (4 pts) The input $L = (3, 3, 2, 2, 2)$ shows that the algorithm is not always optimal. Show that in general,

$$\max(T_1, T_2) \leq \frac{4}{3} M_{\text{opt}}.$$

[Hint: first show that the algorithm is optimal when one of S_1, S_2 contains only one element. Next, consider the last task that terminates otherwise.]

Solution. To simplify notation, we assume that $x_1 \geq \cdots \geq x_n$ and write $T := T_1 + T_2$.

The algorithm is optimal if one of the sets S_1, S_2 contains only one element: if $n = 1$ or $n = 2$ we already know that the algorithm is optimal. Otherwise, that element that is alone is x_1 and since the algorithm has assigned all the other tasks to S_2, $x_2 + \cdots + x_{n-1} \leq x_1$, so that the partition $(x_1), (x_2, \ldots, x_n)$ is indeed optimal.

Without loss of generality, assume $T_1 \geq T_2$. If both sets have at least two elements, letting x_j be the last element in S_1, necessarily $j \geq 3$. From

$$x_1 \geq x_2 \geq x_j \quad \text{and} \quad x_1 + x_2 + x_j \leq T,$$

it follows that $x_j \leq T/3$. Since $T_1 - T_2 \leq x_j$, it follows that $2T_1 = T_1 - T_2 + T \leq x_j + T \leq 4T/3$. Therefore $T_1 \leq 2T/3 \leq 4M_{\text{opt}}/3$. □