École Polytechnique — CSE202 Design and Analysis of Algorithms

Final Exam: January 11th, 2022

Guidelines. Algorithms can be described in pseudo-code or Python code. It is **not** necessary to write standard algorithms like sorting or the merge procedure in Merge Sort. You can just describe where and how they are used.

The number of points indicated in front of each question is an indication of the relative difficulties of the questions.

Some solutions are only sketched and lack additional justification required for a maximal grade.

Divide & Conquer

Question 1 (3 points)**.** *A k-way merge operation.* Suppose you have k sorted arrays, each with n elements, and you want to combine them into a single sorted array of $k \cdot n$ elements.

— Here's one strategy : Using the merge procedure from Merge Sort, merge the first two arrays, then merge in the third, then merge in the fourth, and so on. For instance, given three sorted arrays $A_1 = [1, 5, 8]$, $A_2 = [2, 3, 6]$ and $A_3 = [0, 4, 9]$, we first merge A_1 and A_2 into $A_{12} = [1, 2, 3, 5, 6, 8]$, and second, merge A_{12} and A_3 to get the final result $A = [0, 1, 2, 3, 4, 5, 6, 8, 9]$.
What is the time complexity of this algorithm, in terms of k and n ?

— Give a more efficient solution to this problem, using divide-and-conquer. Justify its complexity.

Solution :

— since the complexity of merging two arrays A and B of size n_1 and n_2, respectively, is $n_1 + n_2$, the complexity of the overall algorithm equals

$$2n + 3n + .. + kn = O(n \cdot k^2)$$

— divide the k arrays into two halves containing an equal number of arrays and call the merging function recursively on each half. Then, apply the standard merge function in Merge Sort on the results obtained from the recursive calls. The base case corresponds to a set containing one array where we simply return that array. The complexity becomes $O(n \cdot k \cdot log\ k)$.

□

Question 2 (3 points)**.** Consider the problem of computing the *number* of *pairs* of integers in an input array that sum to 0. We assume that all the integers are distinct and different from 0. For instance, for the array $A = [-1, 2, 3, 1, -2]$ there are two such pairs $(-1, 1)$ and $(2, -2)$. Describe an algorithm for solving this problem that has the best complexity you can think of. Justify its complexity.
Hint : Sorting may help.

Solution : Here is a pseudo-code of the algorithm

— initialize a counter c to 0

— sort the input array A

— for each index i,

  — search for the value $-A[i]$ using binary search. Let j be the result of the search.

  — if j is an index in the array (i.e., the value $-A[i]$ exists) and $i < j$, then increment c

— return c.

The complexity is $O(n \cdot log\ n)$ where n is the size of the input array.

There is another solution where after sorting, we keep two pointers that initially point to the first and the last position, and iteratively advance one or another depending on the sum of the elements they point to. The complexity remains the same.

□

Randomized algorithms

Bloom Filters

Bloom filters are a *probabilistic* data structure used to store a set of items in a space efficient manner. They support two operations :

— *insert*(x) for adding a new item x, and

— *find*(x) which checks whether an item x is contained in the set.

A Bloom filter is implemented using a table T of size m, each location in the table has one bit, either 1 or 0. It uses k hash functions h_1, h_2, ..., h_k that map items to table indices $0, \ldots, m-1$. The two operations above are implemented as follows :

— *insert*(x) sets the locations $T[h_1(x)]$, ..., $T[h_k(x)]$ to 1.

— *find*(x) returns *true* (i.e., the item x is in the set) when *all* the table locations $T[h_1(x)]$, ..., $T[h_k(x)]$ are equal to 1, and *false*, otherwise.

Question 3 (1 point)**.** It is possible that *find*(x) returns *true* for an item x that has **not** been added to the filter before (i.e., there was no operation *insert*(x) that was executed before). This is called a *false positive*. Describe an execution showing a false positive for appropriately chosen hash functions.

Argue that *find*(x) **cannot** return *false* for an item x that has been added to the filter before.

Solution : For instance, if some other x' has been added to the filter, and $h_i(x) = h_i(x')$ for all $1 \leq i \leq k$, then *find*(x) returns *true*. This is a false positive.

The fact that *find*(x) **cannot** return *false* for an item x that has been added to the filter follows from the definition of Bloom filters. *find*(x) would return *false* if $T[h_i(x)] = 0$ for some $1 \leq i \leq k$, which is not possible after inserting x. Locations are never reverted to 0 after being set to 1. □

Question 4 (1 point)**.** Is it true that increasing or decreasing the number k of hash functions decreases the probability of false positives ? Justify your answer in both cases (increasing or decreasing k).

Solution : Increasing k decreases the probability of collisions, i.e., the probability that for two x and x', $h_i(x) = h_i(x')$ for all $1 \leq i \leq k$, but it requires using more locations in the table, which increases the probability that for some x, there exists a set of items $x_j \neq x$ with $1 \leq j \leq k$ such that $h_j(x) = h_i(x_j)$ for some i. The latter increases the probability of a false positive, since after adding the x_j elements, *find*(x) returns true. Therefore, increasing k may not decrease the probability of false positives.

Decreasing k may increase the probability of collisions, i.e., the probability that for two x and x', $h_i(x) = h_i(x')$ for all $1 \leq i \leq k$, which again, does not decrease the probability of false positives. □

Question 5 (4 points)**.** Assume that the hash functions are independent and that they are uniform, i.e., for each $i \in [1, k]$ and $j \in [0, m-1]$, the probability that $h_i(x) = j$ for a random item x is $1/m$. What is the probability of a false positive after n elements have been inserted ?

Solution : After inserting n elements, there are at most $n \cdot k$ locations which are set to 1. Given an item x, let us look at the probability that $T[h_i(x)]$ is equal to 0, for some $i \in [1, k]$ (after the n insertions). This happens if all the previous $n \cdot k$ locations are different from $h_i(x)$, which for each location, happens with probability $1 - \frac{1}{m}$. Therefore,

$$Pr[\ T[h_i(x)] = 0\ ] \geq \left(1 - \frac{1}{m}\right)^{kn} \approx e^{-kn/m}$$

The latter holds because $(1 - \frac{1}{m})^m \approx e^{-1}$. This is only a lower bound because it is possible that less than $n \cdot k$ locations are set to 1.

We get that

$$Pr[\ T[h_i(x)] = 1\ ] \leq 1 - e^{-kn/m}$$

Now, *find*(x) returns *true* if *all* the $T[h_i(x)]$ equal 1, which happens with probability

$$Pr[\ find(x) = true\ ] \leq \left(1 - e^{-kn/m}\right)^k$$

□

Amortized Analysis

Question 6 (3 points)**.** Imagine we want to store a big binary counter in an array A. All the entries of A are initially set to 0 and at each step we will be simply incrementing the counter. We assume a complexity model where we count the number of times we flip some bit (entry) of the array (from 1 to 0 or from 0 to 1). For instance, here is a trace of the first few increment operations and their cost in terms of complexity :

A[m]	A[m-1]	...	A[3]	A[2]	A[1]	A[0]	
0	0	...	0	0	0	0	
0	0	...	0	0	0	1	first increment flips the bit $A[0] \rightsquigarrow$ 1 flip
0	0	...	0	0	1	0	second increment flips the bits $A[0]$, $A[1] \rightsquigarrow$ 2 flips
0	0	...	0	0	1	1	third increment flips the bit $A[0] \rightsquigarrow$ 1 flip
0	0	...	0	1	0	0	fourth increment flips the bits $A[0]$, $A[1]$, $A[2] \rightsquigarrow$ 3 flips

In a sequence of n increments, the worst-case complexity per increment is $O(log\ n)$, since at worst we flip $log(n) + 1$ bits. But, what is the amortized complexity per increment (the average complexity over a sequence of n increments) ? Justify the answer.
Hint. Think about how many times you flip the bit $A[0]$ *over the* n *increments. Then, how many times you flip* $A[1]$*, and so on.*

Solution : First, how often do we flip A[0] ? Answer : every time. How often do we flip A[1] ? Answer : every other time. How often do we flip A[2] ? Answer : every 4th time, and so on. So, the total cost spent on flipping A[0] is n, the total cost spent flipping A[1] is at most n/2, the total cost flipping A[2] is at most n/4, etc. Summing these up, the total cost spent flipping all the positions in our n increments is at most

$$\sum_{i=0}^{log_2\ n} \frac{n}{2^i} = O(2n)$$

The amortized complexity is 2. □

Search Trees

Question 7 (2 points)**.** We consider the problem of computing the k-th smallest element of a set of elements stored in a binary search tree (BST), the so-called `select` function. As mentioned in class, this can be implemented efficiently by adding a field `size` to each node of the tree, which equals the number of elements in the sub-tree rooted at that node. You can find below the representation defined in class and the `insert` function.

```
class Node:
    def __init__(self,key,left=None,right=None,size=0):
        self.key = key
        self.left = left
        self.right = right
        self.size = size

class BST:
    def __init__(self):
        self.root = None
```

```
    def insert(self,key):
        self.root = self._insert(self.root,key)

    def _insert(self,node,key):
        if node is None: return Node(key)
        if node.key > key:
            node.left = self._insert(node.left,key)
        elif node.key < key:
            node.right = self._insert(node.right,key)
        node.size = 1+size(node.left)+size(node.right)
        return node
```

Write a function `select(self,k)` that returns the k-th smallest element in the tree (as a member of the class `BST`).

Solution :

```
    def select(self,k):
        return self._select(self.root,k)

    def _select(self,node,k):
        if node is None: return None
        if node.left.size == k-1:
            return node
        elif node.left.size > k-1:
            return self._select(node.left,k)
        else:
            return self._select(node.right,k - node.left.size - 1)
```

□

NP-completeness

Question 8 (3 points)**.** In the *half 3-CNF satisfiability problem*, we are given a 3-CNF formula ϕ with n variables and m clauses, where m is even. We wish to determine whether there exists a truth assignment to the variables of ϕ such that exactly half the clauses evaluate to 0 and exactly half the clauses evaluate to 1. Prove that the half 3-CNF satisfiability problem is NP-complete.
Hint. To prove NP-hardness, you can rely on a reduction from the standard 3-CNF problem seen in class. Consider translating a formula ϕ to the formula

$$\phi \wedge T \wedge \ldots T \wedge F \wedge \ldots \wedge F$$

with m copies of $T = y \vee y \vee \neg y$ and $2m$ copies of $F = y \vee y \vee y$, where y is a new variable (not present in ϕ).

Solution : A certificate would be an assignment to input variables which causes exactly half the clauses to evaluate to 1, and the other half to evaluate to 0. Since we can check this in polynomial time, half 3-CNF is in NP.

To prove that it's NP-hard, we show that 3-CNF can be reduced to half 3-CNF. Let ϕ be any 3-CNF formula with m clauses and input variables $x_1, x_2, \ldots, x_n$. Let $\phi' = \phi \wedge T \wedge \ldots T \wedge F \wedge \ldots \wedge F$ as above. Then ϕ' has $4m$ clauses and can be constructed from ϕ in polynomial time. Suppose that ϕ has a satisfying assignment. Then by setting $y = 0$ and the x_i's to the satisfying assignment, we satisfy the m clauses of ϕ and the m T clauses, but none of the F clauses. Thus, ϕ' has an assignment which satisfies exactly half of its clauses. On the other hand, suppose there is no satisfying assignment to ϕ. The m T clauses are always satisfied. If we set $y = 0$ then the total number of clauses satisfies in ϕ' is strictly less than $2m$, since each of the $2m$ F clauses is false, and at least one of the ϕ clauses is false. If we set $y = 1$, then strictly more than half the clauses of ϕ' are satisfied, since the $3m$ T and F clauses are all satisfied. Thus, ϕ has a satisfying assignment if and only if ϕ' has an assignment which satisfies exactly half of its clauses. We conclude that half 3-CNF is NP-hard, and hence NP-complete. □