ShanghaiTech University

CS240 Algorithm Design and Analysis

Fall 2022

Problem Set 1

Due: 23:59, Sept. 30, 2022

1. Submit your solutions to Gradescope (www.gradescope.com).
2. In "Account Settings" of Gradescope, set your FULL NAME to your Chinese name and enter your STUDENT ID correctly.
3. If you want to submit a handwritten version, scan it clearly. Camscanner is recommended.
4. When submitting your homework, match each of your solution to the corresponding problem number.

Problem 1:

Sort the following functions in ascending order of growth.

$f_1(n) = n^{\sqrt{n}}$

$f_2(n) = n^2$

$f_3(n) = (\log n)^{\log n}$

$f_4(n) = (\log n)^n$

$f_5(n) = n^{2/3}$

$f_6(n) = 666^{\sqrt{n}}$

$f_7(n) = 2^n$

$f_8(n) = 100^{100}$

Solution:

$f_8(n) = 100^{100}$

$f_5(n) = n^{2/3}$

$f_2(n) = n^2$

$f_3(n) = (\log n)^{\log n}$

$f_6(n) = 666^{\sqrt{n}}$

$f_1(n) = n^{\sqrt{n}}$

$f_7(n) = 2^n$

$f_4(n) = (\log n)^n$

Problem 2:

Give the time complexity of the following code and explain the reason.

(1)
```
for ( i=1; i < n; i *= 2 ) {
    for ( j = n; j > 0; j /= 2 ) {
        for ( k = j; k <= n; k += 2 ) {
            res += (i * j + j * k );
        }
    }
}
```

(2)
```
for ( i = n; i > 0; i -= 1 ) {
    for ( j = 1; j < n; j *= 2 ) {
        for ( k = 0; k < j; k += 1 ) {
            res += (i * j + j * k );
        }
    }
}
```

Solution:

(1) Running time of the inner, middle, and outer loop is proportional to $n, \log n$, and $\log n$, respectively. Thus the overall Big-Oh complexity is $O\left(n(\log n)^2\right)$.

More detailed optional analysis gives the same value. Let $n = 2^k$. Then the outer loop is executed k times, the middle loop is executed $k+1$ times, and for each value $j = 2^k, 2^{k-1}, \ldots, 2, 1$, the inner loop has different execution times:

j	Inner iterations
2^k	1
2^{k-1}	$\left(2^k - 2^{k-1}\right)\frac{1}{2}$
2^{k-2}	$\left(2^k - 2^{k-2}\right)\frac{1}{2}$
$\ldots$	$\ldots$
2^1	$\left(2^k - 2^1\right)\frac{1}{2}$
2^0	$\left(2^k - 2^0\right)\frac{1}{2}$

In total, the number of inner/middle steps is

$$\begin{aligned}1 + k \cdot 2^{k-1} - \left(1 + 2 + \ldots + 2^{k-1}\right)\frac{1}{2} &= 1 + k \cdot 2^{k-1} - \left(2^k - 1\right)\frac{1}{2} \\ &= 1.5 + (k-1) \cdot 2^{k-1} \equiv (\log_2 n - 1)\frac{n}{2} \\ &= O(n \log n)\end{aligned}$$

Thus, the total complexity is $O\left(n(\log n)^2\right)$.

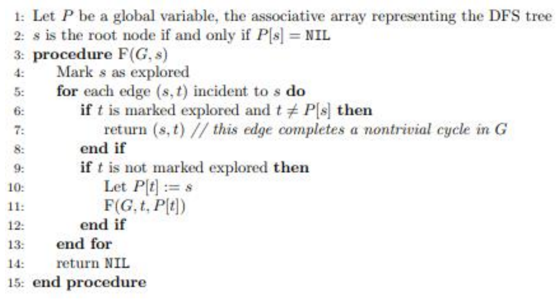
(2) The outer for-loop goes round n times. For each i, the next loop goes round $m = \log_2 n$ times, because of doubling the variable j. For each j, the innermost loop by k goes round j times, so that the two inner loops together go round $1 + 2 + 4 + \ldots + 2^{m-1} = 2^m - 1 \approx n$ times. Loops are nested, so the bounds may be multiplied to give that the algorithm is $O\left(n^2\right)$.

Problem 3:

Given a connected graph $G = (V, E)$ with at most $n + c$ edges where c is a constant and $n := |V|$. Find the MST of G in $O(n)$ running time. You should not only give a brief explanation of your algorithm but also prove the correctness and time complexity.

Solution:

Solution. The crucial subroutine is the following procedure DETECTCYCLEBYDEPTHFIRSTTRAVERAL which we will call simply F. It is a modification of the depth first traversal algorithm. We maintain and update an associative array P which contains the partial DFS tree. (We remark that even though G is an undirected graph, once we specify a root vertex, the DFS tree is a directed graph.) Thus $P[t] = s$ means that node t has node s as parent on the DFS tree. F takes as input a graph G, a node s, and the node which is the parent of s in the depth first tree P. Note that P is constructed by F and grows with each subsequent call to F. F returns the first nontrivial, non-tree edge (u, v), else it returns NIL to indicate that the graph G is already a tree. By nontrivial non-tree edge, we mean an edge that completes a nontrivial cycle in the undirected graph G; a trivial cycle in an undirected graph is one of the form $s \to t \to s$.



```
Let $P$ be a global variable, the associative array representing the DFS tree
$s$ is the root node if and only if $P[s] = $ NIL
procedure F($G, s$)
    Mark $s$ as explored
    for each edge $(s, t)$ incident to $s$ do
        if $t$ is marked explored and $t \neq P[s]$ then
            return $(s, t)$ // this edge completes a nontrivial cycle in $G$
        end if
        if $t$ is not marked explored then
            Let $P[t] := s$
            F($G, t, P[t]$)
        end if
    end for
    return NIL
end procedure
```

Now, with the above subroutine in hand, we may describe the idea of the algorithm to find an MST in a near-tree. We use F to search for a cycle. If there is no cycle, then the graph is a tree and we are done. Otherwise, we delete the heaviest edge from the cycle. We do this c times.

```
 1: procedure NearTreeMST($G$)
 2:     Choose an initial node $s$
 3:     for $i = 1, 2, \ldots, 9$ do
 4:         Let $(u, v) = F(G, s)$
 5:         if $(u, v) =$ NIL then
 6:             return // $G$ is already a tree
 7:         end if
 8:         Let $w =$ LeastCommonAncestor($u, v$)
 9:         Let $e_1$ be the heaviest edge on the path from $u$ to $w$
10:         Let $e_2$ be the heaviest edge on the path from $v$ to $w$
11:         Delete from $G$ the heaviest of $e_1$, $e_2$, $(u, v)$ // to use the Cycle Property of MSTs
12:     end for
13: end procedure
```

To analyze the complexity, write $n := |V|, m := |E|$, so that we have $n - 1 \leq m \leq n + c$ by connectedness and the near-tree property. The running time of F is bounded by that of DFS which is $O(n + m) = O(n)$ on a near-tree. LEASTCOMMONANCESTOR takes $O(n)$ time (see page 96 of KT). Finding the heaviest edge on a path takes $O(n)$ time, thus each of lines 9 and 10 takes $O(n)$. The total running time of this algorithm is thus $O(9n) = O(n)$, as required.

Problem 4:

In the new semester, SIST students have all enrolled in some courses. We asked n students "how many other students took the same course as you?" and collected the answers in an integer array $answer$ where $answer[i]$ is the answer of the i-th student.

Given an array $answer$, return the minimum number of students that could be in SIST.

Solution:

The approach of this problem is to find the **number of groups of students** that have enrolled in the same course and the **number of students** in each group. Below are steps:

- Initialize a variable **count** to store the number of students in each group.
- Initialize a map and traverse the array having key as $answer[i]$ and value as occurrences of $answer[i]$ in the given array.
- now if y students answered x: if $(y\%(x+1) == 0)$, then there must be $(y/(x+1))$ groups of $(x+1)$ students. If $(y\%(x+1)! = 0)$, there must be $(y/(x+1)) + 1$ groups of students.
- Add the product of the number of groups and the number of students in each group to the variable **count**.
- After the above steps, the value of **count** gives the minimum number of students in SIST.

Problem 5:

Suppose you are a security guard at ShanghaiTech University and now troubled by fraud detection. Since criminals may make some student cards to help thieves sneak into the campus, you have confiscated n student cards. Each card is a small plastic sheet with a magnetic stripe which encodes data, and belongs to a unique student. The cards have the same content on the surface because the student service department wants to save cost.

As a security guard, you have to find out whether there is someone colluding with the criminals and copying his student card. You has an equivalence tester, which can tell whether two cards belong to one person, but cannot tell who it is. (We say two cards are equivalent if they belong to one person.)

Assume that to invoke the equivalence tester, each time you can only take two cards and plug them into the equivalence tester. Please determine whether there is a set of more than *n/2* cards that are equivalent in n cards, i.e. belonging to one person, with only *O(n logn)* invocations of the equivalence tester.

Solution:

Divide and Conquer. For convenience, if have a set of n cards in which more than *n/2* cards are equivalent, regard this as a majority.

- Divide this set into two equal subsets.
- Find the majority of the two subsets recursively.
- *i)* If the two subsets have the same majority, then its the majority of the set. *ii)* If the two subsets have different majorities, then check if one of these majorities has more than half equivalents by iteration. If yes, its the majority of the whole set; if not, there is no satisfying result.
- In this case, check the subset requires *O(n)* time, it can be divided *O(log(n))* times, so its *O(n logn)*.