Midterm. 2020-10-22,

100 minutes (9:00 – 10:40)

Question 1.

There is a 2-dimensional array `arr` of size n by n; its elements are integer numbers (each number contains no more than n digits in its decimal representation).

You are given this C++ code with function `main()` calling function `fun()`.

```
1  int fun(int row) {
2    int count = 0;
3    for (int col=0; col<n; col++) {
4      while (arr[row][col]>0) {
5        arr[row][col] /= 2;
6        count++;
7      }
8    }
9    return count;
10 }
11
12
13 int main() {
14   for (int row=0; row<n; row++) {
15     cout << fun(row);
16   }
17 }
```

Analyze the time complexity of this algorithm "inside out":

(A) Estimate with some $O(g(n))$ the time spent on a single iteration of the loop in Lines 5-6. (How fast one can run these lines just once.)

(B) Estimate with some $O(g(n))$ the time spent on the loop Lines 4-7 (for one specific pair of row,col).

(C) Estimate with some $O(g(n))$ the time spent in the outer loop on Lines 3-8. (And also in a single call of `fun(...)`.)

(D) Estimate the time spent in the loop on Lines 14-16.

Note. We use integer division in this algorithm; for example `7/2` equals `3` and `1/2` equals `0`. So it will eventually stop.

Question 2.

Definition of Big-O:
Real-valued function with natural arguments $f(n)$ is in $O(g(n))$ iff there exist $C, n_0 > 0$ such that

$$|f(n)| \le C \cdot |g(n)|,$$

whenever $n > n_0$.

(A) Use the definition of Big-O Notation to prove or to disprove that $f(n) = 100n^2 + \frac{1}{4}n^3$ is in $O(n^3)$.

(B) Use the definition of Big-O Notation to prove or to disprove that $f(n) = (\log_2 n)^2$ is in $O(\log_2 n)$.

The reasoning should either provide the examples of C, n_0 that satisfy the definition for any n; or a method that for any given C, n_0 finds find n such that the definition is not satisfied.

Question 3.

Consider the Queue implementation from `https://bit.ly/35n7bKd`.

Denote a, b, c to be the last 3 digits of your Student ID, and compute the following numbers:

- $F = ((a + b + c) \mod 3) + 2$
- `x1` $= (a + b + c) \mod 10$
- `x2` $= ((a + b) \cdot 2) \mod 10$
- `x3` $= ((b + c) \cdot 3) \mod 10$
- `x4` $= ((c + a) \cdot 7) \mod 10$

The queue Q is implemented as an array of size $N = 6$; its elements have indices from $\{0, 1, 2, 3, 4, 5\}$.

Initially the queue parameters are these:
`Q.front` $= F$,
`Q.length` $= 4$,
`Q.size` $= 6$.
And the content of the array is the following:

i	0	1	2	3	4	5
array[i]	1	3	5	7	9	11

Somebody runs the following code on this queue:

```
Q.enqueue(x1)
Q.enqueue(x2)
Q.dequeue()
Q.dequeue()
// show the state of Q
Q.enqueue(x3)
Q.enqueue(x4)
Q.dequeue()
// show the state of Q
```

After Line 4 (and at the very end) show the current state of the queue `Q`. The state should display the content of the array and also the values of `Q.front` and `Q.length`.
You can use shading, if it helps to visualize the array cells that are not currently used by your queue.
Note. Painting something gray is not required (since front/length indicate the state of your queue anyway). But painting cells gray may be helpful, if you want to visualize where your queue has the useful values (and what is some old garbage – you can shade it over).

Question 4.

Introduction. Binary trees are often represented as arrays (where the array starts with the root node; followed by all the other nodes, displayed layer by layer. If any child of a node in this tree is missing, it is replaced by Λ (capital Lambda denoting an empty tree) in the array. Once we reach the last non-empty node in the tree, this is the last element of the array. For example, the binary tree shown in this picture:

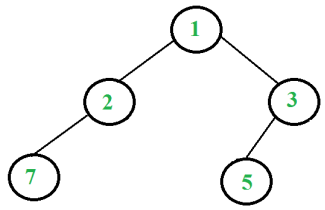

Figure 1: Example binary tree.

is represented by the following array:

$$\texttt{int a[]} = \{1, 2, 3, 7, \Lambda, 5\};$$

Problem. Assume that you have a binary tree that is represented by the following array:

$$\texttt{int a[]} = \{1, 2, 4, \texttt{a}, \Lambda, \Lambda, 6, \texttt{b}, \Lambda, \Lambda, \Lambda, \Lambda, \Lambda, \Lambda, \texttt{c}\}; \quad (1)$$

Values a, b, c are the last three digits taken from your Student ID.
Note. In the original the array contained a mistake (it had four Λ instead of six; but this was wrong, it does not correspond to any tree).

(A) Draw the binary tree represented by the array 1 in your answer. The tree should look nice: Draw left children to the left (and right children to the right) of their parents. Nodes on the same levels should be aligned.

(B) What is the number of internal nodes in this tree? The number of leaves in this tree?

(C) List the vertices of this tree in the post-order traversal order.
Note. You only show real nodes in the post-order sequence (all Λ are just technical symbols indicating absence of nodes; they are not part of the tree).

(D) Write pseudo-code for an algorithm GETPARENT(i) that receives the index i of some node in this array, returns the index of the parent of this node (or -1, if the node has no parent). All indices i are zero-based (in an array of length 10, $i \in \{0, \ldots, 9\}$).

(E) Assume that there is a different array (representing another binary tree) which does not contain any Λ values; all values there represent some nodes. Describe the property such trees must satisfy.

Question 1.

```
1  int fun(int row) {
2    int count = 0;
3    for (int col=0; col<n; col++) {
4      while (arr[row][col]>0) {
5        arr[row][col] /= 2;
6        count++;
7      }
8    }
9    return count;
10 }
11
12
13 int main() {
14   for (int row=0; row<n; row++) {
15     cout << fun(row);
16   }
17 }
```

(annotations: lines 5–6: $O(1)$; lines 4–7: $\times O(n)$; lines 3–8: $\times n$; lines 14–16: $\times n$)

Figure 2: Time complexity (inside out).

To estimate time complexity from the given code or pseudocode, you can work inside out (first estimate the time complexity for a single loop iteration, then estimate how many times the loop itself is executed, then multiply both estimates, etc.).

(A) Lines 5–6.
Time is $O(1)$, since these are just two operations on 4-byte integers. (It is obviously true for the regular `int` type. Even in the case when n is very large number and the array elements do not fit into the 4-byte register, division by 2 can take constant time, if number is written into binary notation: it is just the right shift.)

(B) Lines 4–7.
Time is $O(n) \cdot O(1) = O(n)$.
To verify this claim, note that any array element is a number with n digits (in decimal notation). Its value is at most 10^n. In order to have k iterations of the **while** loop (before `arr[row][col]` turns to 0) we should have $10^n \geq 2^k$, i.e. $k \leq n \cdot \log_2 10$ is in $O(n)$.

(C) Lines 3–8.
Time is $n \cdot O(n) = O(n^2)$.
Indeed, the for-loop (Line 3) executes n times; so we multiply n by $O(n)$ and get $O(n^2)$.

(D) Lines 14–16.
Time is $O(n) \cdot O(n \cdot n) = O(n^3)$. Method `main()` calls function `fun` n times.

Question 2.

(A) Answer: True.
We prove that $f(n) = 100n^2 + \frac{1}{4}n^3$ is in $O(n^3)$.

Proof. Select $n_0 = 1$, $C = 100\frac{1}{4}$. Denote $g(n) = n^3$. We check that $f(n)$ is in $O(g(n))$. For each $n \geq n_0 = 1$ we have $n^2 \leq n^3$ and therefore

$$|f(n)| = |100n^2 + \frac{1}{4}n^3| \leq |100n^3 + \frac{1}{4}n^3| =$$

$$= |100\frac{1}{4}n^3| \leq 101\frac{1}{4}|n^3| = C \cdot |g(n)|.$$

(B) Answer: False.
We state that $f(n) = (\log_2 n)^2$ is not in $O(g(n))$, where we denote $g(n) = \log_2 n$.
Assume from contrary that somebody has selected n_0 and C such that for every $n \geq n_0$ we have:

$$|f(n)| \leq C|g(n)|.$$

We pick n such that $\log_2 n > C$ or $n > 2^C$ and simultaneosly $n \geq n_0$ (we can have the maximum of both 2^C and n_0).

Note. This example can also demonstrate, why it is not easy to estimate the Big-O notation from computer-generated function graphs. If you pick, say $C = 100$, the values n where $|f(n)|$ exceeds $C|g(n)|$ may become quite large.

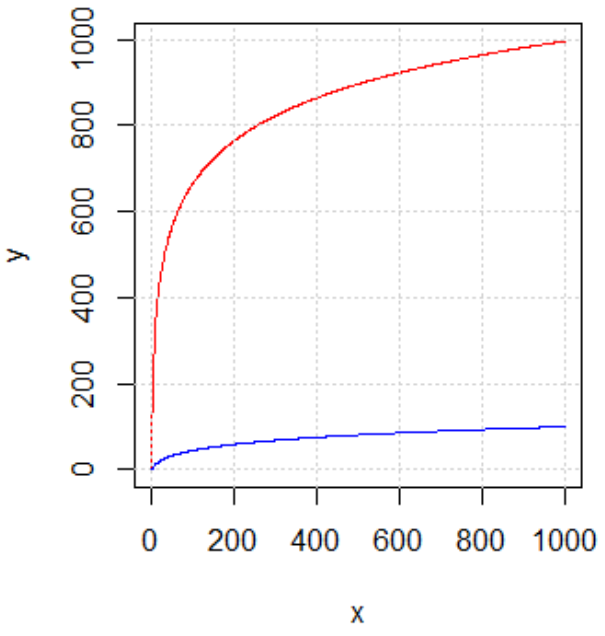

Figure 3: Insufficient interval for x.

See Figure 3. It seems that the red graph $y = 100 \cdot g(x) = 100 \cdot \log_2 x$ is much larger than

the blue graph $y = f(x) = (\log_2 x)^2$. The problem with these graphs is the very short interval $x \in [1; 1000]$ where these functions are compared.
If you consider $x \in [1; 5 \cdot 10^{30}]$, it is easy to see that the logarithm squared (the blue function $f(x)$) overtakes the red function $100 \cdot g(x) = 100 \cdot \log_2 x$. See Figure 4. In practice it is much better to use limit calculus (because with function graphs it is hard to tell, how far you need to draw before you can see which expression is bigger):

$$\lim_{x\to\infty} \frac{(\log_2 x)^2}{\log_2 x} = \lim_{x\to\infty} \log_2 x = +\infty.$$

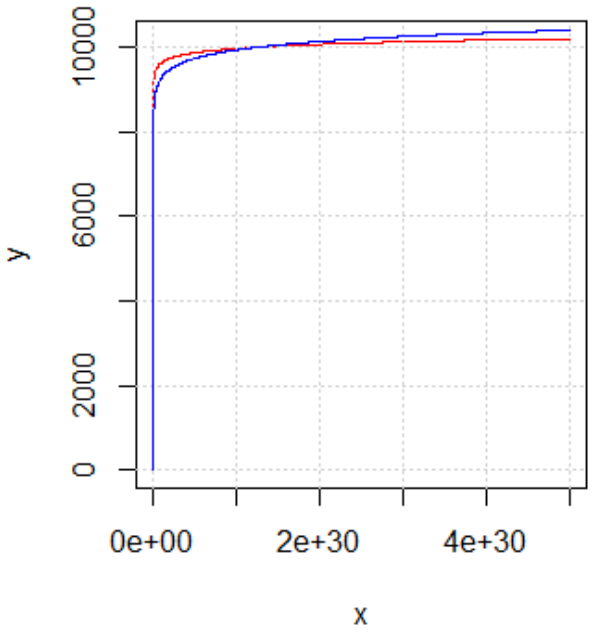

Figure 4: $g(x) = (\log_2 x)^2$ not in $O(\log_2 x)$.

Question 3. Let us show queue after each queue operation. Let us consider one particular example. for your a, b, c the result may look different; it might be "rotated", because F (the front pointer) is different. Also the newly enqueued values $(x_1, \ldots)$ will likely be different. Your queue will also have 4 entries (unshaded cells) at the start and 5 entries at the very end.
We take $abc = 789$. For these values we have these initial values:

$$F = 2, \quad L = 4, \quad x_1 = 4, \quad x_2 = 0, \quad x_3 = 1, \quad x_4 = 2.$$

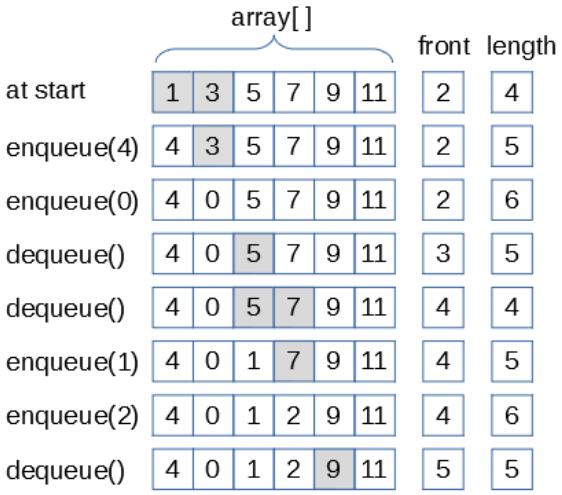

Figure 5: Queue states.

Question 4.
(A) The array and corresponding tree are shown below:

`int a[] = {1, 2, 4, a, Λ, Λ, 6, b, Λ, Λ, Λ, Λ, Λ, Λ, c};`

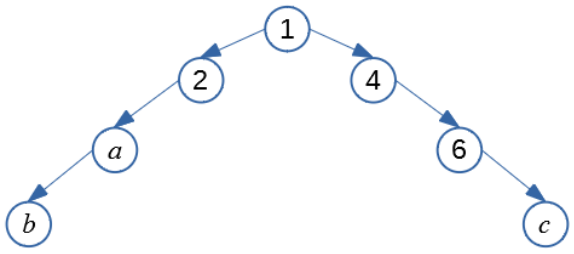

Figure 6: Binary tree for the array.

(B) There are 5 internal nodes and 2 leaves.

(C) Postorder sequence for the nodes (namely, first visit both subtrees in postorder, then the parent) is the following:

$$b, a, 2, c, 6, 4, 1.$$

(D) The pseudocode is given below:

```
getParent(i)
1  if i mod 2 == 0 :
2      return (i − 2)/2
3  if i mod 2 == 1 :
4      return (i − 1)/2
```

You can also merge both formulas and get this formula:

$$\textsc{getParent}(i) = \left\lfloor \frac{i-1}{2} \right\rfloor .$$

Here we use the floor function (rounding down).

(E) Such trees are called *complete trees* (all levels in such trees are full, except maybe the last level, which can be only partially full, but is filled without any interruptions from the left side).