# FINAL (2022-05-13)

See https://bit.ly/3KxuMdV for the official information on the time and location of this exam.

**Question 1:**

**(A)** Order functions in *increasing* order by their asymptotic growth rate: Write them in some sequence $g_1, g_2, \ldots, g_8$ such that $g_1 = O(g_2)$, $g_2 = O(g_3)$, ..., $g_7 = O(g_8)$.

$$f_1(n) = n^e, \quad f_2(n) = e^n, \quad f_3(n) = \binom{n}{n-3}, \quad f_4(n) = \sqrt{2^{\sqrt{n}}},$$
$$f_5(n) = \binom{n}{4}, \quad f_6(n) = 2^{\log_2^4 n}, \quad f_7(n) = n^{3(\log_2 n)^2}, \quad f_8(n) = (n-1)(n+1)(n^2+1)(n^4+1).$$

**(B)** Find function $g(n)$ such that $T(n)$ is in $\Theta(g(n))$. Here $T(n)$ is defined by a recurrence:

$$T(n) = \begin{cases} T\left(\left\lfloor \frac{n}{3} \right\rfloor\right) + T\left(\left\lfloor \frac{2n}{3} \right\rfloor\right) + n, & \text{if } n > 0, \\ 1, & \text{if } n = 0. \end{cases}$$

**Answer:**

**(A)** To classify the functions, we identify functions with polynomial growth rate ($f_1, f_3, f_4, f_5$) and order them by the exponent of the polynomial, which is either visible in the expression of that function or can be found by some algebra.

Next, we identify functions growing faster than polynomials (typically, because their exponent is not constant, but growing). They are $f_7, f_6, f_4, f_2$. Some of them might seem difficult to compare. Taking logarithm from both sides may help. Note that all polynomial expressions $P(n)$ that are in *Theta(n^k)* (regardless of the exponent $k$) have logarithms $\log_2 P(n)$ that are in $\Theta(k \log_2 n) = \Theta(\log_2 n)$. So, if the logarithm of some function grows faster than $\log_2 n$, then it grows faster than a logarithm.

1. $f_1(n) = n^e \approx n^{2.71828}$ (we claim that this is the slowest growing function from the list; it is in $O(n^3)$, observe that for large $n$ it grows slower than the polynomials of the 3rd degree),
2. $f_3(n) = \binom{n}{n-3} = \frac{n(n-1)(n-3)}{6} = \Theta(n^3)$ (grows as fast as $n^3$).
3. $f_5(n) = \binom{n}{4} = \frac{n(n-1)(n-2)(n-3)}{24} = \Theta(n^4)$ (grows as fast as $n^4$).
4. $f_8(n) = (n-1)(n+1)(n^2+1)(n^4+1) = \Theta(n^8)$ (grows as a polynomial of 8th degree).
5. $f_7(n) = n^{3(\log_2 n)^2}$, since $\log_2 f_7(n) = 3\log_2^2 n \cdot \log_2 n = \Theta(\log_2^3 n)$ – any polynomial (including $f_8(n)$ will be in $O(f_7(n))$, since the logarithms of polynomials are in $\Theta(\log_2 n)$, so they do not grow as fast as $\Theta(\log_2^3 n)$.
6. $f_6(n) = 2^{\log_2^4 n}$, observe that $\log_2 f_6(n) = \log_2 2^{\log_2^4 n} = \log_2^4 n$.
7. $f_4(n) = \sqrt{2^{\sqrt{n}}}$, observe that $\log_2 f_4(n) = \log_2 2^{\left(\frac{1}{2}\sqrt{n}\right)} = \frac{1}{2}\sqrt{n}$. Observe also that the square root grows faster than any power of a logarithm. To see that $f_6(n)$ is in $O(f_4(n))$ verify that the ratio of their logarithms $\lim_{n\to\infty} \frac{\log_2^4 n}{\sqrt{n}} = 0$. (You could try to use L'Hopital's rule to find this limit).

8. $f_2(n) = e^n$. This is the fastest growing function of all, since its logarithm is $\log_2 f_2(n) = \log_2 e^n = n \cdot \log_2 e = \Theta(n)$. This function $n$ grows faster than the square root function from the previous function.

Many more examples how to arrange functions by their asymptotic growth rate (Big-O) can be found in other sources, for example, https://bit.ly/3HR8yUi.

**(B)** Answer: $T(n) \in \Theta(n \log n)$. To understand the growth rate, compute a few first values of this function:

| $n$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $T(n)$ | 1 | 3 | 6 | 12 | 13 | 20 | 25 | 26 | 34 | 46 | 47 |

For example, $T(10)$ is computed like this:

$$T(10) = T(3) + T(6) + 10 = (T(1) + T(2) + 3) + (T(2) + T(4) + 6) + 10 =$$
$$= T(1) + 2T(2) + T(4) + 19 = (T(0) + T(0) + 1) + 2 \cdot (T(0) + T(1) + 2) + (T(1) + T(2) + 4) + 19 =$$
$$= (1 + 1 + 1) + 2 \cdot (1 + 3 + 2) + (3 + 6 + 4) + 19 = 47.$$

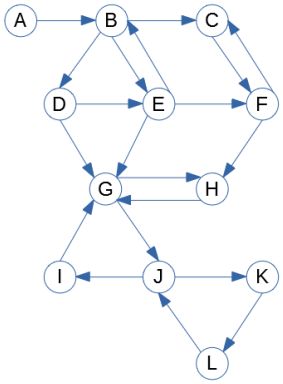
The recurrence is similar to those solvable by Master theorem, https://bit.ly/3OpaxSl ; one could get such a recurrence for a variant of Mergesort algorithm (which splits an array of $n$ elements into two subarrays of unequal lengths $n/3$ and $2n/3$ respectively). Let us show that the function $T(n)$ is in $O(n \log_2 n)$.

We can apply the recurrence formula certain number of times depending on $n$. Every time it is applied to all the terms $T(n_i)$, they split into smaller terms (for example, $T(n_i) = T(n_i') + T(n_i'') + n_i$, where $n_1' + n_i'' \approx n_i$ (it may be slightly smaller because we are taking the floor function).

The total number of times we can multiply some number $n$ with $(2/3)$ until the result is less than 1 is equal to $\log_{3/2} n = \Theta(\log_2 n)$. So the splitting can happen no more than $C \cdot \log_2 n$ times, where $C$ is some positive constant. Every time we split, we also add a number which equals to $n$ (or slightly smaller).

Finally, we get that $T(n)$ is in $O(n \log_2 n)$ since we add quantity $O(n)$ every time we split and we split $O(\log_2 n)$ times. We observe that the regular Merge sort also has the same time complexity.

**Question 2:** Run Kosaraju algorithm to find strongly connected components.




**(A)** Run the DFS on the given graph. (Whenever you run out of traversable nodes and need to pick a new vertex, select the alphabetically first one. Also, when you visit the children of a given node – visit them in an alphabetical order.)

Label each vertex with two numbers `d/f`, where `d` is the discovery time, but `f` is the finishing time during the DFS traversal. (Both `d` and `f` are integers from the interval $[1; 24]$.)

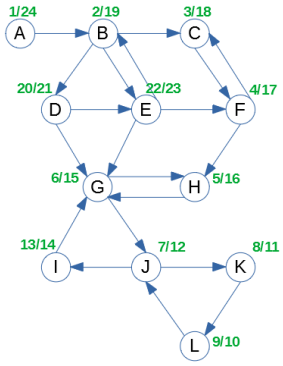
**(B)** As requested by the Kosaraju algorithm, transpose the directed graphs matrix (i.e. flip all the arrows), and run the DFS again. (This time the ordering of the nodes is different: In Kosaraju algorithm they are determined by the `d/f` numbers found in the previous step. Use this ordering every time when you run out of traversable nodes or need to visit the children nodes.)

In this new graph mark all the strongly connected components you have found. (Components are marked by drawing an oval shape around all the nodes in that component.)

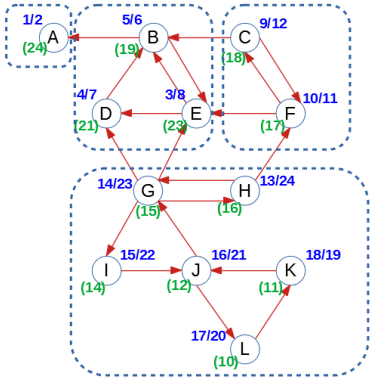
**(C)** Estimate the time complexity of this algorithm – find the smallest (slowest growing) $g(n)$ such that the runtime of the algorithm $T(n)$ is in $O(g(n))$.

**Answer:**

**(A)** We run DFS algorithm on the original graph, visit all vertices in alphabetic order, and mark each vertex with two numbers (time moment when DFS enters the node – discovery time, time moment when DFS exits the node – finishing time). Both numbers are written in green.




**(B)** Now we flip all the edges in the original graph (now marked dark red). Also preserve the finishing times of the DFS done in the previous step (A) – shown in green. Now we visit all vertices by decreasing finishing time (starting from the vertex "A" marked "24", we immediately run into a dead end, next go to the vertex "E" marked "23" and so on). Now show the discovery/finishing times in this next DFS traversal in blue.




**(C)** The answer depends on how the graph is represented and what is meant by the parameter $n$. Let $n$ denote the number of vertices in a graph; and let $m$ denote the number of its edges. Kosaraju's algorithm means running DFS algorithm twice; it costs $O(n+m)$ (and in-between the graph is transposed by reversing all its edges; it costs $O(m)$). The total time is $O(n+m)$.

If the graph is represented as an $n \times n$ matrix, then the DFS algorithm needs to scan all the rows (or all the columns for the transposed graph); it costs $O(n^n)$.

**Question 3:** Consider the following sequence of hexadecimal digits:

$$T = \texttt{255044462D312E35}\ldots$$

The hash function is determined by the following formula:

$$h\left(a_0 a_1 a_2 \ldots a_{n-1}\right) = \left(\sum_{k=0}^{n-1} a_k 16^{n-1-k}\right) \bmod 109$$

**(A)** Select the window size $s = 6$ and find the hash values of the first two substrings ($P_0 = \texttt{255044}$ and $P_2 = \texttt{550444}$) in the given text.

**(B)** Describe the algorithm (using constant time) to skip a digit from the start of the pattern. Your algorithm can use the previous hash value and the window size $s$ as inputs.

Run this algorithm to get $h(P_1) = h(\texttt{55044})$ from $h(P_0) = h(\texttt{255044})$ skipping the hex digit "2" at the very beginning (assuming that the window size is $s = 6$).

**(C)** Describe the algorithm (using constant time) to append a digit to the end of the pattern. Your algorithm can use the previous hash value and the window size $s$ as inputs.

Run your algorithm to get $h(P_2) = h(\texttt{550444})$ from $h(P_1) = h(\texttt{55044})$. (Your hash value for $h(P_2)$ should coincide with the value obtained in (A).)

**Answer:**

**(A)** The rolling hash algorithm (summing the polynomial expression with base 16) means that the hashable strings need to be understood as hexadecimal numbers and then divided by the prime number 109 and hashcode is the remainder.

$$h(\text{"}\texttt{255044}\text{"}) = \texttt{255044}_{16} \bmod 109 = 2445380 \bmod 109 = 74$$
$$h(\text{"}\texttt{550444}\text{"}) = \texttt{550444}_{16} \bmod 109 = 5571652 \bmod 109 = 8$$

Such hash function behaves well, probabilities for all the remainders $\{0, 1, \ldots, 108\}$ are distributed uniformly. It could be used just like any other hash function.

The next question is about the efficiency (since the sliding window only shifts forward one byte at a time and the inputs can be sufficiently long – much longer than the 6 digits in our example). For relatively short hexadecimal numbers one could make this faster by doing memory copy and some bit masking. But for other bases (except $2, 4, 8, 16, \ldots$) and for longer sliding windows one would need to scan through all the digits, so it is roughly $O(s)$, where $s$ is the length of the window. For example, if $s = 6$, then the computation is this:

$$\texttt{255044}_{16} = (((((2 \cdot 16 + 5) \cdot 16 + 5) \cdot 16 + 0) \cdot 16 + 4) \cdot 16 + 4) \bmod 109$$

In the subsequent subtasks (B), (C) we need to speed this up.

**(B)** Here is the pseudocode to skip digit $d$ and the current window size (the length of the hashable string) is $s$. We also need *oldHash* – the previous hash function. Algorithm also uses two constants $b = 16$ and $m = 109$.

Note that we do not need to know the actual hashable string, since the SKIP$(\ldots)$ must work in $O(1)$ time, we cannot scan the entire text, only do a fixed number of arithmetic operations.

SKIP$(\mathit{oldHash}, d, s)$:
  $\mathit{digitValue}(s) = 16^{s-1} \bmod 109$
  **return** $(\mathit{oldHash} - \mathit{digitValue}(s) \cdot d) \bmod 109$

It might seem that $\mathit{digitValue}(s)$ needs to raise the base 16 to the power $s - 1$, which cannot be done in constant time. On the other hand, such values (for all relevant sliding window sizes $s$) can be precomputed and cached; so it does not slow down the rolling hash algorithm.

**The numeric example:** As we know, $h(\text{"}\texttt{255044}\text{"}) = 74$. Once we skip the first digit "2", we run the above pseudocode to compute SKIP$(74, 2, 6)$

$$(74 - 2 \cdot 16^{6-1}) \bmod 109 = 82.$$

This coincides with the hashfunction value computed inefficiently:

$$h(\text{"}\texttt{55044}\text{"}) = \texttt{55044}_{16} \bmod 109 = 348228 \bmod 109 = 82.$$

**(C)** Here is the pseudocode for appending a new character (digit) $d$.

APPEND($oldHash, d$):
    **return** $(16 \cdot oldHash + d) \bmod 109$

Appending a new digit to the end of a hexadecimal number means multiplying all the accumulated digits and their weights by the base 16 (this is equivalent to shifting the respective digit one position to the left) and adding the new digit $d$.

**Numeric example:** We have $h(\texttt{55044}) = 82$. Appending one more digit $d = 4$ leads to the following:

$$h(\texttt{550444}) = \textsc{append}(h(\texttt{55044}), 4) = \textsc{Append}(82, 4) = (16 \cdot 82 + 4) \bmod 109 = 8.$$

This value coincides with the result of the computation in (A).

**Question 4:**

**(A)** Build the KMP (Knuth-Morris-Pratt) prefix function to search for this pattern: $P = \texttt{ababacab}$.

**(B)** Show how KMP works to find *all occurrences* of the pattern in the following text: $T = \texttt{abaababacababa}$.

Namely, write all the letters of this text in a horizontal line. Under this text show multiple copies of the pattern (copied with different offsets so that letters in the pattern are compared to the letters in the text $T$ located directly above it). For each copy of the pattern circle those letters that were compared with the above text.

**(C)** How many letter-to-letter comparisons would be made, if $P$ is searched in $T$ using the naive search algorithm? How many were used by the KMP algorithm in (B)?

**Answer:**

**(A)** Here is the prefix function for the pattern $P = \texttt{ababacab}$:

| $j$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| $\pi(j)$ | 0 | 0 | 1 | 2 | 3 | 0 | 1 | 2 |

To verify that it is correct, find the overlaps of all prefixes of ababacab with themselves. The examples below show how the prefixes can be aligned with shifted copies with themselves.

```
aba      abab     ababa    ababac          ababaca         ababacab
  aba      abab     ababa         ababac          ababaca         ababacab
pi(3)=1  pi(4)=2  pi(5)=3  p(6)=0          p(7)=1          p(8)=2
```

**(B)** The KMP algorithm execution is shown in the image below. All the colored letters are those, which are actually compared to the pattern string (shown at the very top). The letters that match are shown blue, letters that do not match are shown red, if the matching reaches the end of the pattern (i.e. the entire pattern is found), it is shown in green.

The only full match of the pattern with the text happens when $i = 10$ and $k = 7$. From here we can compute the offset of the pattern which is $i - k = 10 - 7 = 3$. It is the only output of this algorithm. After that we try to shift the pattern to the right, but it cannot cause full match anymore since part of the pattern is outside the searchable text.

| $i$ = | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | a | b | a | a | b | a | b | a | c | a | b | a | b | a | |
| | **a** | **b** | **a** | **b** | a | c | a | b | | | | | | | k = 0, 1, 2, 3 |
| | | | a | **b** | a | b | a | c | a | b | | | | | k = 1 |
| | | | | **a** | **b** | **a** | **b** | **a** | **c** | **a** | **b** | | | | k = 0, 1, 2, 3, 4, 5, 6, 7 |
| | | | | | | | | | | a | b | **a** | **b** | **a** | k = 2, 3, 4 |

**(C)** The KMP string search compares 16 letters (all the bold ones in the above example).

If we use naive string search algorithm instead, it would need $4+1+2+8+1+4+1+2+1+5=29$ letter comparisons. See the diagram below – every time we find a mismatch, the pattern is shifted ahead by one unit only.

```
abaababacababa
--------------
abab
 a
  ab
   abababacab
    a
     abab
      a
       ab
        a
         ababa
```

**Question 5:** Consider this list of strings:

```
S = ["Croatia", "Iceland", "Ireland", "Denmark", "Bulgaria", "Andorra"]
```

Insert this list into a hash table using the following hash function in Python:

```
hash('ABC') % 8
```

The above expression calculates the hash value for the string `'ABC'`. To make this hash function predictable, before you run the Python command-line or the IDE, set the environment variable using one of these commands:

```
export PYTHONHASHSEED=0
  OR
$Env:PYTHONHASHSEED=0
  OR
set PYTHONHASHSEED=0
```

**(A)** Draw the contents of hash table (eight slots, 0 to 7), if the collisions are handled by linear probing.

**(B)** Assume we want to insert the seventh entry. This entry is yet unknown and the Python's hash function can take every remainder modulo 8 with the same probability. What is the expected number of comparisons before this entry can be inserted into the hash table? (If the entry inserts into an empty slot, it is one comparison; if it needs to do one linear probing to move to the next cell,it is two comparisons; if it does two linear probings, then it is three comparisons, etc.)

**(C)** Before the seventh entry was inserted, the hash table was considered full and its size was doubled from 8 to 16 slots. (The new hash function is `hash('ABC') % 16`). Draw the new contents of the hash table containing the same six entries as before.

**Answer:**

**(A)** Here is a short program computing hash function values:

```
>>> S = ["Croatia", "Iceland", "Ireland", "Denmark", "Bulgaria", "Andorra"]
>>> [hash(x) % 8 for x in S]
[7, 4, 2, 0, 1, 6]
```

Note that there are no collisions at all. The hash table looks like this:

| $T[n]$ | **Entry** |
|---|---|
| T[0] | `Denmark` |
| T[1] | `Bulgaria` |
| T[2] | `Ireland` |
| T[3] | NULL |
| T[4] | `Iceland` |
| T[5] | NULL |
| T[6] | `Andorra` |
| T[7] | `Croatia` |

**(B)** If the hash value of the new entry is 3 or 5 (both entries that are currently free) we need just one comparison. If the hash value is 2 or 4 (just one step before a free entry), we need two comparisons. If the hash value is 1, we need three comparisons. If the hash value is 0 we need four comparisons (as the first three slots are filled in). Finally, if the hash value is 7 or 6, we need five or six comparisons respectively (as the linear probing "wraps around" the hash table and starts probing at the beginning).

Here is the expression of the probability:

$$\frac{2}{8}\cdot 1+\frac{2}{8}\cdot 2+\frac{1}{8}\cdot 3+\frac{1}{8}\cdot 4+\frac{1}{8}\cdot 5+\frac{1}{8}\cdot 6=\frac{26}{8}=\frac{13}{4}.$$

**(C)** Here is the Python snippet we need:

```
>>> [hash(x) % 16 for x in S]
[15, 4, 2, 8, 1, 14]
```

| $T[n]$ | **Entry** |
|---|---|
| T[0] | `Denmark` |
| T[1] | `Bulgaria` |
| T[2] | `Ireland` |
| T[3] | NULL |
| T[4] | `Iceland` |
| T[5] | NULL |
| T[6] | NULL |
| T[7] | NULL |
| T[8] | NULL |
| T[9] | NULL |
| T[10] | NULL |
| T[11] | NULL |
| T[12] | NULL |
| T[13] | NULL |
| T[14] | `Andorra` |
| T[15] | `Croatia` |

Some random items (in our case `Andorra` and `Croatia`) have jumped 8 positions ahead in the new hashtable.