# ENGG1340 Programming Technologies / COMP2113 Computer Programming II

## Assignment 4

### Deadline: 3 May (Saturday), 2025 23:59

### General Instructions

Submit your assignment via VPL on Moodle. Ensure that your program can execute, and generate the required outputs in VPL. Work incompatible with the VPL may not be marked.

For C/C++ programs, ensure compilation with the gcc C11/C++11 standard on our standard environment. This is already enforced in VPL. If you are testing in your own environment, use the following commands to compile your programs:

For C++ programs:

```
g++ -pedantic-errors -std=c++11 -o [executable name] [yourprogram].cpp
```

For C programs:

```
gcc -pedantic-errors -std=c11 -o [executable name] [yourprogram].c
```

As a developer, ensure that your code works flawlessly in the intended environment, not just your own. While you may develop your work in your own environment, always test your program in our standard environment (i.e., the VPLs) before submission.

### Evaluation

For tasks requiring **user input**, utilize the **standard input**. Likewise, your program should **output/print** through the **standard output**. Strict adherence to the sample output format is required, or your answer may be marked incorrect.

Your code will be automatically graded for technical correctness. Essentially, we use test cases to evaluate your solution, failure to pass any of the test cases may result in zero marks. Partial credits are generally not given for incomplete solutions as it may be challenging to objectively assess incomplete program logic. However, your work may still be considered on a case-by-case basis during the rebuttal stage.

Additional test case will be used during grading. Scoring full marks on VPL does not ensure full marks in the assignment. Sample test cases may or may not encompass all boundary cases. Designing proper test cases to verify your program's accuracy is part of the assessment.

### Academic dishonesty

Your code will be cross-checked with other submissions and online sources for logical duplication. Note that providing your work to others, aiding others in copying, or copying from others will be considered plagiarism, and will be dealt with as per departmental policy. Please refer to the course information notes for more details.

**Use of generative AI tools, like ChatGPT, is not permitted** for all assignment.

### Getting help

You are not alone! If you are stuck, post your query on the course forum. This assignment should be educational and rewarding, not frustrating. We are here to help, but we can only do so if you reach out.

**Please avoid spoilers on the discussion forum.** Do not post any code directly related to the assignments. You are, however, encouraged to discuss general concepts on the forums.

### Submission

Deadlines are strictly enforced. Resubmission beyond the submission period will not be accepted.

Late Policy:

- If you submit within 2 days after the deadline, 30% deduction.
- If you submit within 3-5 days after the deadline, 50% deduction.
- After that, no mark.

## Problem 1: Bigram analyzer

A bigram is a sequence of two adjacent elements from a string. For example the token “Bigram!” has 6 bigrams: `Bi`, `ig`, `gr`, `ra`, `am`, `m!`.

Write a C++ program, `1.cpp`, that reads tokens from user input until an empty line is received. Count all bigrams in these tokens and output all unique tokens that contains the most frequently occurring bigram. While you may choose any method to solve this task, utilizing the Standard Template Library (STL) could simplify your solution.

### Input:

- The program should read lines from user input until an empty line is received.
- All input lines, except the last one, consist of one or more tokens. Tokens are separated by one whitespace character, and may consist of any printable ASCII characters.
- The last input line will always be an empty line.

### Output:

- The program should identify the most frequently occurring bigram and then output all **unique tokens** containing that bigram.
- Tokens must be listed in order of their first appearance.
- If there is no bigram in the input, the program outputs nothing.

### Requirements and assumptions:

- The program should be case sensitive. For instance, `Bigram!` and `bigram!` should be considered as two different tokens. Similarly, `Bi` and `bi` should be considered as two different bigrams.
- Tokens with only one character do not have a bigram.
- If multiple bigrams are the most frequently occurring, select the bigram that is first encountered as the most frequently occurring bigram.
- There is no length limit for the input. There could be as many token as that the VPL can handle.
- To avoid stability issues with VPLs and to adhere to memory limits, do not include `bits/stdc++.h`. Instead, include individual library as needed. Including `bits/stdc++.h` may cause your program to exceed memory limits and fail to compile.

### Sample Test Cases

| 1_1 | |
|---|---|
| Input: | `Here lists the 16 containers provided by Standard Template Library in C++11:`<br>`array, vector, deque, forward_list, list, set, map,`<br>`multiset, multimap, unordered_set, unordered_map,`<br>`unordered_multiset, unordered_multimap, stack, queue,`<br>`priority_queue` |
| Output:<br>(bigram is `or`) | `vector,`<br>`forward_list,`<br>`unordered_set,`<br>`unordered_map,`<br>`unordered_multiset,`<br>`unordered_multimap,`<br>`priority_queue` |

| 1_2 | |
|---|---|
| Input: | `Old MacDonald had a farm, E-I-E-I-O`<br>`And on that farm he had a cow, E-I-E-I-O` |
| Output:<br>(bigram is `E-`) | `E-I-E-I-O` |

| 1_3 | |
|---|---|
| Input: | `a e i o u` |
| Output: | *The program outputs nothing* |

## Problem 2: Run-length encoding

"Run-length" refers to the number of times a symbol is repeated consecutively. For example, the string `abbaaa` is seen as `a` with a run-length of 1, then `b` with a run-length of 2, and finally another `a` with a run-length of 3.

Run-length encoding encodes a sequence of symbols based on their run-length. Consider the following encoding scheme:

- To encode a string, the string is first split into runs of symbols, each with a maximum length of 10. For example, the string `abbbaaaaaaaaaaa` can be split into `a`, `bbb`, `aaaaa`, `aaaaaa`.
- For each run of symbols, encode it as the symbol followed by the a single digit that equals the run-length minus 1. For instance, `a` will be encoded as `a0`, and `bbb` will be encoded as `b2`.

Given an encoded string such as `a0b2a4a5`, which is encoded using the above method, it will be decoded as `abbbaaaaaaaaaaa`.

Write a **C program**, `2.c`, that reads such a run-length encoded string from user, decodes it, and outputs the corresponding sequence of symbols.

**IMPORTANT: For this question, you must submit a C program. No mark would be given if you submit a C++ program.**

### Assumptions:

- The input may contain letters, digits, or symbols. There will be no whitespaces in the encoded string.
- The input is always a valid run-length encoded string following the encoding scheme described above. The length of the input string is at least 2 characters and at most 100 characters.

### Sample Test Cases

| 2_1 | |
|---|---|
| Input: | `a0b2a4a5` |
| Output: | `abbbaaaaaaaaaaa` |

| 2_2 | |
|---|---|
| Input: | `*1!7` |
| Output: | `**!!!!!!!!` |

| 2_3 | |
|---|---|
| Input: | `1111111111` |
| Output: | `1111111111` |

## Problem 3: Postfix notation evaluation

Postfix notation presents operands before operators in a mathematical notation. For example, the expression `1 2 + 3 4 + -` in postfix notation is equivalent to $(1+2)-(3+4)$ in conventional notation.

A stack is a collection of elements that is accessed in a **last in, first out** order. It primarily supports two operations:

- **Push**: Append an element to the collection.
- **Pop**: Removes and returns the last element in the collection.

Expressions in postfix notation can be evaluated using a stack with the following procedure:

- Read one token from the expression.
  - If the token is a number, push it to the stack.
  - If the token is an operator, pop two values from the stack, apply the operation and push the result back to the stack.
- Repeat until no more token could be read. The result of the evaluation is at the top of the stack.

Write a **C program**, `3.c`, that evaluate an expression in postfix notation.

**IMPORTANT: For this question, you must submit a C program. No mark would be given if you submit a C++ program.**

### Input

- The program should read an expression in postfix notation from user input. A `=` sign is added to the end of expression to indicate the end of input.
- For this problem, all operands in the expression will be single digit. Therefore, space is eliminated from the input. For instance, $(1+2)-(3+4)$ will be input as `12+34+-=`.

## Output

The program should:

- Implement a dynamic array and use it as a stack, with an initial capacity of 1.
- Read characters one by one from user. Each character is considered as one token.
  - If the token is a single digit, push it to the stack. If the stack is full after the push, expand the stack by doubling its capacity.
  - If the token is an operator, which can either be `+` or `-`, apply the corresponding operation.
  - If the token is `=`. The program should terminate.
- For each token read except `=`, the program should output the current capacity and the content of the stack after processing the token. Refer to the example and the sample test cases below for the exact format.

### Example

Input: `12+34+-=`

| State | Action | Stack | Capacity | Output |
|---|---|---|---|---|
| *initial* | Initialize a stack with a capacity of 1. | $\boxed{\phantom{0}}$ | 1 | *nil* |
| Read `1` | Push $1$ to stack.<br>Stack expanded as it is full. | $\boxed{1}\boxed{\phantom{0}}$ | 2 | `[2] 1` |
| Read `2` | Push $2$ to stack.<br>Stack expanded as it is full. | $\boxed{1}\boxed{2}\boxed{\phantom{0}}\boxed{\phantom{0}}$ | 4 | `[2] 1 2` |
| Read `+` | Pop $2$ and $1$ from stack.<br>Calculates $1 + 2 = 3$, push $3$ to stack. | $\boxed{3}\boxed{\phantom{0}}\boxed{\phantom{0}}\boxed{\phantom{0}}$ | 4 | `[4] 3` |
| Read `3` | Push $3$ to stack. | $\boxed{3}\boxed{3}\boxed{\phantom{0}}\boxed{\phantom{0}}$ | 4 | `[4] 3 3` |
| Read `4` | Push $4$ to stack. | $\boxed{3}\boxed{3}\boxed{4}\boxed{\phantom{0}}$ | 4 | `[4] 3 3 4` |
| Read `+` | Pop $4$ and $3$ from stack.<br>Calculates $3 + 4 = 7$, push $7$ to stack. | $\boxed{3}\boxed{7}\boxed{\phantom{0}}\boxed{\phantom{0}}$ | 4 | `[4] 3 7` |
| Read `-` | Pop $7$ and $3$ from stack.<br>Calculates $3 - 7 = -4$, push $-4$ to stack. | $\boxed{-4}\boxed{\phantom{0}}\boxed{\phantom{0}}\boxed{\phantom{0}}$ | 4 | `[4] -4` |
| Read `=` | Program terminates. | | | |

### Notes on memory allocation:

Include `stdlib.h` for memory allocation in C.

```c
#include <stdlib.h>
```

To implement a dynamic array in C, we can use `malloc()` to allocate the memory needed to a pointer. We use `sizeof()` to calculate the size of the memory. Note that since `malloc()` returns a generic `void *` type, it is necessary to cast it to a suitable type.

```c
int n = 1; // capacity of array
int * dynArr = (int *)malloc(n * sizeof(int)); // allocate memory
```

Note the values in the allocated array is not initialized. In some systems, the values are initially zero, but it is never a guarantee. You must initialize the values before using them.

To expand the dynamic array, we can use `realloc()` to reallocate the memory needed.

```c
dynArr = (int *)realloc(dynArr, n * sizeof(int)); // expand dynArr to store n integers
```

Finally, when the array is not needed anymore, we can free the allocated memory with `free()`.

```c
free(dynArr);
```

## Assumptions:

- The input will only contain a single line.
- Tokens can only be one of the following: `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`, `+`, `-`, `=`.
- The last token in the input will always be `=`, all other tokens will not be `=`.
- Input expression will always be valid. The program will not receive a `+` or `-` when the stack is empty. The program may receive a `=` when the stack has one and only one element.
- There is no length limit for the input. There could be as many token as that the VPL can handle.

## Sample Test Cases (See the example above for a detailed explanation)

**3_1**

Input:

```
12+34+-=
```

Output:

```
[2] 1
[4] 1 2
[4] 3
[4] 3 3
[4] 3 3 4
[4] 3 7
[4] -4
```

**3_2**

Input:

```
7=
```

Output:

```
[2] 7
```

**3_3**

Input:

```
1234567890+-+-+-+-+=
```

Output:

```
[2] 1
[4] 1 2
[4] 1 2 3
[8] 1 2 3 4
[8] 1 2 3 4
[8] 1 2 3 4 5
[8] 1 2 3 4 5 6
[8] 1 2 3 4 5 6 7
[16] 1 2 3 4 5 6 7 8
[16] 1 2 3 4 5 6 7 8 9
[16] 1 2 3 4 5 6 7 8 9 0
[16] 1 2 3 4 5 6 7 8 9
[16] 1 2 3 4 5 6 7 -1
[16] 1 2 3 4 5 6 6
[16] 1 2 3 4 5 0
[16] 1 2 3 4 5
[16] 1 2 3 -1
[16] 1 2 2
[16] 1 0
[16] 1
```