Programmering for computerteknologi
Hand-in Assignment Exercises

Week 7: Programming larger software projects

Please submit your solutions by the **Monday** after the autumn break(2025-10-20).

At the beginning of each question, it is described what kind of answer you are expected to submit. If Text answer AND Code answer is stated, then you need to submit BOTH some argumentation/description and some code; if just Text answer OR Code answer, then just some argumentation/description OR code. The final answer to the answers requiring text should be one pdf document with one answer for each text question (or text and code question). Make sure that you have committed your code solutions to your GitHub Repository.

Note: the **Challenge** or **Optional** exercises are *optional*, the others mandatory (i.e. you **have** to hand them in).

Exercises

1)

Code answer In the lecture, we discussed how we could implement an approximate sine function using the Taylor series equation:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$$

(a) Write this function in a C program so that it calculates the sine function with precision up to n Taylor series terms, e.g. the above example shows 4 terms. Your sine function should accept both the x value and n precision as input. The signature of your function should be:

```c
double taylor_sine(double x, int n) {
  // implement your function here
  return -1.0;
}
```

Write your Taylor series implementation of sine as a **library** consisting of: a header file and a source file (with no **main()** function).

(b) Write some tests for different values of x (try both small and large input values), and compare your function output with the ANSI C sin function.

Info

To use the ANSI C sin function do the following:

1. Include the `math.h` header file.

```c
#include <math.h>

int main() {
  double pi = 3.14;
  printf("sin(pi) = %f\n", sin(pi));

  return 0;
}
```

2. Link against `math.a`
   - If you are using `gcc`/`clang` in the terminal do `gcc <sourcefiles> -lm` or `clang <sourcefiles> -lm`
   - If you are using the "Run" button in VSCode see the guide from **assignment-4**.

Write your test program (which will have a **main()** function) separately from your library, i.e. **create a new file** where you only include the library header file. Compile your test program by *linking* with your Taylor Sine library.

(c) Answer the following questions using your test program, and please **provide your answers as comments in your test program**: Which intervals of input x did your function give a similar result to the ANSI C sin function? What impact did increasing the precision have (i.e. increasing the number of Taylor series terms)?

2)

Code answer Stacks are containers where items are retrieved according to the order of insertion, independent of content. Stacks maintain *last-in, first-out order (LIFO)*. The abstract operations on a stack include:

Function	**Description**
`push(x, s)`	Insert item `x` at the top of stack `s`.
`pop(s)`	Return (and remove) the top item of stack s
`initialize(s)`	Create an empty stack
`full(s)`, `empty(s)`	Test whether the stack can accept more pushes or pops, respectively. *There is a trick to this, see if you can spot it!*

Note that there is no element search operation defined on standard stacks. Defining these abstract operations enables us to build a stack module to use and reuse without knowing the details of the implementation. The easiest implementation uses an array with an index variable to represent the top of the stack. An alternative implementation, using linked lists, is better because it can't overflow.

Stacks naturally model piles of objects, such as dinner plates. After a new plate is washed, it is pushed on the top of the stack. When someone is hungry, a clean plate is popped off the top. A stack is an appropriate data structure for this task since the plates don't care which one is used next. Thus one important application of stacks is whenever **order doesn't matter**, because stacks are particularly simple containers to implement.

(a) Implement a stack based on **singly-linked lists** as discussed in the lecture.

(b) Test your implementation. **Create a new file**, where you include your stack library header file. You should expect the following "laws" to hold for any implementation of a stack. **Hint:** you should enforce these conditions using `assert` statements:

   (1) After executing `initialize(s);` the stack `s` must be empty.

   (2) After executing `push(x, s); y = pop(s);` the `s` must be the same as before execution of the two commands, and `x` must equal `y`.

   (3) After executing `push(x0, s); push(x1, s); y0 = pop(s); y1 = pop(s);` the stack `s` must be the same as before execution of the four commands, `x0` must equal `y1`, and `x1` must equal `y0`.

> 💡 **Remark**
>
> Stack order is important in processing any properly nested structure. This includes parenthesised formulas (push when you see a "(" token, pop when you see a ")" token), recursive program calls (push on a procedure entry, pop on a procedure exit — we will be discussing **recursion** in Lecture 9), and depth-first traversals of graphs (push on discovering a vertex, pop on leaving it for the last time).

3) **Optional**

> *i* **Info**
>
> This exercise is not a challenge exercise, but it is still just **optional**. We have included it to give you the opportunity to practice working with pointers if you would like.

Create your own string library, with several of tests for each function (in a separate test program). Your library must include functions for the following tasks:

(a) Calculate the length of a string.

(b) Find the next occurrence of a character starting from a given index, e.g. the next 'o' in the string "Hello World" from index 0 is at index 4. The next 'o' in the string "Hello World" from index 5 is at index 7.

(c) Count the number of occurrences of a given character, e.g. the character 'o' in the string "Hello World" occurs 2 times.

(d) Implement a substring function that takes a string as input, a start index and an end index, and returns a string, e.g. substring of "Hello World" from 1 to 4 is "ello".

(e) Implement a tokenize function that takes a string as input and returns a list of strings by splitting the input string into substrings that are separated by the space character (' '), e.g. giving "Hello World" to your tokenize function should return two strings: "Hello" and "World". You can decide what the list return type is, e.g. you could use an array of strings, or your own linked list where the value of each node is a string.

> 🔥 **Tip**
>
> One way to implement this could be to use your *find next occurrence* function and *substring* function within a for loop.

> 💡 **Remark**
>
> This is called *string tokenisation*: splitting a string into a set of sub-strings according to a delimiting character. In general, the delimiter does not necessarily need to be ' ' (space), other common delimiters are ',' (comma) and ';' (semi-colon).

4) **Challenge**

Shoemaker's Problem

A shoemaker has N orders from customers that must be satisfied. The shoemaker can work on only one job each day, and jobs usually take several days. For the i^th job, the integer $T_i (1 \leq T_i \leq 1000)$ denotes the number of days it takes the shoemaker to finish the job. But popularity has its price. For each day of delay before starting to work on the i^th job, the shoemaker has agreed to pay a fine of $S_i (1 \leq S_i \leq 10000)$ cents per day. Help the shoemaker by writing a program to find a sequence of jobs with a minimum total fine.

Input

The input begins with a single positive integer on a line by itself indicating the number of cases, followed by a blank line. There is also a blank line between two consecutive cases. The first line of each case contains an integer reporting the number of jobs N, where $1 \leq N \leq 1000$. The *ith* subsequent line contains the completion time T_i and daily penalty S_i for the i^th job.

Output

For each case, your program should print the sequence of jobs with minimal fine. Each job should be represented by its position in the input. All integers should be placed on only one output line, and each pair separated by one space. If multiple solutions are possible, print the first one in *lexicographic order*. The output of two consecutive cases must be separated by a blank line.

Example

```
4
3 4
1 1000
2 2
5 5
```

```
2 1 3 4
```