Change Log

We may make minor changes to the spec to address/clarify some outstanding issues. These may require minimal changes in your design/code, if at all. Students are strongly encouraged to check the change log regularly.

18 September

- Linked list ADT added to admissible ADTs.

16 September

- Images modified to make clear which edges are included in a partial order graph.
- Requirements for stages 1 & 2 clarified.

Version 1: Released on 14 September 2018

Objectives

The assignment aims to give you more independent, self-directed practice with

- advanced data structures, especially graphs
- graph algorithms
- asymptotic r

Admin

Marks	2 marks for stage 1 (correctness) 3 marks for stage 2 (correctness) 3 marks for stage 3 (correctness) 4 marks for stage 4 (correctness) 2 marks for complexity analysis 1 mark for style ——————— Total: 15 marks
Due	23:59 on **Monday** 8 October (week 11)
Late	2.25 marks (15%) off the ceiling per day late (e.g. if you are 25 hours late, your maximum possible mark is 10.5)

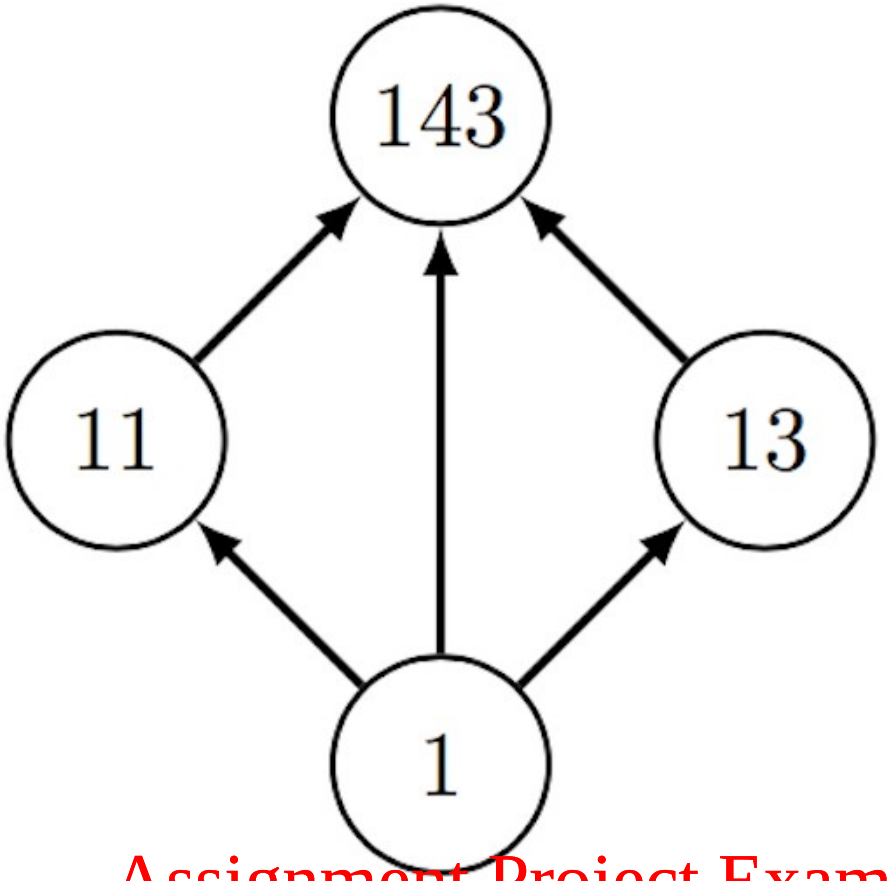
Background

A **partially ordered set** ("poset") is a set S together with a partial order $\preccurlyeq$ on the elements from *S*.

A **partial order graph** for a finite poset $(S,\preccurlyeq)$ is a directed graph ("digraph") with

- the elements in S as vertices
- a directional edge from s to t if, and only if, $s \preccurlyeq t$ and $s \neq t$

Example:



where

- S = {1, 11, 1
- s ≼ t iff s is a di

A **monotonically increasing sequence** of length *k* quence of elements from S,

$$s_1 < s_2 < \ldots s_{k-1} < s_k$$

such that $s_i \preccurlyeq s_{i+1}$ and $s_i \neq s_{i+1}$, for all i=1…*k*-1. Examples:

- 1 ≺ 11 ≺ 143 and 1 ≺ 13 ≺ 143 are monotonically increasing sequences of length 3 over the poset from above.
- 1 ≺ 143 121 is a monotonically increasing sequence of length 2 over this poset.

Aim

Your task is to write a program `poG.c` for computing a partial order graph from a given specification and then find and output *all longest* monotonically increasing sequences that can be constructed over this poset.

Your program should:

- accept a single positive number *p* on the command line;
- compute the set S_p of all (positive) divisors of *p*;
- **Task A:**
  - build and output the partial order graph over S_p corresponding to a specific partial order (see below);
- **Task B:**
  - output all longest monotonically increasing sequences over this partial order.

Your program should include a time complexity analysis, in Big-Oh notation, for

1. your implementation for Task A, depending on the number *n* of divisors of *p* and the length *m* of the decimal *p*;
2. your implementation for Task B, depending on the number *n* of divisors of *p*.

Hints

You may assume that

- the command line argument is correct (a number $p \geq 1$);
- *p* is at most 2,147,483,647 (the maximum 4-byte `int`);
- *p* will have no more than 1000 divisors.

If you find any of the following ADTs from the lectures useful, then you can, and indeed are encouraged to, use them with your program:

- stack ADT : `stack.h`, `stack.c`
- queue ADT : `queue.h`, `queue.c`
- list ADT : `list.h`, `list.c`
- graph ADT : `Graph.h`, `Graph.c`
- weighted graph ADT : `WGraph.h`, `WGraph.c`

You are free to modify any of the four ADTs for the purpose of the assignment (but without changing the file names). If your program is using one or more of these ADTs, you should submit both the header and implementation file, even if you have not changed them.

Your main program file should start with a comment: `/* ... */` that contains the time complexity of your solutions for T

Stage 1 (2 marks)

For stage 1, you should demonstrate that you can b h correctly from all divisors of input *p* and the following partial order:

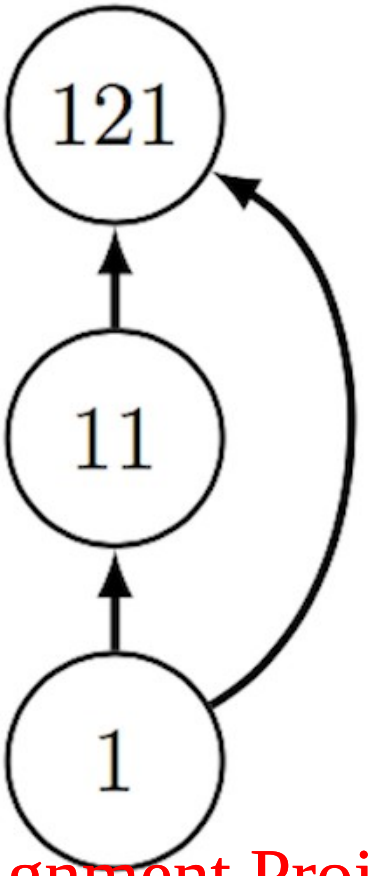
$$x \preccurlyeq y \text{ iff } x \text{ is a divisor of } y$$

All you need to do for Task B at this stage is to output all nodes of the graph in ascending order.

Here is an example to show the desired behaviour of your program for a stage 1 test:

```
./poG 121
Partial order:
1: 11 121
11: 121
121:

Longest monotonically increasing sequences:
1 < 11 < 121
```



Hint: The only tests fo rder is identical to the stricter partial order for stage a monotonically increasing sequence.

Stage 2 (3 marks)

For stage 2, you should extend your program for stage 1 such that it puts an additional constraint on the partial order of the divisors of *p*:

x ≼ y iff

- x is a divisor of y, **and**
- all digits in x also occur in y

For example, 11 ≺ 143 since 1 is also contained in 143, but 16 ⊀ 128 since 6 is not a digit in 128.

All you need to do for Task B at this stage is to find and oputput the path that starts in 1 and always selects the next neighbour in ascending order until you reach a node without outgoing edge.

Here is an example to show the desired behaviour of your program for a stage 2 test:

```
./poG 9481
Partial order:
1: 19 9481
19: 9481
499: 9481
9481:
```

```
Longest monotonically increasing sequences:
1 < 19 < 9481
```


Hint: All tests for this stage will be such that the only lon ing sequence is the unique path from 1 to *p* obtained by always moving to the next n er.

Stage 3 (3 marks)

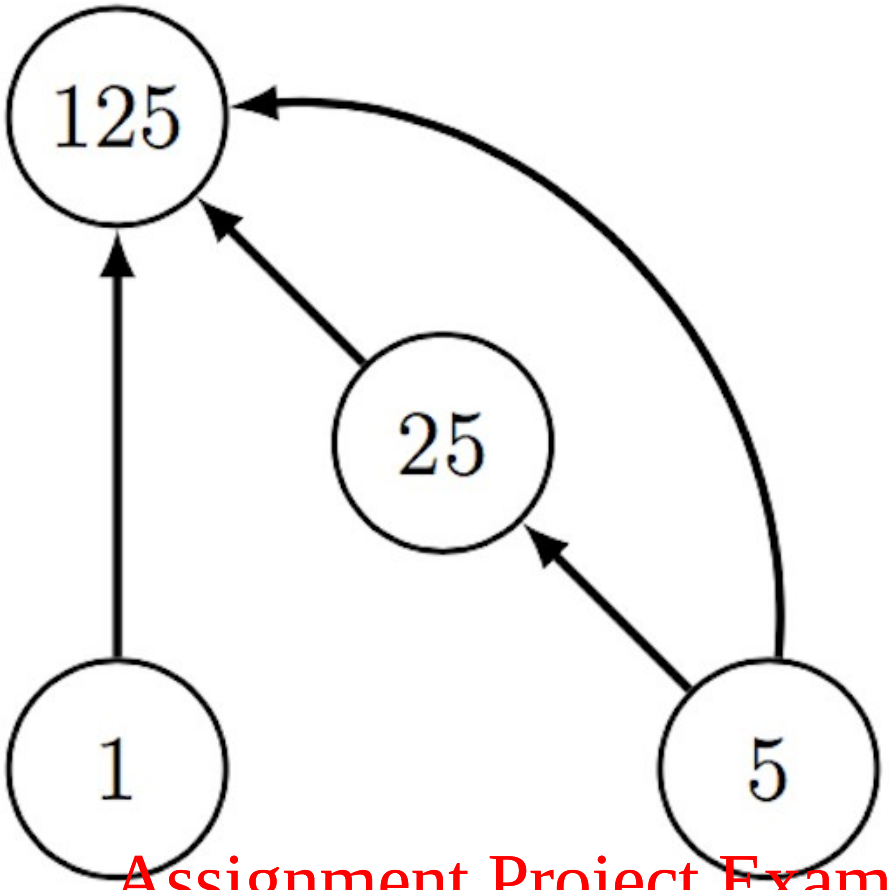
For stage 3, you should demonstrate that you can find a *single* longest monotonically increasing sequence.

All tests for this stage will be such that there is a unique longest sequence.

Here is an example to show the desired behaviour of your program for a stage 3 test:

```
./poG 125
Partial order:
1: 125
5: 25 125
25: 125
125:

Longest monotonically increasing sequences:
5 < 25 < 125
```



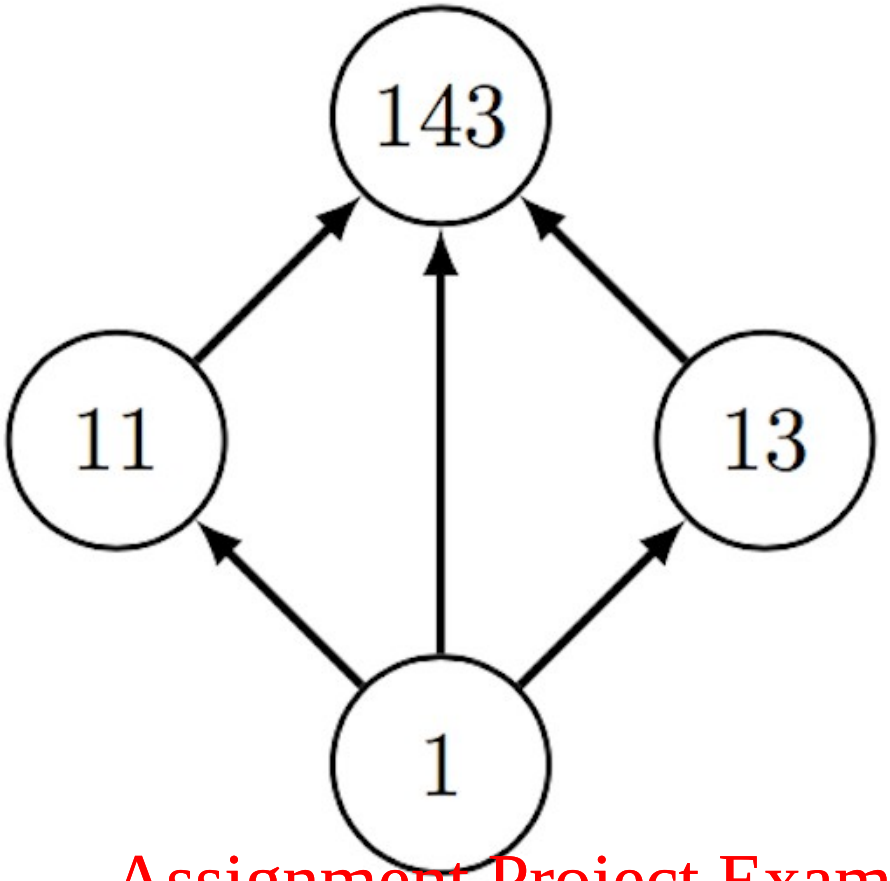
Stage 4 (4 mar

For stage 4, you should extend your program for stage 3 such that it outputs, in ascending order, all monotonically increasing sequences of maximal len

Here is an example to show the desired behaviour age 4 test:

```
./poG 143
Partial order:
1: 11 13 143
11: 143
13: 143
143:

Longest monotonically increasing sequences:
1 < 11 < 143
1 < 13 < 143
```



Note:

- It is required that the sequences be printed i

Testing

We have created a script that can automatically test your program. To run this test you can execute the `dryrun` *program for the corresponding assignment, i.e.* `assn2`*. It expects to find, in the current directory, the program* `poG.c` *and any of the admissible ADTs (*`Graph,WGraph,stack,queue,list`*) that your program is using, even if you use them unchanged. You can use dryrun as follows:*

```
~cs9024/bin/dryrun assn2
```

Please note: Passing the dryrun tests does not guarantee that your program is correct. You should thoroughly test your program with your own test cases.

Submit

For this project you will need to submit a file named `poG.c` and, optionally, any of the ADTs named `Graph,WGraph,stack,queue,list` that your program is using, even if you have not changed them. You can either submit through WebCMS3 or use a command line. For example, if your program uses the `Graph` ADT and the `queue` ADT, then you should submit:

```
give cs9024 assn2 poG.c Graph.h Graph.c queue.h queue.c
```

Do not forget to add the time complexity to your main source code file `poG.c`.

You can submit as many times as you like — later submissions will overwrite earlier ones. You can check that your submission has been received on WebCMS3 or by using the following command:

```
9024 classrun -check assn2
```

Marking

This project will be marked on functionality in the first instance, so it is very important that the output of your program be ***exactly*** correct as shown in the examples above. Submissions which score very low on the automarking will be looked at by a human and may receive a few marks, provided the code is well-structured and commented.

Programs that generate compilation errors will receive a very low mark, no matter what other virtues they may have. In general, a program that attempts a substantial part of the job and does that part correctly will receive more marks than one attempting to do the entire job but with many errors.

Style considerations include: readability, structured programming and good commenting.

Plagiarism

Group submissions will not be allowed. Your program must be entirely your own work. Plagiarism detection software will be used to compare all submissions pairwise (including submissions for similar projects in previous years, if applicable) and serious penalties will be applied, particularly in the case of repeat offences.

- ***Do not copy ideas or code from others***
- ***Do not use e to see your code, not even after t***

Please refer to the on-line sources to help you unde is and how it is dealt with at UNSW:

- Plagiarism and Academic Integrity
- UNSW Plagiarism Policy Statement
- UNSW Plagiarism Procedure

Help

See FAQ for some additional hints.

Finally …

Best of luck and have fun! Michael