1

In simple cases, an array may be created like this:

```
string AAA[1000];
```

but sometimes a declaration in this form is needed:

```
string * BBB;
```

Short Answers:

i. What other step is required before `BBB` can actually be used as an array?

BBB is just an uninitialised variable that *can* point to an array. It must be given an array to point to

BBB = new string[size];

(instead, BBB could be made to point to another existing array:

BBB = AAA;

but that is often a bad thing to do).

ii. What happens if a program does not take that extra step? Perhaps doing something like this:

```
{ string * BBB;
  BBB[17] = "easy";
```

The program attempts to store the string in some arbitrary and unpredictable place in memory. In this case it would most probably cause a segmentation fault and stop the program. In other cases the result is completely unpredictable.

iii. What advantages does the second method (`BBB`) have over the first (`AAA`) - What is it possible to do with `BBB` that can not be done with `AAA`?

A pointer can be made to point somewhere else, so if the array initially created is the wrong size, it can be deleted and a new one made instead. Also, the size specified with new does not need to be an already known constant.

iv. What is a likely cause of a "Segmentation Fault", and what might a sensible programmer do when faced with one?

Something very much like (ii), using an uninitialised or invalid pointer, or using an invalid index into an array.

v. What do the C++ operations `new` and `delete` do - What is their purpose?

"new X" creates an object of type X using heap memory and provides a pointer to that object, "new X[size]" creates a whole array of Xs in the heap. Heap memory is not automatically "recycled" when the variable referring to it ceases to exist (when the function containing it returns), so objects of this sort will continue to exist and be safe for as long as they are needed.

"delete Y" or "delete [] Y" signals that the object(s) pointed to by Y is no longer needed, so the memory occupied may be recycled and used to create other objects when new is used again.

vi. What happens if a program uses `new` frequently, but never uses `delete`?

A "memory leak". The program will eventually run out of heap memory and fail.

vii. What is a constructor?

A method with the same name as the struct/class containing it. A constructor is called automatically every time an object is created. Its job is to perform any initialisation needed before the object is first used. If any constructors for X have been defined, it is impossible to create an object of type X without a suitable constructor being called.

viii. I want to define a simple struct to represent pet mice.
It must have a string to store the mouse's name, and another string for its colour. It must also have a number to record how much I paid for it, and a true/false value to record whether my cat ate it or not. Additionally, the struct must have a suitable constructor.
I am too lazy to do it. Write the complete definition in C++ for me.

```
struct mouse
{ string name, colour;
  int price;
  bool eaten;

  mouse(string n, string c, int p)
  { name = n;
    colour = c;
    price = p;
    eaten = false; } };
```

I am assuming that mice only get eaten *after* then have been initialised/created.

2

You may have heard of the Collatz Sequence before. It is a very simple thing - take any number, and if it is even divide it by two; if it is odd multiply it by three and add one.

if n is even, n = n÷2

if n is odd, n = n×3+1

Repeat that process over and over again until the number is equal to one.

For example, if you start with 6, the sequence is 6, 3, 10, 5, 16, 8, 4, 2, 1. It is impossible to predict how long the sequence will be. Starting from 31, the sequence is 107 steps long, but starting from 32 it is only 6 steps long.

I have attempted to write a function that works out the sequence from any starting point, puts the entire sequence in an array, and returns that array as a result. Unfortunately, I have got the function wrong:

```
int * collatz(int n)
{ int sequence[];
```

That's the big mistake. I didn't know how big the array would need to be, so I just left the size out. That is not allowed. The best correction is to define a struct that implements a vector of ints, and use that. You've seen that in class, so I'll do it the lazy way here

```
int capacity = 100;
int * sequence = new int[capacity];
```

```
int length = 0;
while (n != 1)
{
```

Before trying to put something in the array, I'd better make sure there is enough room for it

```
if (length >= capacity)
{ int * replacement = new int[capacity + 100];
  for (int i = 0; i < length; i += 1)
    replacement[i] = sequence[i];
  delete [] sequence;
  sequence = replacement;
  capacity += 100; }
```

```
  sequence[length] = n;
  length = length+1;
  if (n%2 == 0)
    n = n/2;
  else
    n = n*3+1; }
return sequence; }
```

It is essential that this function should not be *very* wasteful of memory.

At worse, that solution might waste the space taken by 99 ints. I wouldn't call that *very* wasteful. And of course, the +100 could be replaced by a smaller increase.

Correct the function.

3

Reminders: if N is the amount of data in use,
- a Linear algorithm takes time proportional to N,
- a Quadratic algorithm takes time proportional to N^2,
- a Cubic algorithm takes time proportional to N^3,
- an Exponential algorithm takes time proportional to 2^N,

a

Here is a very badly designed sorting algorithm (it doesn't even work)

```
          for (int i=1; i<N; i+=1)
```

The following statements will be executed exactly N-1 times with different values for i

```
          { int p = 0, q = N;
            for (int x=1; x<N; x+=1)
```

for each value of i, this next conditional statement will also be executed N-1 times

```
            { if (A[x] < A[p])
                p = x; }
            for (int y=1; y<N; y+=1)
```

and so will this one

```
            { if (A[y] > A[q])
                q = y; }
            A[p] = A[q];
            A[q] = A[p]; }
```

What kind of algorithm is it (in terms of linear, quadratic, cubic, etc)?
Explain why.

So in total, there are a few little statements that get executed N-1 times, but a statement that looks like "if (A[x] < A[p]) p = x" will be executed $2\times(N-1)\times(N-1)$ - which is $2N^2-4N+2$ - times.

The time is only relevant if it is long enough to notice, which means that N can't be very small. If N is not small, then $2N^2$ will completely outweigh (N-1) or 4N and make them insignificant. So the time it takes will be approximately proportional to N^2 - Quadratic.

b

If a Quadratic algorithm takes 10mS (0.01 seconds) to process 1,000 data items, how long should it take to process:

$\text{time} \approx k\,N^2$

$0.01 \approx k\,(1000)^2$

$10^{-2} \approx k\,10^6$

$k \approx 10^{-2} / 10^6$

$k \approx 10^{-8}$

i. 2,000 items

$\text{time} \approx k\,(2000)^2 = 10^{-8}\times 4\times 10^6 = 4\times 10^{-2} = 0.04$ seconds

ii. 3,000 items

$\text{time} \approx k\,(3000)^2 = 10^{-8}\times 9\times 10^6 = 9\times 10^{-2} = 0.09$ seconds

iii. 1,000,000 items

$\text{time} \approx k\,(1000000)^2 = 10^{-8}\times 10^{12} = 10^4 \approx 3$ hours +

iv. 100 items

$\text{time} \approx k\,(100)^2 = 10^{-8}\times 10^4 = 10^{-4} = 0.0001$ seconds

c

If a Quadratic algorithm takes 1 second to process 1,000,000 data items, and you discover a Linear algorithm that does the same job, roughly how long would your new algorithm take?

$\text{time} \approx k\,N^2$

$1 \approx k\,(1000000)^2$

$1 \approx k\,10^{12}$

$k \approx 10^{-12}$

Assuming that the basic operations performed repeatedly by both algorithms are somewhat similar (likely, but by no means definite), k would not have changed very much, so now we have

$\text{time} \approx k\,N = 10^{-12}\times 10^6 = 10^{-6} = 1$ micro-second

That's the best we can do given the information we've got.

d

If a Cubic algorithm takes 10mS (0.01 seconds) to process 100 data items, how long should it take to process:

i. 1,000 items

by exactly the same process, 10 seconds

ii. 1,000,000 items

and 10,000,000,000 seconds, or 300+ years. Cubic is bad.

e

If an Exponential algorithm takes 1mS (0.001 seconds) to process 10 data items, how long should it take to process:

exponential means time $\approx k\,2^N$

$10^{-3} \approx k\,2^{10}$

and as $2^{10} \approx 10^3$, $k \approx 10^{-6}$

i. 20 items

time $\approx 10^{-6} \times 2^{20} \approx 10^{-6} \times 10^{6} = 1$ second

ii. 40 items

time $\approx 10^{-6} \times 2^{40} \approx 10^{-6} \times 10^{12} = 1{,}000{,}000$ seconds = nearly two weeks.

4

Draw a picture of a cat sitting on a quadratic chair.

```
                    _
                   | |
                   | |
                   | |
      ^ ^          | |
     (. .)         | |
     ==v==         | |
    (     )        | |
    (     )        | |
    ( v v )=====>  | |
--------------------  |
| ------------------  |
| |                | |
| |                | |
| |                | |
| |                | |
| |                | |
| |                | |
```