CSCI 2270
Data Structures and Algorithms
Lecture 7

Elizabeth White
elizabeth.white@colorado.edu
Office hours: ECCS 128
Wed 1-2pm
Thurs 2-3pm

Dynamic memory arrays

Please read Chapter 2 of CLRS (Intro to Algorithms) for this week.

It's very symbol-heavy, and we'll cover the important parts in lecture. Try to focus on the gist of the insertion sorting algorithms, and practice reading the pseudocode for insertion sort.

Resizing the int_array

```
void resize(int_array& arr) {
        // remember the size of the new bigger array
        arr.capacity *= 2;
        // reserve memory for this bigger array and store
        // its starting address in bigger_array
        int* bigger_array = new int[arr.capacity];
        // copy data from small array to big array
        for (unsigned int k = 0; k < arr.count; ++k)
                bigger_array[k] = arr.data[k];
        // release memory for smaller array; now arr.data
        // stores the address of memory we no longer own
        delete [] arr.data;
        // update arr.data to store the address of the
        // bigger array instead of that crummy address
        arr.data = bigger_array;
}
```

Another way to copy

Pointer arithmetic!

To copy array p to array q,

```
copy( starting address in p,
      stopping address in p,
      starting address in q)
```

Copies every entry from the starting address in p to *one entry before* the stopping address in p

Writes these into q starting at the starting address in q

Allows us to copy parts of an array

The 2 arrays can't overlap, though. For that, you would write your own loop.

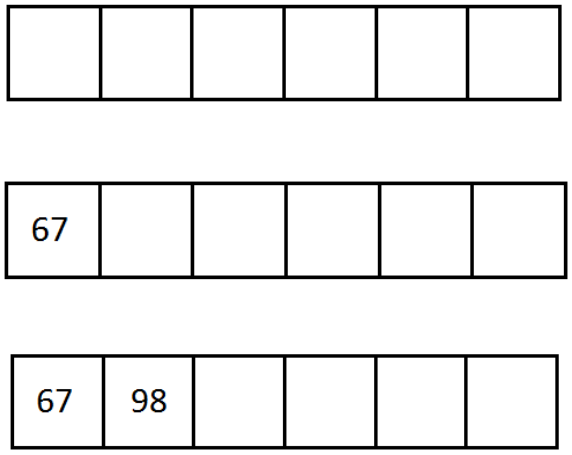
Example of copy

```
#include <algorithm>
using namespace std;
void copy_array()
{
	int* arr1 = new int[10];
	for (unsigned int k = 0; k < 10; ++k)
		arr1[k] = k*k;
	for (unsigned int k = 0; k < 10; ++k)
		cout << arr1[k] << " ";
	cout << endl;
	int* arr2 = new int[20];
	copy(arr1, arr1 + 10, arr2);
	copy(arr1, arr1 + 10, arr2 + 10);
	for (unsigned int k = 0; k < 10; ++k)
		cout << arr1[k] << " ";
	cout << endl;
}
```

Adding to the int_array

```
void add(int_array& arr, const int& payload)
```

1. Check if array is full and call resize if so
2. Add payload to next open slot in array
3. Update count

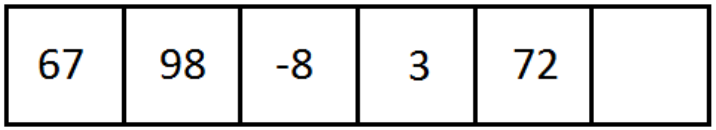


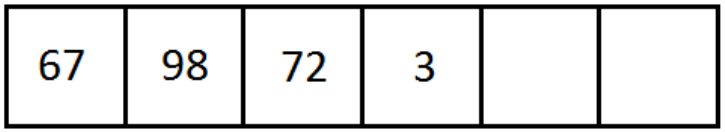

```
struct int_array {
        int* data;
        unsigned int count;
        unsigned int capacity;
        static const unsigned int
DEFAULT_CAPACITY = 20;
        };
```

Removing from the int_array

```
void remove(int_array& arr, const int& target)
```

1. Find index of target integer to remove, if it's there (return false if it's not).
2. If target was found in data array, move last integer stored in data array over the integer in target's slot.
3. Update count.






```
struct int_array {
        int* data;
        unsigned int count;
        unsigned int capacity;
        static const unsigned int
DEFAULT_CAPACITY = 20;
        };
```

Searching the int_array

```
void contains(int_array& arr, const int& target)
```

1. Find index of target integer, if it's in the array
2. If the target is present at least once, return true.
3. Otherwise, return false.

```
struct int_array {
        int* data;
        unsigned int count;
        unsigned int capacity;
        static const unsigned int
DEFAULT_CAPACITY = 20;
        };
```

Bug Hunting

Most problems come from reading or writing outside the bounds of the array.

Check that you can:

Add more than 20 integers to the array

Find integers at the start, end, and middle of the array

Not find integers that aren't in the array

Remove integers from the start, end, and middle of the array

Clear an array and add more elements to it

Finish the program with no crashes at the end

Next week: make the array sorted

Need to change add and remove; nothing else

```
struct int_array {
        int* data;
        unsigned int count;
        unsigned int capacity;
        static const unsigned int
DEFAULT_CAPACITY = 20;
        };
```

Algorithms

From chapter 1 of CLRS

- Algorithms convert data inputs to answer outputs
- Algorithms are recipes for solving problems

Algorithms come with speed limits on performance

Example: lab this week, unsorted array of integers.

- adding an integer to the data is fast (unless we resize)
  - find data[count],
  - increase count (2 operations)
- removing an integer is slower, because we have to find it
  - this takes time proportional to the count

Algorithms

Example: lab next week, sorted array.

Speed of algorithm changes:

adding can be slower

removing is still ‘about the same’

contains can get much, much faster

This will give you your first example of an algorithm’s performance depending on the data structure it’s using.

Some problems, like the traveling salesman, have no good algorithmic solutions as they get larger.

(If you invent one, you’ll be famous or rich.)

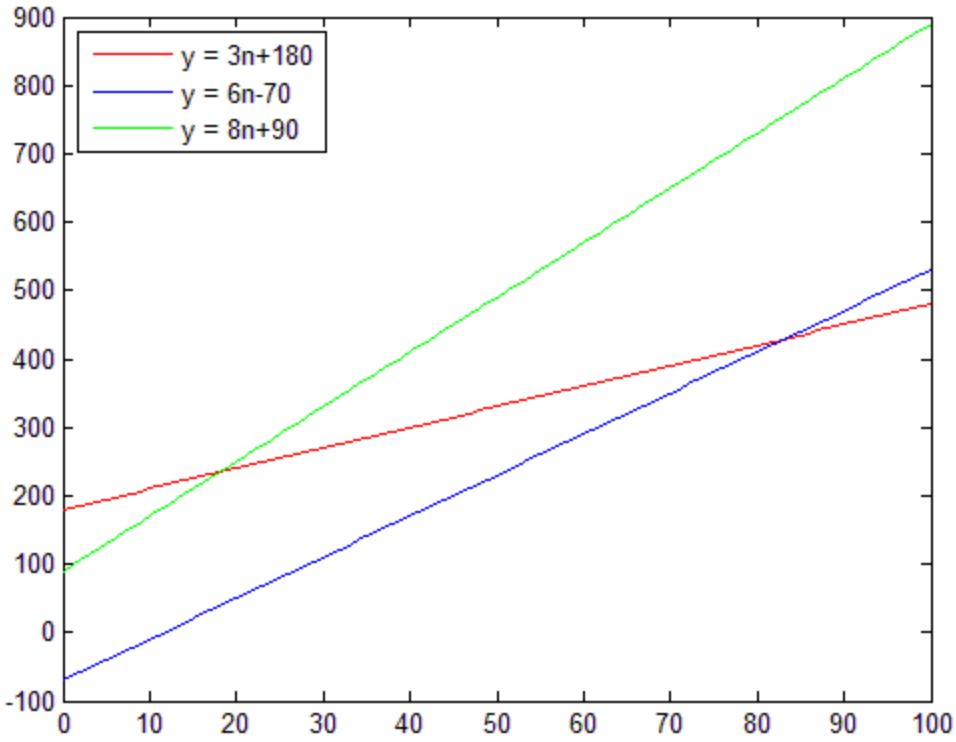
Speed of a linear algorithm

A linear function of n is a straight line:

y = cn + d, where c and d are constants

Most of the time, we won't care about the slope m, or the intercept b.

So all of these lines are linear order, or O(n), to us.



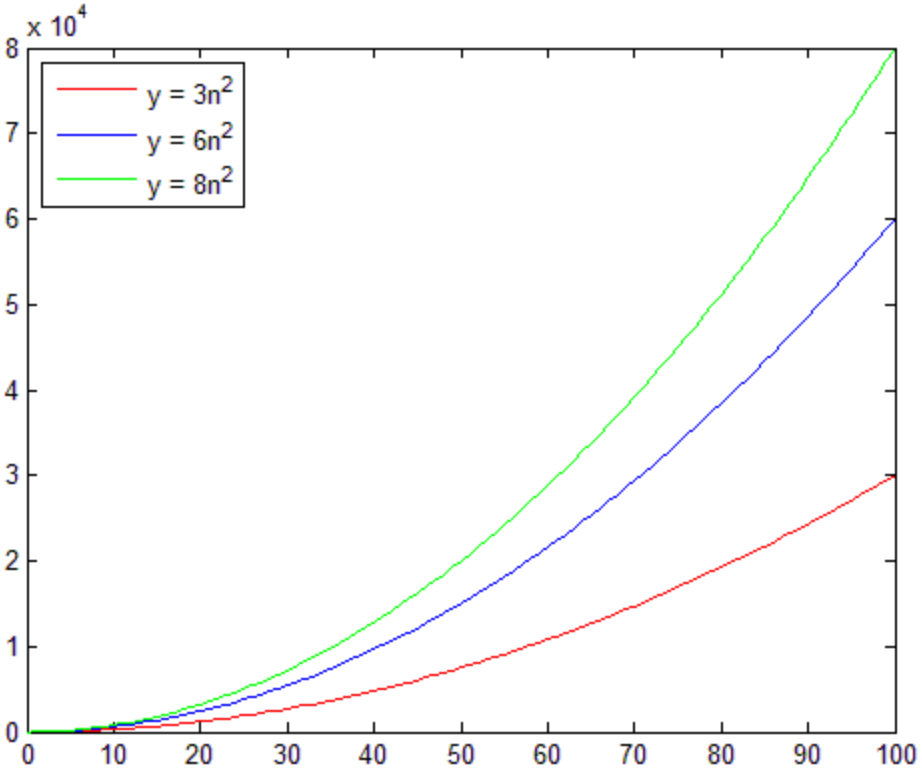
Speed of a quadratic algorithm

A quadratic function of n is a parabola:

$y = cn^2$ for some constant c.

Most of the time, we won't care about the coefficient c in front; we only care that the exponent of n in this case is 2.

All of these curves are $O(n^2)$ to us.



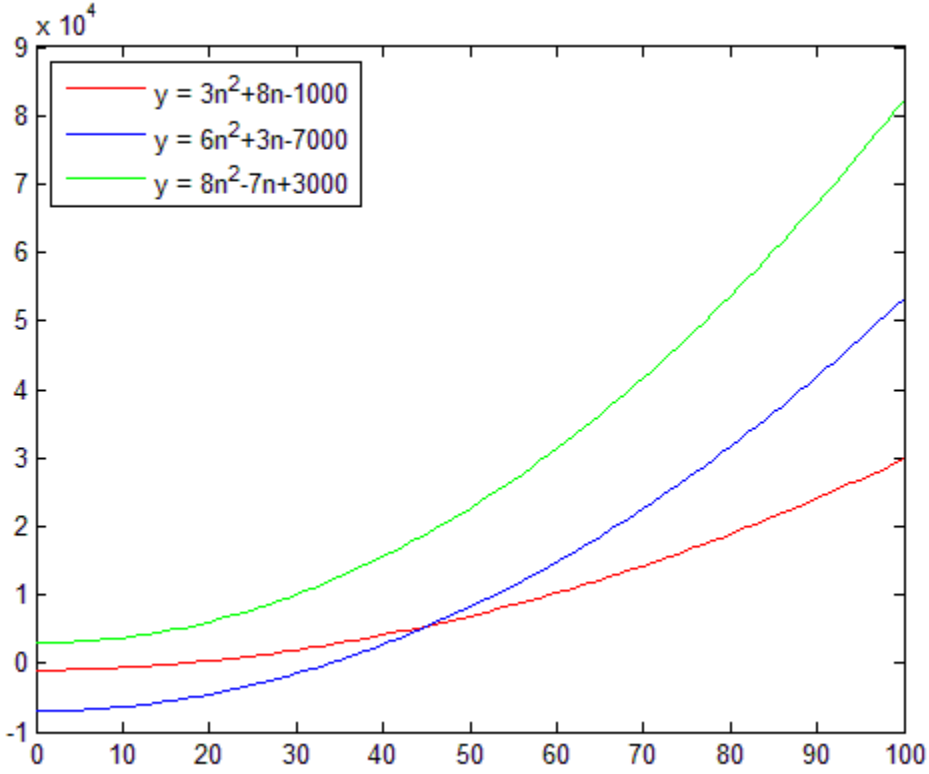
Speed of a quadratic algorithm

A quadratic function of n can have lower-order terms:

$y = cn^2 + dn + e$ for some constants c, d, and e.

But we also don't care about lower-order terms involving n.

So a function like $y = 3n^2 + 8n - 1000$ is still just $O(n^2)$.



Speed of a logarithmic algorithm

A logarithmic function of n grows more slowly:

y = c*log(n) + d for some constants c and d.

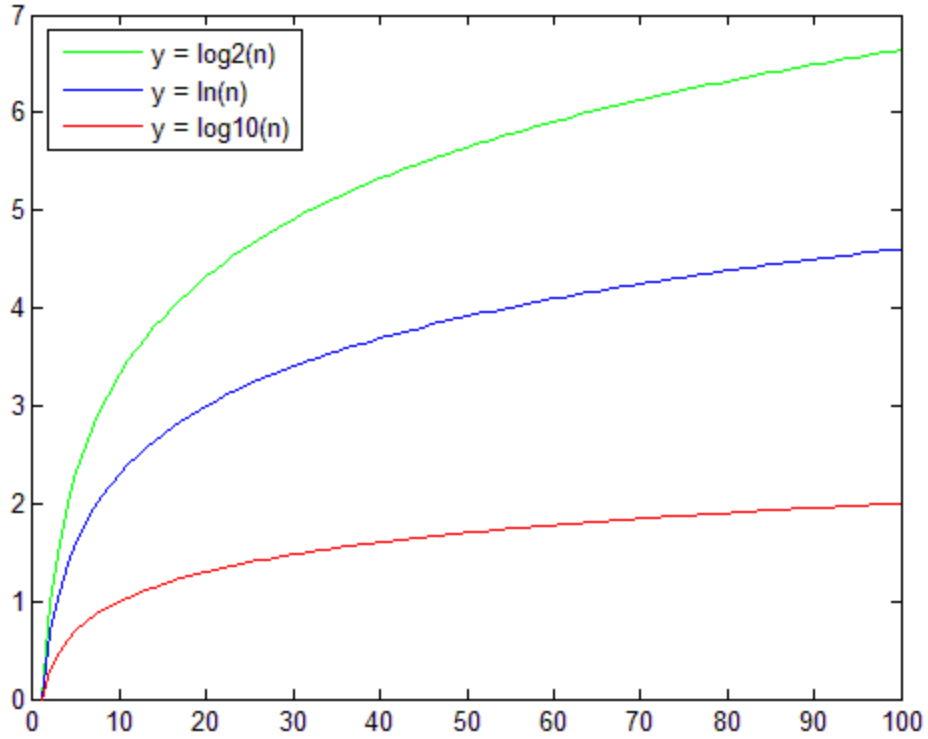
Here, we mostly don't care about the c and d terms.

We also don't worry about the kind of logarithm, so y = log10(n),

y = ln(n), and

y = log2(n)

are all O(lg n) to us.



Growth of functions

As n grows,

$$O(1) < O(\lg n) < O(n^{1/2}) < O(n) < O(n \lg n) < O(n^2) < \ldots$$
$$O(n^3) < \ldots < O(2^n) < O(n!)$$

Some expected times for different algorithms:

add to unsorted array:	O(1)
find an item in an unsorted array:	O(n)
solve travelling salesman:	O(n!)
find an item in a sorted array:	O(lg n)