Welcome!

COMP1511 18s1
Programming Fundamentals
Don’t panic!

assignment 3 out now!
week 11’s tute/lab help you get started

week 11
lab solutions now out weekly test due friday
don’t forget about help sessions!
see course website for details

there will be lectures next week!
(week 13)
Questions?

unanswered questions?

ask on Ed

edstem.org/courses/1950/

https://echo360.org.au/

note: you may need to go via Moodle

https://moodle.telt.unsw.edu.au

(let me know if you can/can't access it!)
What topics are you confused about? What questions do you have?

What is your response?
COMP1511 gets you thinking like a programmer solving problems by developing programs expressing your ideas in the C language.
future COMP courses
(e.g. COMP2521)

gets you thinking like a computer scientist
knowing fundamental techniques/structures
able to reason about applicability/effectiveness
able to analyse behaviour/correctness of programs
COMP1511 vs future COMP courses

for now... just a taster

(you’ll have to take COMP2521 for more!)
introducing: **analysis**

putting the **science** in **computer science**

for when “it works!” isn’t good enough
what makes software good?
what makes software good?

correctness?
what makes software **good**?

correctness?

efficiency?
what makes software good?

correctness?

efficiency?

clear, maintainable code?
what makes software **good**?

correctness?

efficiency?

clear, maintainable code?

**usability**?
today: efficiency
COMP1511 focuses on writing correct programs but efficiency is also important often need to consider:

- execution time
- memory use

a correct but too slow program can be useless
Efficiency

Efficiency often depends on the size of the data being processed. Understanding this dependency lets us predict program performance on larger data.

....

Informal exploration in COMP1511 - much more in COMP2521 and COMP3121
how can we find out whether a program is efficient or not?

**empirical approach** - run the program several times with different input sizes and measure the time taken

**theoretical approach** - try to count the number of `operations`` performed by the algorithm on input of size \( n \)
```c
int linear_search(int array[], int length, int x) {
    for (int i = 0; i < length; i = i + 1) {
        if (array[i] == x) {
            return 1;
        }
    }
    return 0;
}
```
Linear Search Unordered Array - Informal Analysis

Operations:

- start at first element
- inspect each element in turn
- stop when find X or reach end

If there are N elements to search:

- best case check 1 element
- worst case check N elements
  
if in list on average check N/2 elements
if not in list check N elements
int linear_ordered(int array[], int length, int x) {
    for (int i = 0; i < length; i = i + 1) {
        if (array[i] == x) {
            return 1;
        } else if (array[i] > x) {
            return 0;
        }
    }
    return 0;
}
Linear Search Ordered Array - Informal Analysis

Operations:

- start at first element
- inspect each element in turn
- stop when find \( X \) or find value \( X \) or reach end

If there are \( N \) elements to search:

- best case check 1 element
- worst case check \( N \) elements
- if in list on average check \( N/2 \) elements
- if not in list on average check \( N/2 \) elements
```c
int binary_search(int array[], int length, int x) {
    int lower = 0;
    int upper = length - 1;
    while (lower <= upper) {
        int mid = (lower + upper) / 2;
        if (array[mid] == x) {
            return 1;
        } else if (array[mid] > x) {
            upper = mid - 1;
        } else {
            lower = mid + 1;
        }
    }
    return 0;
}
```
Operations:

- start with entire array
- at each step halve the range the element may be in
- stop when find X or range is empty

If there are N elements to search

- best case check 1 element
- worst case check log2(N)+1 elements
- if in list on average check ~log2(N) elements
Binary Search Ordered Array - Informal Analysis

log₂(N) grows very slowly:

- log₂(10) = 3.3
- log₂(1000) = ~10
- log₂(1000000) = ~20
- log₂(1000000000) = ~30
- log₂(1000000000000) = ~40

Physicists estimate $10^{80}$ atoms in universe: $\log₂(10^{80}) = 240\,[\text{ex}]$

Binary search all atoms in universe in < 1 microsecond
let’s look at:

sorting
Sorting

**sort**: rearrange a sequence so it is in non-decreasing order

**why?**
- sorted sequence can be searched efficiently
- items with equal keys are located together

**why not?**
- simple obvious algorithms too slow to sort large sequences
  (better algorithms can sort very large sequences)
there are many different sorting algorithms

we'll look at one slow obvious algorithm:

bubblesort

and at one fast algorithm:

quicksort

(SortVis: https://sorting.alhinds.com)
Bubblesort

go through the array, comparing pairs of elements

swap the elements if they’re in the wrong order

... repeat until sorted
// our array of numbers
3  1  4  9  5

// compare the first pair
[3] [1]  4  9  5

// they're in the wrong order, so swap
[1] [3]  4  9  5

// compare the second pair

// compare the third pair
1  3  [4] [9]  5

// compare the fourth pair
1  3  4  [9] [5]

// they're in the wrong order, so swap
1  3  4  [5] [9]
```c
void bubblesort(int array[], int length) {
    int swapped = 1;
    while (swapped) {
        swapped = 0;
        for (int i = 1; i < length; i = i + 1) {
            if (array[i] < array[i - 1]) {
                int tmp = array[i];
                array[i] = array[i - 1];
                array[i - 1] = tmp;
                swapped = 1;
            }
        }
    }
}
```
Quicksort

faster than bubblesort

**divide and conquer**

(make the problem smaller each time)

works by dividing the array into two smaller arrays
then sorting the two smaller arrays

...

it does this by choosing a **pivot**
then moving all of the **smaller** elements to its left and
all of the **larger** elements to its right
void quicksort(int array[], int length) {
    quicksort1(array, 0, length - 1);
}

void quicksort1(int array[], int lo, int hi) {
    if (lo >= hi) {
        return;
    }
    int p = partition(array, lo, hi);
    // sort lower part of array
    quicksort1(array, lo, p);
    // sort upper part of array
    quicksort1(array, p + 1, hi);
}
```c
int partition(int array[], int lo, int hi) {
    int i = lo, j = hi;
    int pivotValue = array[(lo + hi) / 2];
    while (1) {
        while (array[i] < pivotValue) {
            i = i + 1;
        }
        while (array[j] > pivotValue) {
            j = j - 1;
        }
        if (i >= j) {
            return j;
        }
        int temp = array[i];
        array[i] = array[j];
        array[j] = temp;
        i = i + 1;
        j = j - 1;
    }
    return j;
}
```
If we instrument quicksort and bubble sort code, we see:

- bubblesort is proportional to $n^2$
- quicksort is proportional to $n \log n$

if $n$ is small, little difference

if $n$ is large, huge difference

for large $n$, you need a good sorting algorithm like quicksort