Efficiency

- COMP1511 focuses on writing programs.
- Effciency is also important. Often need to consider:
  - execution time
  - memory use.
- A **correct** but slow program can be useless.
- Efficiency often depends on the size of the data being processed.
- Understanding this dependancy lets us predict program performance on larger data
- Informal exploration in COMP1511 - much more in COMP2521 and COMP3121

Analysis of Algorithms

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

Linear Search Unordered Array - Code

```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

Linear Search Ordered Array - Code

```
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

Binary Search Ordered Array - Code

```
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;
}
```

Binary Search Ordered Array - Informal Analysis

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:

- $\log 2(10) = 3.3$
- $\log 2(1000) = 10$
- $\log 2(1000000) = 20$
- $\log 2(1000000000) = 30$
- $\log 2(1000000000000) = 40$

Physicists estimate 10^{80} atoms in universe: $log2(10^{80}) = 240$

Binary search all atoms in universe in < 1 microsecond

Sorting

- Aim: rearrange a sequence so it is in non-decreasing order
- Advantages
  - sorted sequence can be searched efficiently
  - items with equal keys are located together
- The problem of sorting
  - simple obvious algorithms too slow to sort large sequences
  - better algorithms can sort very large sequences
- sorting extensively studied and many algorithms proposed.
- We'll look at one slow obvious algorithm: **bubblesort**
- And at one fast algorithm: **quicksort**
- We'll assume sorting array of ints.
- Straight-forward to extend code to handle other types of items (e.g. strings) and other data structures.

Bubblesort - Code

```
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;
            }
        }
    }
}
```

Bubblesort - Code Inst

```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 - Code

```
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);
}
```

Quicksort Partition - Code

```
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;
```

Quicksort and Bubblesort Compared

If we instrument quicksort and bubble sort code, we see:

Array size (**n**)	bubblesort operations	quicksort operations
10	81	24
100	8415	457
1000	981018	9351
10000	98790120	102807

- bubblesort is proportional to $\mathbf{n}^2$
- quicksort is proportional to $\mathbf{n} log_2(\mathbf{n})$
- if **n** is small, little difference
- if **n** is large, huge difference
- for large **n**, you need a good sorting algorithm like quicksort