CM 10227: Lecture 11

Dr. Rachid Hourizi and Dr. Michael Wright

December 13, 2016

Resources

- More help with this course
  - Moodle
  - E-mail - programming1@lists.bath.ac.uk
- Online C and Java IDE
  - https://www.codechef.com/ide
  - Remember to select Java from the drop down menu.
- Books
  - Java by Dissection (Free pdf online)
  - The Java Tutorial (https://docs.oracle.com/javase/tutorial/)

Resources

- The places that you can get additional support if you are finding the pace of the course a little fast now include
  - A labs (Continued from week 1)
  - B labs
  - ... Wednesday 11:15-13:05 EB0.7
  - ... Fridays 17:15 to 19:15 in CB 5.13)
  - The Drop in Session
  - ... booked 20 min appointments
  - ... Friday 11.15-13.05 1E 3.9
  - PAL sessions (Mondays 14:15 to 15:05 1E 3.9)

Last Week

- Java API Libraries

This Week

- Complexity

Complexity

- To this point in the course, we have, on a number of occasions talked about algorithms, pseudo code and implementation (more detailed definitions of each one will follow, below)
- We have also discussed the fact that most if not all programming problems can be approached in many different ways
- i.e. that different algorithms can be employed to address the challenges of a given programming task

- That different programmers will take different approaches to developing pseudo-code when fleshing out the details of a given algorithm ...
- ... and that even quite detailed pseudo code leaves room for different programmers to make different design decisions during implementation
- We have not, yet, however talked about the ways in which we might compare algorithms (i.e. decide whether one is 'better' or another 'worse')

Definition

Algorithm: a process or set of rules to be followed in calculations or other problem-solving operations

Definition

Pseudo Code: a notation resembling a simplified programming language, used in program design.

Definition

Implementation: the process of putting a decision or plan into effect; execution.

Problem...

- We need a machine, implementation independent way of comparing algorithms

- Complexity is a (the?) way we can tell whether one method is better than another.
- We get a measure of how long an algorithm will take to get the answer.
- We seek a measure of the size of a problem: e.g., number of items to sort, numbers of items to search, but this can be any other measure we wish.

- We need to know
  - how long it will take to solve, and
  - how much memory the solution will need.
- It's good to have an estimate of these two; time complexity and space complexity.

Big O notation

- We need some basic vocabulary to describe the behaviours.
- The basic idea is that we have the size of the problem (n), and we want to determine how long it takes to run/how much memory it takes/etc., and how that depends on n.

- $10n^3 + 15n^2 + 20 = 45$ (where $n = 1$)
- $n = 2 : 10n^3 + 15n^2 + 20 = 160$
- $n = 10 : 10n^3 + 15n^2 + 20 = 11{,}520$
- $n = 10 : (10 \times \underline{\mathbf{1000}}) + (15 \times 100) + 20 = 11{,}520$
- $O(n^3)$

Example : Constant Complexity - O(1)

```
private int[] myArray = new int[100000];
private int index = 0;

public void addItemToArray(int value){
    myArray[index] = value;
    index++;
}
```

Example : Linear Complexity - O(n)

```
private int[] myArray = new int[100000];

public void fillMyArray(int value){
    for(int i=0; i<myArray.length; i++){
        myArray[i] = value;
    }
}
```

Example : Quadratic Complexity - $O(n^2)$

```
private int[] myArray = new int[100000];

public void fillMyArray(int value){
    for(int i=0; i<myArray.length; i++){
        int sum = i;
        for(int j=0; j<myArray.length; j++){
            sum += myArray[j];
        }
        myArray[i] = value+sum;
    }
}
```

Example : Logarithmic Complexity - O(log n)

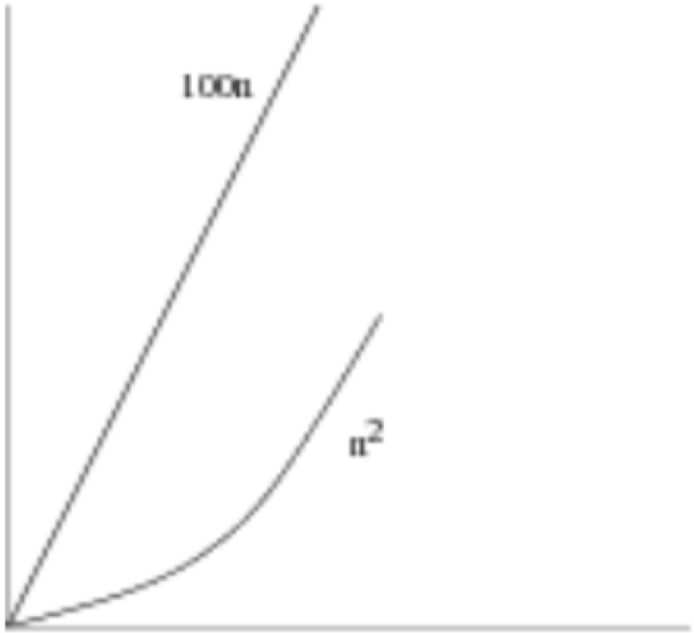
```
public void printSomeValues(int value){
    for(int i=1; i<value; i=i*2){
        System.out.println(i + ", ");
    }
}

public void printSomeOtherValues(int value){
    for(int i=value; i>1; i=i/2){
        System.out.println(i + ", ");
    }
}
```

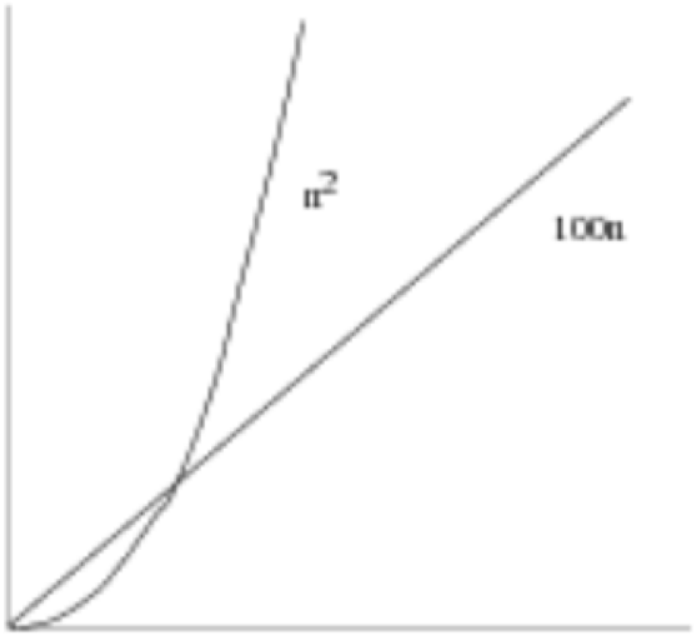
Example : Exponential Complexity - $O(2^n)$

```
public int fib(int number){
    if (number <= 1){
        return number;
    }
    else{
        return fib(number - 2) + fib(number - 1);
    }
}
```

- Suppose we find two algorithms that take times n^2 and $100n$ to run respectively.
- At first, the n^2 seems better, but the square term is eventually going to dominate:
- For n= 1million, n squared is a quadrillion. So the $100n$ algorithm is actually the better one (for large datasets).



small n



large n

- Next, suppose we have an algorithm that takes time $n^2 + 100n$.
- For $n = 1000000$, the $100n$ is 100 million, which is just $1/100\%$ of a quadrillion.
- So when we are talking about big values for n we need only think about the quadratic term as that forms the overwhelming part of the time.
- The linear term is important for small n, but we are really concerned with big n.
- We can simplify by ignoring the small stuff.

- We say the problem has complexity $O(n^2)$.
- This notation means that the problem grows like $an^2 +$ stuff where a is some constant, and "stuff" grows slower than n^2, and so will be minuscule relative to n^2 for large n
- this statement is valid for all large enough values of n
- this statement says nothing about small n, or what "large enough" is supposed to mean
- we're not too concerned as to whether it is $100n^2 +$ stuff or $2n^2 +$ stuff

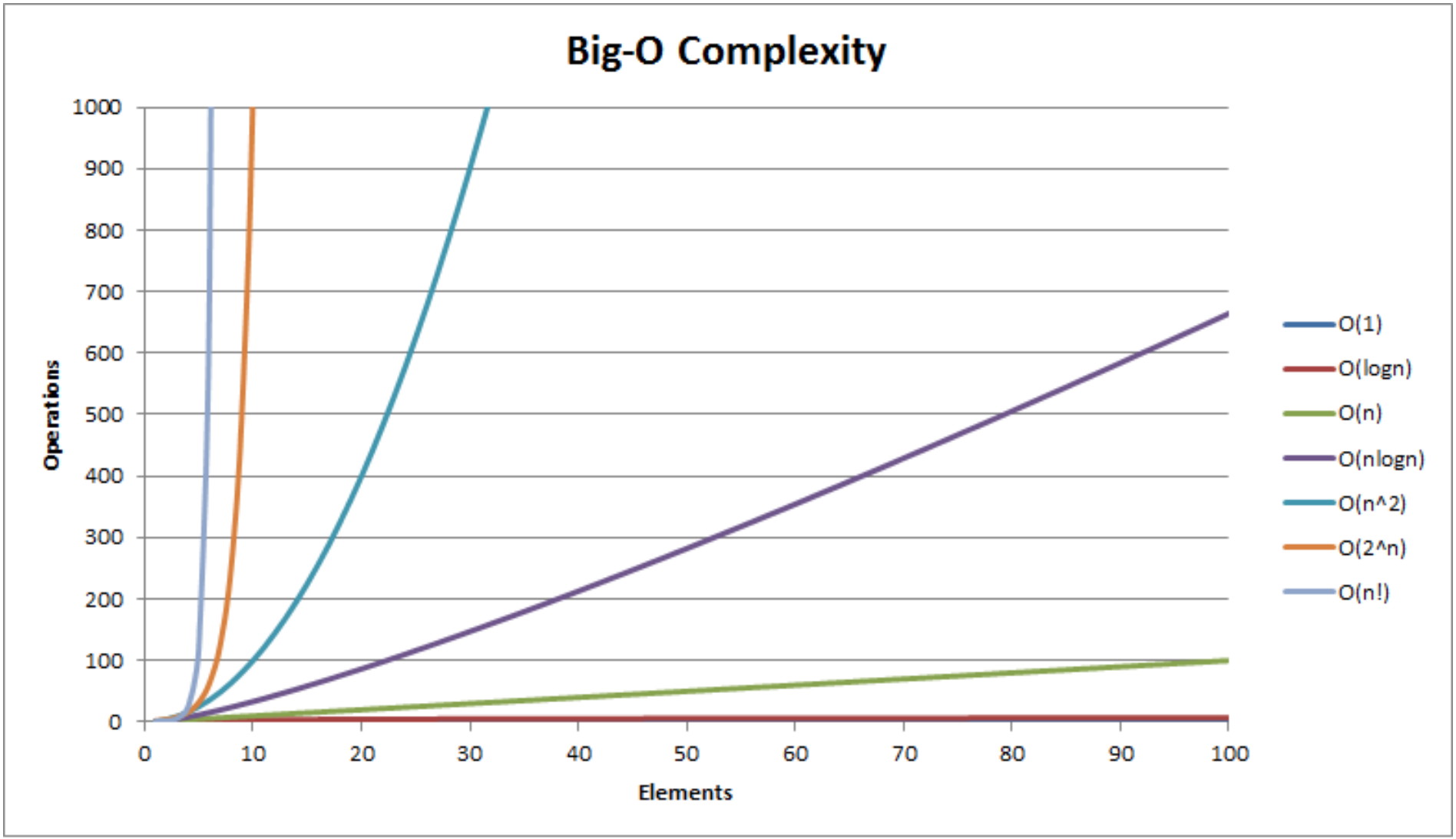
- The important part is how the time grows with n.
- All sorts of factors determine the actual time a program takes to run: the speed of the computer, how good the compiler of the language is, whether the processor has a good multiply operation, and so on.
- It is much more useful to compare relative times for various sizes of problem, i.e., n.
- If the complexity is $O(2^n)$, if a problem of size $n = 10$ is doubled it takes 2^{10} =1024 times as long. (because $2^{2n}/2^n = 2^n$)
- If a problem of size $n = 100$ is doubled it takes 2^{100} or approx.. 10^{30} times as long.

- It's best to avoid solutions that take exponential time, but some problems don't appear to have faster solutions.
- For example the **Travelling Salesman** (Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?)
- Note that just because they don't *appear* to have fast solutions doesn't mean they don't!
- New solutions to old problems are being found all the time.

- The smaller the degree the better
- Linear is nice,
- Logarithmic is better.

- There is a hierarchy:
- constant $<$ logarithmic $<$ $1/r$th power $<$ linear $<$ quadratic $<$ rth power $<$ exponential

n	n^2	$\log n$	$n \log n$	2^n
2	4	1	2	4
5	25	2.322	11.610	32
10	100	3.322	33.22	1024
100	10000	6.644	664.38	1.27×10^{30}
1000	10^6	9.966	9966	1.07×10^{301}
10^6	10^{12}	19.932	1.993×10^7	9.90×10^{301029}


Big-O Complexity
Operations
1000
900
800
700
600
500
400
300
200
100
0
0
10
20
30
40
50
60
70
80
90
100
Elements
O(1)
O(logn)
O(n)
O(nlogn)
O(n^2)
O(2^n)
O(n!)

Problems with Different Components

- Suppose a problem that has two parts, the first takes time O(n), the second O(n2).
- Taken together, the whole problem has complexity
- $O(n) + O(n^2) = O(n^2)$, as the linear part can be disregarded for large n.
- Note this does not imply that $O(n) = 0$
- It means that $c_1 n + c_2 n^2 < cn^2$ for some c and large n.
- Similarly, if a problem has two parts, both taking time O(n), then the total is $O(n) + O(n) = O(n)$.
- A problem of complexity O(1) is one that takes no more than a constant time to solve, regardless of its size. Again, that the constant time could be 1ms or 1 million years: the important point is it is independent of n.
- Here the hidden constant can be very important.

- Slow functions...

- Confusingly, we say a function is slow if it grows slowly (like log).
- When an algorithm complexity corresponds to a slow function, it is fast, i.e., the time it takes to run increases slowly.

- Cases - Algorithms are measured in many ways, but some popular cases are

- Best case: What is the best possible situation for this algorithm? Data are allowed to be in just the right values and in just the right places to make the algorithm perform at its best.
- Average case: What happens in the average case? This should be the behaviour we would expect to see normally with data that are in no particular configuration.

- Worst case: What is the worst that can happen? When the data happen to be in just the wrong configuration.
- Common case: Is there something special we know about the data we can use to our advantage? For example, nearly sorted data. Ideally, we want the common case to be the same as the best case.

Sorting Algorithms and their Complexity

- Lets look at some example of sorting and think about their complexity

- Selection Sort
- Insertion Sort
- Bubble Sort
- Merge Sort
- Quick Sort

Selection Sort

Selection sort pseudocode

```
function selection_sort(int[] A):
// given a list A of n numbers indexed 0 to n-1
    for i from 0 to n-2
        k=i //index of the smallest value so far
        for j from i+ to n-1
            if A[j] < A[k]
            then k=j
        swap A[i] and A[k]
```

Selection Sort Complexity

- The complexity of this algorithm is fairly easy to compute:
- the first i loop takes $n$steps,
- the next $n - 1$, and so on.

- The total is
- $n+(n-1)+(n-2)+\cdots+3+2= n(n+1)/2 - 1 = O(n^2)$
- Notice this time is independent of the data: even if the data
- is already sorted it takes $O(n^2)$ time!

Selection Sort Complexity

- Time: best $O(n^2)$, average $O(n^2)$, worst $O(n^2)$.
- Space $O(n)$. More precisely: $n + O(1)$
- Other criteria: moves items directly to their destination, which can be important if the cost of moving is high.
- A quadratic algorithm, so bad for large datasets.

Finer Comparisons

- Sometimes the complexity of sorting algorithms is more finely subdivided.
- We might not just count the number of steps, but also
  - the number of comparisons made,
  - and the number of moves made.
- For example, selection sort takes $O(n^2)$ comparisons but only $O(n)$ moves.

- Sometimes moving an object is much more expensive than comparing
- Example: think of sorting a line of cars into number plate order:
- (you want to move the cars as little as possible).
- In such a case you might want an algorithm that uses as few moves as possible, even to the extent of doing a lot more comparisons.
- Sometimes, a comparison is more expensive than a move.
- In this case you would want to minimise comparisons.
- For example, when sorting long strings of nearly-identical DNA into order, the comparison would be quite costly.

Insert Sort

Insert Sort: Algorithm

```
function insertion_sort(int[] A):
    for i = 1 to length(A) - 1
        j = i
        while j > 0 and A[j-1] > A[j]
            swap A[j] and A[j-1]
            j = j - 1
        end while
    end for
```

What is the worst case complexity?

Insert Sort Complexity

- If the data are already ordered, the j loop exits immediately every time, and we take $O(n)$ steps.
- If not, the comparing against the sorted part takes $1+2+3+\cdots+(n-2)=O(n^2)$ steps again.
- Time: best O(n), average $O(n^2)$, worst $O(n^2)$.
- Space: O(n)
- Other criteria: exceptionally good for nearly sorted data. Low overhead, so good for small datasets. Bad, since the $O(n^2)$ means this performs really poorly for large datasets. A lot of shuffling, thus is poor if moving objects is costly.
- Comparisons: $O(n^2)$. Moves: $O(n^2)$.

Bubble Sort

Bubble Sort Algorithm

Repeatedly steps through the list to be sorted, compares each pair of adjacent items and swaps them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted

What is the best case complexity for bubble sort?

```
function bubble_sort(int[] A):
    n = length(A)
    repeat
        swapped = false
        for i = 1 to n-1 inclusive do
            /* if this pair is out of order */
            if A[i-1] > A[i] then
                /* swap them and remember something
                   changed */
                swap( A[i-1], A[i] )
                swapped = true
            end if
        end for
    until not swapped
```

Bubble Sort Complexity

- If the data are already sorted, we make one pass through and stop immediately.
- Otherwise, we do about
- $(n-2)+(n-3)+\cdots+3+2+1 = O(n^2)$ steps.

- Time: best O(n), average $O(n^2)$, worst $O(n^2)$
- Space O(n).
- Other criteria: good for nearly sorted data. Low overhead, so good for small datasets. Bad, since the $O(n^2)$ means this performs really poorly for large datasets.
- Not so good for sorting lists. Comparisons: $O(n^2)$. Moves: $O(n^2)$.

Merge Sort

Merge Sort: algorithm

```
function merge_sort(int[] A):
    if(A.length == 1){
        return A;
    }
    else{
        return merge(merge_sort(A[0..A.length/2]),
            merge_sort(A[A.length/2..A.length]);
    }

function merge(int[] left, int[] right):
    int[] result;
    while left AND right are not empty:
        if first(left) <= first(right) then
            append first(left) to result
            left = rest(left)
        else
            append first(right) to result
            right = rest(right)
```

Merge Sort Complexity

- The complexity of this algorithm takes a little thought.
- The algorithm splits the list in half, sorts each half and then merges the two, so
- Time: best O(n log n), average O(n log n), worst O(n log n).
- Space: 2n.

- Other criteria: not so good as bubble sort with nearly sorted data.
- Merge requires extra space or extra time to shuffle.Two list merge sort has very stable predictable behaviour.
- One improvement is not to recurse all the way down to single elements: better is to switch to, say, insertion sort or even bubble sort when n is small enough.
- This takes advantage of the low overhead of such a sort and its speed on small datasets.

Crossover of Bubble and Merge Sort


time
bubble
merge
size
crossover
point

- The overhead of an algorithm is the amount of messing about in the algorithm that is needed to make it work but doesn't really contribute much.
- For example, shuffling in the mergesort is overhead bubblesort only has a tiny amount of overhead, but is a poor algorithm.
- mergesort has more overhead, but is a better algorithm.
- This is why they have a crossover point: where the larger overhead is outweighed by the better algorithm.
- The choice of algorithm is never easy to make!

Quick Sort

Quick Sort Algorithm

- Pick an element, called a pivot, from the array.
- Partitioning: reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.
- Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

Quick Sort Complexity

- Just as with mergesort, we find the algorithm takes time O(n log n), if we get even splits every time.
- Unfortunately, we can have bad splits too...
- ... in the worst case we get a partition into 0 plus $n - 1$ elements
- So, if we get bad splits, the time is $O(n^2)$.

- Choosing the pivot is critical
- When the data is already sorted or nearly sorted, quicksort does really poorly.
- Ditto for reversed or nearly reversed data.
- It is a strange feature of quicksort that sorted data is its worst case!

- Finding the pivot point

- One approach, rather counter-intuitively, is to randomise the order of the data before using quicksort: this might well give us a better run time!
- Random element as pivot
- Use two pivots
- Use information about the pivots

- Quick sort finding the middle

- It turns out that we can find the middle value of an array in O(n) time.
- Consider the algorithm **select** which:

- 1 consider the array in groups of 5 values: sort each 5 (using bubble sort, perhaps) and find the middle value of each
- 2 call select recursively on these middle values to find the overall middle value

- Using select to find a pivot, we can guarantee that quick sort has a good behaviour, with additional O(n log n) time (O(n) for log n sort steps) taken to find the pivot.
- Thus, still a total time of O(n log n), but somewhat longer than previously.
- Of course, it may or may not be worthwhile spending this amount of time finding the pivot when one of the methods above might be just as good: it all depends on the data.

Combining Quicksort with Insertion Sort

- Just as with any other divide and conquer method, there is no need to recurse all the way down to single elements in quicksort.
- Better is to stop at 3 or 4 elements and sort these directly;
- or revert to insertion (or bubble, selection or other low-overhead sort) when the list is short enough.
- This cut-over length must be determined experimentally as it will vary from implementation to implementation.

Summary

- Looked at complexity and Big O notation
- Examined complexity through different sorting algorithms