Section 2

Algorithm Complexity

Algorithm Complexity

- This section of the course is aimed at providing a framework for the analysis of algorithms.
- Constructing such a framework is not a simple task - we need some way to compare the performance of many very different algorithms. The results we end up with should be equally applicable to any platform.
- It s not sufficient to simply implement all the algorithms we wish to compare in Java, run them on a PC, and use a stopwatch to calculate timings. This will tell us nothing about how the same algorithms might run when written in C++ running on a supercomputer.
- This means our method of analysis must be
  - Independent of the hardware the algorithm will be run on. (Is it a PC or a Supercomputer?)
  - Independent of the implementation. (Is it written in Java or Visual Basic?)
  - Independent of the type of algorithm input (For a sorting algorithm - are we sorting numbers or words?)

Algorithm Analysis

- Our framework should allow us to make some predictions about our algorithms.
- In particular it should allow us to predict how efficiently our algorithms will scale e.g. given an analysis of a sorting algorithm we should be able to predict how it will perform given 10, 100 and 1000 items.
- As we shall see later implementation details and hardware speed are often irrelevant. Our framework will allow us to make design decisions about the appropriateness of one algorithm as opposed to another.
- Often in computer science there are several useful programming paradigms suitable for solving a problem e.g. divide and conquer, backtracking, greedy programming, dynamic programming.
- Proper analysis will allow us to decide which paradigm is most suitable when designing an algorithm.

Algorithm Analysis

- Definitions:
  - Basic Operation:
    - The performance of an algorithm depends on the number of basic operations it performs.
  - Worst-Case input:
    - Often when analysing algorithms we tend to take a pessimistic approach. The worst-case input for an algorithm is the input for which the algorithm performs the most basic operations.
    - For example a specific sorting algorithm may perform very well if the input (the numbers) it is given to sort are already partially sorted. It may perform less well if the input is arranged randomly.
    - We generally consider the worst case scenario for every algorithm as this provides a fair (and as it turn out simpler) basis for comparison.
    - Sometimes we will consider the average-case time for an algorithm. This is often far more difficult to evaluate - it often depends on the nature of the input, rather than the algorithm. e.g. On average a sorting algorithm may sort an English passage of words far faster than its worst-case analysis predicts, simply because English contains many short words. Sorting passages in German may result in an average-case time closer to the worst-case.

Basic Operations

- Deciding exactly what a basic operation is can seem a little subjective.
- For general algorithms like sorting, the basic operation is well defined (in this case the basic operation is comparison).
- A basic operation may be
  - An addition , multiplication or other arithmetic operation.
  - A simple comparison.
  - A complex series of statements that moves a database record from one part of the database to another.
  - A method which calculates the log of a number.
- In fact in general the actual specification of the basic operation is not that important.
- We need only ensure that its performance is roughly constant for any given input. e.g. a method which calculates the factorial of a number would be inappropriate as a basic operation as calculating the factorial of 5 may be far faster than calculating the factorial of 150.

A simple example

```
public static void main(String args[])
{
    int x,y;
    x= 10;
    y = Math.Random() % 15;
    // y is a random number between 0 and 14
    while ((x>0)&&(y>0))
    {
        x = x-1;
        y = y-1;
    }
}
```

- What is the *basic operation* in this algorithm?
- What is the *worst-case* for the algorithm?
- What is the *average-case* for the algorithm?

A simple example

```
public static void main(String args[])
{
    int x,y;
    x= 10;                                               Initialisation
    y = Math.Random() % 15;
    // y is a random number between 0 and 14
    while ((x>0)&&(y>0))
    {
        x = x-1;          Basic Operation
        y = y-1;
    }
}
```

- We tend to ignore the time required to do initialisation - this is constant for any given input.
- The running time of the program obviously depends on how many times the loop goes round.
- The work done inside the loop forms the basic operation.

Case Times

- It should be fairly obvious that the worst-case for this algorithm is 10 basic operations. This happens if the value of y is 10 or more. The loop will always end after 10 iterations.
- Y can take on values between 0 and 14 and the number of basic operations for each value of y is as follows:

•	y	: 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
•	basic ops	: 0	1	2	3	4	5	6	7	8	9	10	10	10	10	10

- Since each value of y is equally likely the average number of iterations is around 6.33 (95/15).
- This average analysis would change if y was more likely to have higher values i.e. it depends on the nature or distribution of y. The worst case analysis will not change.
- Usually we not too concerned with algorithms like this since they have a constant worst case - this sort of thing only worries computer game programmers.

Input Size

- More often we are concerned with algorithms that vary according to input size e.g.

```
public static void simple(int n)
{
     while (n>0)
     {
          n = n-1;
     }
}
```

- The worst-case for this algorithm is n basic operations. In general our worst-case analyses will be in terms of the input size.
- This example is somewhat misleading as ‘input size’ is determined by a single integer. Some better examples more typical input sizes are:

sorting	: Input size - number of items to sort
	: Basic Operation : comparison of two items
multiplication	: Input size - number of digits in both numbers
	: basic operation - single digit multiplication

Input Size

Database searching	: Input size - No. of database records
	: basic operation - checking a record
Maze solving	: Input size - No of paths between junctions
	: basic operation - checking for next junction or finish.

- Working with basic operations
  - We need some rules or operators which allow us to work out how many basic operations are performed in the worst-case of an algorithm.
  - Sequences : The time taken to execute a sequence of basic operations is simply the sum of the time taken to execute each of the operations.
  - T(sequence(b1;b2;b3)) = T(b1)+T(b2)+T(b3) - usually we roll these into a single basic operation.
  - Alternatives : When we have an if statement which will perform one basic operation or another, we calculate the time as the longest basic operation
  - T(if (cond) b1 else b2) = Max(T(b1),T(b2))

Calculating times

  - Iteration - when we have a loop the running time depends on the time of the basic operation and the *worst-case* number of iterations.
  - T(loop) = T(b1) * worst-case number of iterations

- **Exercise** : Identify the basic operations and input sizes in the following segments of code. Determine the running time.
- Segment 1-

```
int n=5;
while(n>0)
    n=n-1;
```

- Segment 2 -

```
public static void algor(int n)
{
int x=5;
        if (n>0)
                x=x+5;
        else
                x=x-5
}
```

Calculating times

- Segment 3

```
public static void algor(int n)
{
int x=5;
        while (n>0) {
            n=n-1;
            if (n%2==0)
                x=x+5;
            else
                x=x-5
     }
}
```

- Segment 4

```
public static void algor(int n)
{
int x=5;
   if (n %2==0)
        while (n>0) {
            n=n-1;
            if (n%2==0)
                x=x+5;
            else
                x=x-5
     }
}
```

Calculating times

- Segment 1-

```
 int n=5;
 while(n>0)
    n=n-1;
```

- Basic Operation : n=n-1 - we will call this b1
- Time to execute b1 : T(b1) - This is just notation
- Worst case number of iterations - loop is performed at most 5 times
- Time = T(b1) * 5
- Input Size : In the above expression all the elements are constant therefore this piece of code has a constant running time. Input size would not have any effect on this code.

Calculating times

- Segment 2

```
public static void algor(int n)
{
int x=5;
        if (n>0)
                x=x+5;  } Basic operation  - b1
        else
                x=x-5;  } Basic operation  - b2
}
```

- With an if statement we assume the worst. We know one alternative or the other must be executed so at worst the running time is the slowest of the two alternatives. Time = Max(T(b1), T(b2))
- This piece of code has no loop - again the running time just contains constant elements. Input size has no effect.

Calculating times

- Segment 3

```
public static void algor(int n)
{
int x=5;
        while (n>0) {
            n=n-1;          }  Basic operation - b1
            if (n%2==0)
                x=x+5;      }  Basic operation - b2
            else
                x=x-5       }  Basic operation - b3
     }
}
```

- Working from the inside out - The if statement = Max(T(b2),T(b3)) . This is in sequence with basic operation b1. The total time to perform the operations inside the loops is T(b1) + Max(T(b2),T(b3))
- The loop depends on the value of n - It will perform at most n iterations
- Time = n * [T(b1) + Max(T(b2),T(b3))] (we assume n>=0)
- This equation depends on n - as n gets larger the time increases. The input size is n.

Calculating Times

- Segment 4

```
public static void algor(int n)
{
int x=5;
     if (n%2==0)
        while (n>0) {
            n=n-1;                        //basic operation b1
            if (n%2==0)
                x=x+5;                    //basic operation b2
            else
                x=x-5                     //basic operation b3
     }
}
```

- As before, time for the loop is Time = n * [T(b1) + Max(T(b2),T(b3))]
- The loop is enclosed inside an if statement. This if statement either gets executed or it doesn’t - this implies the running time is Max(n * [T(b1) + Max(T(b2),T(b3))], 0)
- Obviously this is equivalent to n * [T(b1) + Max(T(b2),T(b3))]
- This equation depends on n - as n gets larger the time increases. The input size is n.

Input Size

- As mentioned previously we are not concerned with algorithms which have a constant running time. A constant running time algorithm is the best kind of algorithm we could hope to use to solve a problem.
- Code segments 1 and 2 both have constant running times. We could not possibly substitute an alternative algorithm which would be generally faster. When we reach a situation like this improvements in speed depends purely on how well a piece of code is implemented - a purely mundane task.
- Code segments 3 and 4 both had a running time which was proportional to n. (Some constant times n). With these problems if n doubles the running time doubles.
- When we attempt to analyse algorithms in a general sense it is the dependency on input size which is most important.

Input Size Dependency

- The following table illustrates 4 different algorithms and the worst number of basic operations they perform

Input Size	Algorithm 1	Algorithm 2	Algorithm 3	Algorithm 4
1	1	2	4	8
10	1024	200	40	11.32
100	2^{100}	20000	400	14.64
1000	2^{1000}	2000000	4000	17.97

- If we assume that each algorithm performs the same basic operations - and that a typical computer can execute 1 million basic operations a second then we get the following execution times.

Input Size Dependency

Input Size	Algorithm 1	Algorithm 2	Algorithm 3	Algorithm 4
1	1 μs	2μs	4μs	8μs
10	1.024ms	200μs	40μs	11.32μs
100	2^{75}years	20ms	0.4 ms	14.64μs
1000	2^{975}years	2s	4 ms	17.97μs

- These algorithms have running times proportional to 2^n, n^2, n and $\log_2(n)$ for input size n.
- These figures should illustrate the danger of choosing an ‘optimal’ algorithm by simply implementing several algorithms to solve a problem and choosing the fastest.
- If we were to choose based on an input size of 1 then we would probably choose the first one!

Big-*O* Notation

- It should be clear from the previous example that the efficiency of an algorithm depends primarily on the proportionality of the running time to the input size.
- This is the framework we have been searching for all along.
- It is independent of implementation details, hardware specifics and type of input.
- Our task when performing an analysis is to describe how the running time of a given algorithm is proportional to the algorithms input size. i.e. we make statements like - “the running time of this algorithm will quadruple when the input size doubles” or “the running time is 5 times the log of the input size”.
- Statement like these can be open to different interpretations - since algorithm analysis is so central to Computer Science a more formal notation exists. This notation is known as Big-*O* notation.

Big-*O* Notation

- Big-*O* notation - formal definition
- Let *f*:**N**->**R** i.e. f is a mathematical function that maps positive whole numbers to real numbers. E.g. *f*(n) = 3.3n => *f*(2) = 6.6, *f*(4) = 13.2
- In fact *f* abstractly represents a timing function - consider it as a function which returns the running time in seconds for a problem of input size n.
- Let *g*:**N**->**R** be another of these timing functions
- There exists values *c* and *n0* (where n0 is the smallest allowable input size) such that
- ***f*(n) <= *c*g(*n) when n>=*n0*** or ***f*(n) is *O*(*g*(n))**
- Less formally - given a problem with input size n (n must be at least as big as our smallest allowable input size *n0*) our timing function *f*(n) is always less than a constant times some other timing function *g*(n)
- *f*(n) represents the timing of the algorithm we are trying to analyse.
- *g*(n) represents some well known (and often simple) function whose behaviour is very familiar

Big-*O* Notation

- What's going on? How is this useful?
- During algorithm analysis we typically have a collection of well known functions whose behaviour given a particular input size is well known.
- We've already seen a few of these
  - Linear : g(n) = n - if n doubles running time doubles
  - Quadratic : g(n) = n^2 - if n doubles running time quadruples
  - Logarithmic : g(n) = lg(n) - if n doubles the running time increase by a constant time
  - Exponential : g(n) = 2^n - as n increases the running time explodes.
- These functions allow us to classify our algorithms. When we say *f*(n) is *O*(*g*(n)) we are saying that given at least some input size f will behave similarly to g(n).
- Essentially this expresses the idea of an algorithm taking at most some number of basic operations.

Big-*O* Notation

- Examples:
  - 3n is *O*(n)
  - 4n +5 is *O*(n)
  - 4n +2n +1 is *O*(n)
  - 3n is *O*(n^2)
  - n*Max[T(b1),T(b2)] is *O*(n)
  - n* (T(b1) + n*T(b2)) is *O*(n^2)
- The last two examples might represent the running times of two different algorithms. It is obvious that we would prefer our algorithms to be *O*(n) rather than *O*(n^2). This classification using *O*-notation allows us to make the appropriate decision and pick the first of the two.

Analysis Example

- Our task is to decide on an algorithm to find the largest number in an array of integers.
- We'll use two different algorithms and analyse both to see which is computationally more efficient.
- Algorithm 1:

```
public static int findLargest(int numbers[])
{
          int largest;
          boolean foundLarger;
          foundLarger = true;

          // Go through the numbers one by one
          for (int i=0; (i < numbers.length) && (foundLarger); i++)
          {
                    foundLarger = false;
                    // look for number larger than current one
                    for (int j=0; j< numbers.length; j++)
                    {
                              // This one is larger so numbers[i] can't be the largest
                              if (numbers[j] > numbers[i])
                                        foundLarger = true;
                    }
                    // If we didnt find a larger value then current number must be largest
                    if (!foundLarger)
                              largest = numbers[i];
          }
          return largest;
}
```

Algorithm analysis

- In order to analyse the algorithm we must first identify the basic operations. They appear in bold in the previous slide. We'll label them b1..b5
- Usually we tend to ignore the constant times b1 and b5. They're just included here for completeness.
- Next we have to identify the input size. The input size is simply the number of elements in the array (we'll call this n). In general the input size is always obvious.
- We have to work out the timing for this algorithm and then use *O*-notation to decide how to categorise it.
- Using the rules we've seen earlier we get
- Time = T(b1) + n * (T(b2) + n * Max (T(b3),0) + Max(T(b4),0)) + T(b5)
- If we eliminate the redundant Max functions and multiply this out we get
- Time = n^2*T(b3) + n*(T(b2) + T(b4)) + T(b1) + T(b5)

Algorithm Classification

- Using *O*-notation :
  - We must try an classify our Time function (*f*(n)) in terms of one of the standard classification functions (*g*(n)).
  - recall the definition of *O*
    - ***f*(n) <= *c*g(*n) , n >= n0**
  - Our time function has a n^2 term in it so it could not possibly be *O*(n) i.e. linear. We could never pick a constant *c* that would guarantee $n^2 <= c*n$.
  - Our time function is *O*(n^2) - Essentially the function is of the form
  - $an^2 + bn + d$, where a, b and d are just arbitrary constants. We can always choose a sufficiently large value *c* such that : $an^2 + bn + d <= c*n^2$
  - In general if the timing function for an algorithm has the form of a polynomial then we can simply take the highest power in the polynomial and this will give us our *g*(n).

Algorithm Analysis

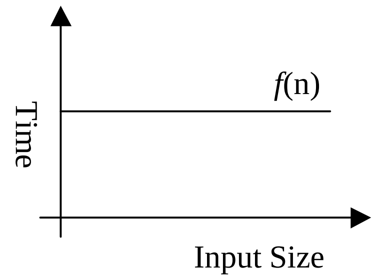
- Now we will analyse a different algorithm which performs the same task.

```
public static int findLargest(int numbers[])
{
          int largest;
          largest = numbers[0];
          // Go through the numbers one by one
          for (int i=1;  i < numbers.length ; i++)
          {
                    // Is the current number larger than the largest so far ?
                    if (numbers[i] > largest)
                              largest = numbers[i];
          }
          return largest;
}
```

- A similar analysis to the previous example gives
- Time = n*T(b2) + T(b1) + T(b3)
- The highest power of n is 1 this implies this algorithm is O(n) or linear.
- O(n) is a large improvement on $O(n^2)$. This algorithm is much more efficient.

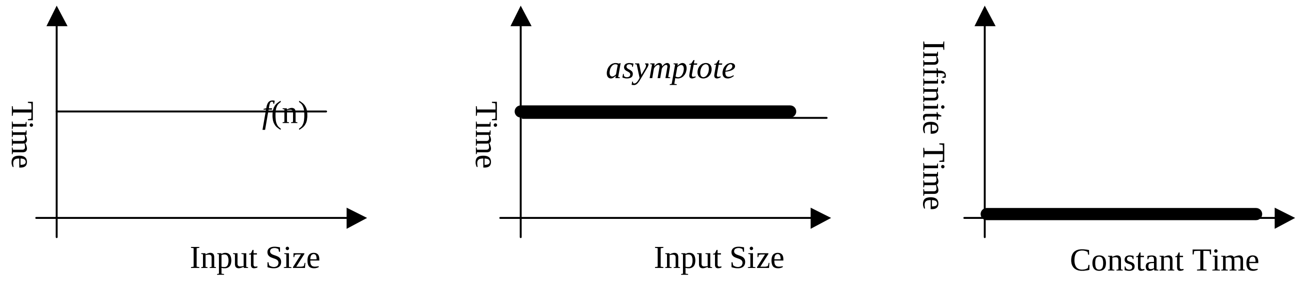
Algorithm Classification

- *O*-notation is nothing more than a mathematical notation that captures the idea of the slope of an asymptote.
- Programs with constant running times look like this as input size varies.



- The actual position of the line is irrelevant. When we use *O*-notation we are implicitly performing the following steps.
  - 1. Picture the running time of the algorithm *f*(n)
  - 2. Calculate the asymptote to the graph of this function as n goes to infinity
  - 3. Ignore the position of the asymptote, move it back to the origin an compare this slope with the slope of standard graphs.

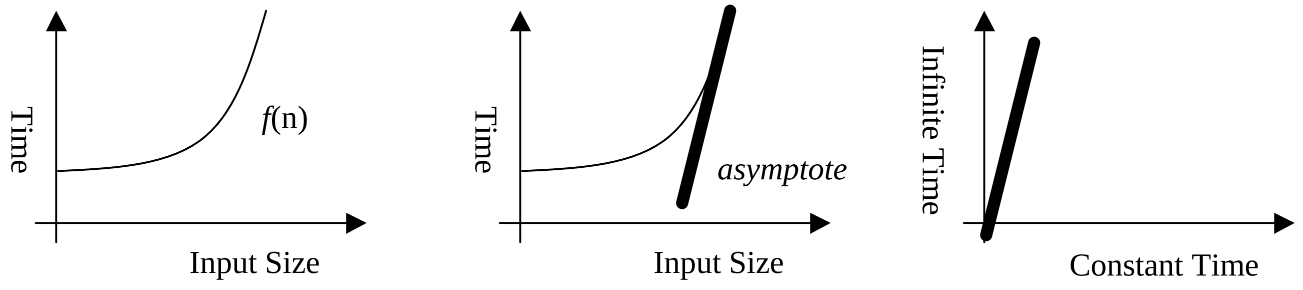
Algorithm Analysis



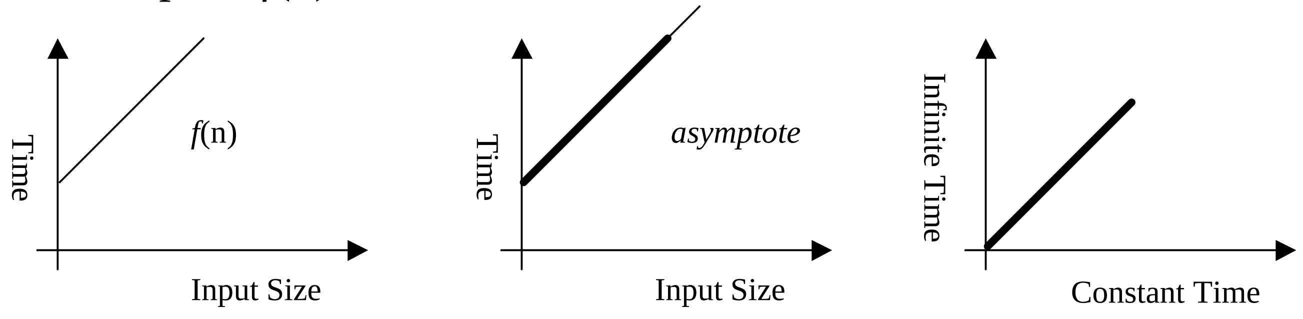
- We've performed the three implicit steps here. Note that the asymptote to a line is just the line itself.
- Constant running times all map to the black line in the diagram above. An infinite running time would map to the y axis. All other algorithms lie somewhere in between. For the moment we won't include the standard ones.
- Lets examine how our previous two algorithms would perform.
- Before we can actually graph the running time we must first give values to the time it takes to execute the basic operations. We'll can just assume they all take 1 unit of time each. The actual value we pick is irrelevant. It won't affect the slope of the asymptote.

Algorithm Analysis

- Algorithm 1 : Time = n^2*T(b3) + n*(T(b2) + T(b4)) + T(b1) + T(b5)
  - this implies $f(n) = n^2 + 2n + 2$
- Following the three step from before we get

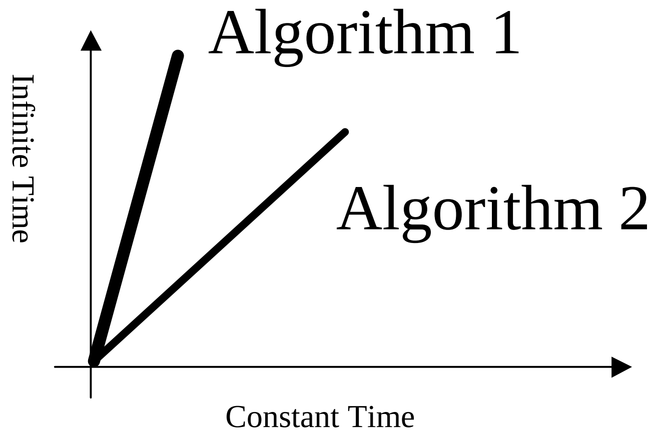


- Similarly for algorithm 2 : Time = n*T(b2) + T(b1) + T(b3)
  - this implies $f(n) = n+2$



Algorithm Analysis

- Comparing these directly



- It is clear that algorithm 1 is less efficient than algorithm 2.
- The lines for algorithm 1 and algorithm 2 represent the standard lines for *O*(n) and *O*(n^2).

OM-Notation

- We use *OM*-notation to describe an algorithm which takes at least some number of basic operations. It represents a lower bound on the worst case running time of an algorithm.
- *OM* notation - formal definition
- Let *f*:**N**->**R** and *g*:**N**->**R**
- There exists values *c* and *n0* such that
- ***f*(n) >= *c*g(*n) when n>=*n0*** or ***f*(n) is *OM*(*g*(n))**
- Less formally - given a problem with input size n (n must be at least as big as our smallest allowable input size *n0*) our timing function *f*(n) is always at least as large as a constant times some other timing function *g*(n).
- *OM*-notation is often more difficult to work out than *O*-Notation.
- Examples :
  - 10n is *OM*(n)
  - $10n + 2n^2$ is *OM*(n^2)

THETA-notation

- Again let *f*:**N->R** and *g*:**N->R**
- If *f*(n) is *O*(*g*(n)) and *f*(n) is *OM*(*g*(n)) then we can say
- *f*(n) is *THETA*(*g*(n))
- *f*(n) is said to be asymptotically equivalent to *g*(n)
- Really what this says is that algorithms *f*(n) and *g*(n) behave similarly as n grow larger and larger.
- If we find a specific algorithm is *THETA*(*g*(n)) then this is an indication that we cannot improve on the design of this algorithm. Its an indication that it is time to hand the algorithm over to some data monkey to let them improve the implementation.
- This does not mean that we couldn't replace the algorithm with a better one. It just means that this specific algorithm is as well designed as it could be.

Notation Rules

- f(n) is O(g(n)) if and only if g(n) is OM(f(n))
- If f(n) is O(g(n)) then
  - f(n) + g(n) is O(g(n))
  - f(n) + g(n) is OM(g(n))
  - f(n) * g(n) is O($g(n)^2$)
  - f(n) * g(n) is OM($f(n)^2$)

- Examples : let f(n) = 2n g(n)= $4n^2$
  - f(n) is O(g(n))
  - 2n + $4n^2$ is O(n^2)
  - 2n + $4n^2$ is OM(n^2)
  - 2n * $4n^2$ is O(n^4)
  - 2n * $4n^2$ is OM(n^4)