CS 608 Algorithms and Computing Theory

Week 2: Algorithm Analysis

1. Algorithm

An algorithm is a step-by-step procedure for solving a given problem in a finite amount of time.

2. Analysis of algorithms

If you have two algorithms to solve a problem, we need to identify the better of the two algorithms. We need to "measure" the efficiency of an algorithm so that we can compare the two algorithms.

Given an algorithm, it may not behave the same way for all kinds of input data. The algorithm may be efficient for certain input data and may be bad for certain other kind of input data. We consider three cases:

- **Best case:** Defines the input for which the algorithm takes the least amount of time to finish.
  The best case performance of an algorithm does not really reflect of the efficiency of the algorithm. Just because an algorithm works well for the best input data, it does not mean it is a good algorithm. Typically the input data will not be the best input.
- **Worst case:** Defines the input for which the algorithm takes a long time to finish.
  The worst case performance of an algorithm does not really reflect of the efficiency (or deficiency) of the algorithm. Just because an algorithm works badly for the worst input data, it does not mean it is a bad algorithm. Typically the input data will not be the worst input. The worst case scenario guarantees that for any input, the performance of the algorithm will not be worse than this.
- **Average case:** Defines the behavior of the algorithm for random input data.
  The average case performance of an algorithm is the one we normally like to analyze. This case gives the typical behavior of the algorithm.

3. The Big-O notation

We measure the efficiency of an algorithm by considering the number of computational steps the algorithm takes to solve the given problem. The computational steps may be number of multiplications, number of data movements and so on. Let us denote this measure T(n), where n is the size of the input data. We may not be able represent T(n) as a concrete formula. Instead, we use a notation, called Big-O, which gives an upper bound of T(n).

Definition of Big-O

$T(n) = O(f(n))$ if there are positive constants c and n_0 such that $T(n) \leq cf(n)$ for all $n \geq n_0$.
This mathematical definition may hard to comprehend. But, the spirit of this definition is easy to appreciate.

Let me explain this informally. If the number of input data elements n is small, the speed of an algorithm is not important. The current computers are so fast for small n any algorithm completes the job quickly.

For example, consider two algorithms to solve a problem with n data elements. Assume that the first algorithm takes 1000n steps and the second takes $4n^2$ steps.
Notice that $1000n < 4n^2$, for $n > 250$. This means the second algorithm takes less time than the first algorithm for small values of n (<250). For large n (>250), the first algorithm takes less time than the second algorithm.

If n is small, both the algorithms take the roughly same amount of time to complete the job. But, if n is very large, the time taken by these vary significantly. Thus, we consider n to be large when we discuss the analysis of algorithms.

In terms of the big-O notation, the algorithm with 1000n steps is O(n) and the algorithm with $4n^2$ is $O(n^2)$.

In practice, given a function, T(n), to find the big-O representation, we don't go through the rigorous method of finding n_0, c, and f(n), as given in the definition. Instead, we adopt

this short cut: Given T(n), consider only the largest exponent of n, ignoring all constant factors and constants to get a of big-O representation.

Examples:

1. $T(n) = 100n^3 - 15n^2 + n - 1000 = O(n^3)$
2. $T(n) = 2000 + 16n^2 = O(n^2)$
3. $T(n) = \frac{4n^3+100n^2+1000}{15n^2+25}$ (treat this as $\frac{n^3}{n^2}$) $= O(n)$
4. $T(n) = \sqrt{n} + 200 = O(\sqrt{n})$
5. $T(n) = 42 = O(1)$ (This is constant – no n. Thus it is represented O(1)).

Certain standard growth rates:

Time complexity	**Name**
1	Constant
logn	Logarithmic
n	Linear
nlogn	Linear logarithmic
n^2	Quadratic
n^3	Cubic
2^n	Exponential

The table is in the increasing rates of growth. That is,

$1 < \log n < n < n\log n < n^2 < n^3 < 2^n$

Note:

In the definition of big-O notation, we use $T(n) \leq cf(n)$, for $n \geq n_0$. This in other words means, the growth of T(n) is less that the growth of f(n). The function f(n) is an upper bound for T(n). By definition f(n) is NOT the least upper bound. Let me explain this with a simple example,

Consider the algebraic expression, $4 - x^2$, for a positive integer x, we can safely say that

$4 - x^2 < 4$. 4 is an upper bound for the expression $4 - x^2$. This means $4 - x^2$ is always less than 4 (for all positive x). Also notice that $4 - x^2$ is also less than any number larger than 4. That is, as examples, $4 - x^2 < 20$, $4 - x^2 < 100$, and so on. So 4, 20, 100 are all upper bounds for $4 - x^2$. (4 is the least upper bound).

Similarly, suppose $T(n) = 4n^2 + 3n - 50$, then $T(n) = O(n^2)$, then these are also theoretically true: $T(n) = O(n^3)$, $T(n) = O(n^4)$, $T(n) = O(n^5)$, and so on. But the best answer is $T(n) = O(n^2)$.

4. The Ω notation

As we saw, the big-O gives an upper bound for a function. The Ω notation gives lower bound for a function. Here is the definition for Ω notation:
$T(n) = \Omega(f(n))$, if there is a positive constant c and n_0 such that $T(n) \geq cf(n)$ for $n \geq n_0$.

In practice we are not interested in the Ω notation. In the analysis of an algorithm, we are interested in finding out the upper bound of the time complexity not the lower bound.

Note: the character Ω is the Greek letter Omega.

5. The Θ notation

$T(n) = \Theta(f(n))$ if and only if $T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$.
That is, the upper bound and lower bound for the function are same.
In practice, we don't use this notation too often.

Note: the character Θ is the Greek letter Theta.

6. Maximum subsequence sum problem

Designing efficient algorithms to solve problems is very important. Sometimes, the difference in the performance between two algorithms to solve the same problem may be dramatic. To illustrate this important point the Weiss textbook describes a problem of finding the largest sum found in any contiguous subarray of a given array of numbers (containing both positive and negative numbers).
The Weiss' book discusses four algorithms to solve this problem. This problem is also discussed in a very popular (and respected) book: Programming Perls by Jon Bently. The

same four algorithms are discussed in Bently's book. The time complexity of these four algorithms are $O(n^3)$, $O(n^2)$, $O(n^2)$ and O(nlogn). Briefly study the four algorithms. The Weiss's textbook has given Java code for the implementation of all the four algorithms.

I suggest that you read Bently's book (page 69) as well. Here is a link for Bently's book: https://maniali.wikispaces.com/file/view/Programming_pearls.pdf

7. Running time calculations in programs

To calculate the running times of various statements in a program, we use the following general rules:

1. **Simple statements:**
   Simple statements, such as assignment, adding a constant, multiplying by a constant, and declaration of variables, take constant time. Without losing accuracy, we these statement to count as one unit of time.
2. **Loops:**
   The running time of a loop is calculated by adding the running times of all the statements within the loop multiplied by the number of iterations.
   Example:

```
for(i=1; i<=n; i++){
  x++;
  y++;
}
```

   The loop has two statements (1 unit of time each). The loop iterates n times. Thus the total running time for the loop is 2n = O(n).
3. **Nested loops:**
   The running time of a nested loop is the product of the sizes loops and the running times of the statements in the nested loop.
   Example:

```
for(i=1; i<=n; i++)
  for(j=1; j<=n; j++){
     x++;
     y++;
}
```

The nested loop has two statements (1 unit of time each). The nested loop has two loops. The outer loop iterates n times and the inner lop iterates n times. Thus the total running time for the nested loop is $2n^2 = O(n^2)$.

4. **Consecutive statements:**
   Add the running times of each of the statements and then express in big-O.

5. **If/else:**
   The **if** itself is one unit of time. Look at the **if** black and the **else** block. Calculate the running times of these blocks. To express the running times in big-O notation, just consider the larger of the running times of the **if** block and the **else** block. After all, the big-O represents the upper bound.

   Example:

```
If (a < b){
 for (i=1;i<=n;i++)
   x++;
}
else{
  for(i=1;i<=n;i++)
    for (j=1;j<=n;j++)
        y++;
}
```

   The **if** itself takes 1 unit of time. The running time of the **if** block is n; the running time of the **else** block is n^2. Thus the total running time is $1 + n^2 = O(n^2)$.

8. logarithmic complexity

O(logn) appears in algorithms which work the following way: In solving a problem with n elements, the algorithm cleverly splits the n elements into two parts (normally the mid-point), and works on one half ignoring the other half.

A good example in which O(nlogn) appears is the famous binary search. We will discuss binary search in a later chapter. We will briefly see how O(logn) comes the big-O representation.

Briefly this the binary search algorithm:
Consider a sorted array of n numbers:

10	12	21	23	27	29	31	33	35	39	42	44	45	48	52	54	58	60	62

The problem is, given an element, say 58, we need to find if the number appears in the array of n elements. Here is the binary search algorithm: first compare 58 to the middle element of the array. In this case it is 39. Because 58 > 39, we will NOT look at the left half of the array. We focus on the right half of the array. The problem is cut in half. The same strategy is continued: we now compare 58 to the mid-point of the right half of the array, and so on. We continue this process until the size of the subarray is reduced to 1.

Given the original size of the array to be n, the question is how many steps does it require to reach subarray size of 1?

Let us denote the number of steps to be x. After x steps the subarray size is reduced to 1.
The size of the subarray after step 1 is $\frac{n}{2}$
The size of the subarray after step 2 is $\frac{n}{2^2}$
The size of the subarray after step 3 is $\frac{n}{2^3}$
....
....
The size of the subarray after step x is $\frac{n}{2^x}$
And the last size is 1.

So, $\frac{n}{2^x} = 1$
$2^x = n$
Taking $\log_2$ on both sides,
$\log_2(2^x) = \log_2 n$
$x \log_2 2 = \log_2 n$ (using log rules)
$x = \log_2 n$ (remember $\log_2 2 = 1$)

So number of steps in binary search is O(logn).

Note: In the representation of big-O notation, the base of log is not important: Remember the formula,

$\log_a m = \frac{log_b m}{log_b a}$

Notice that $\log_a m$ and $\log_b m$ differ by a constant factor $\frac{1}{log_b a}$.

And big-O representation constant factor can be ignored.

$O(\log_2 n) = O(\log_{10} n) = O(\log_e n)$.

Normally, base is not shown in the big-O expressions.

Another example of O(logn)

Consider this simple loop:

```
for(i=1; i<n; i=2*i){

}
```

Notice that the value of i is being doubled after each iteration. The variable i assumes these values: 1, 2, 4, 8, 16, ... The loop stops when i becomes = n.

We need to find out how many steps does it take for i to reach n. Let us say it takes x steps.

Again, the variable i assumes these values: 1, 2, 2^2, 2^3, 2^4, ..., 2^x.

$2^x = n$

$x = \log_2 n$

$x = O(\log n)$.

9. Practical experimentation with running time

Java provides a simple way of measuring the time taken for a program (or a part of the program) to run.

We will **currentTimeMillis**() method of the System class.

This method returns a **long** integer, which is the number of milliseconds since midnight, January 1, 1970 UTC.
See https://docs.oracle.com/javase/7/docs/api/java/lang/System.html for details on this method.

If you want to measure the time taken by a segment of the program, call this method (and save the return value) just before the code segment starts and call again (and save the return value) immediately after the code segment ends. Compute the difference between these two values, which gives the time elapsed in milliseconds. You can easily convert this into seconds (search for this on the Web).

---

10. Horner's method of evaluating a polynomial

Computer scientists and programmers keep looking for algorithms to reduce the run time complexity of solving a problem.
Here is a simple example how the evaluation of a polynomial can be done more efficiently than the direct method of evaluating each term individually and adding them.
As an example, consider a polynomial:
$4x^3 + 15x^2 + 6x + 17$
Let us evaluate this polynomial for x = 3.

One way of doing this is evaluate each term separates and then add. That is, find 4*x*x*x, 15*x*x, 6*x for x = 3and then add individual values and the constant 17 to get the final result.

This example requires 6 multiplications (count the number of *).

In general, evaluating a polynomial of degree n in this method requires n(n+1)/2 multiplications and n additions (this the upper bound). (This can be reduced 2n – 1 multiplications and n additions by evaluating the powers of x iteratively.)

The following method, called Horner's method, makes evaluation of a polynomial efficient by reducing the number of multiplications.

Here is Horner's method requires that we rewrite the given polynomial before evaluationg it.

Example: Let us take the same polynomial:
$4x^3 + 15x^2 + 6x + 17$
We write this as follows:
$(4*x^2 + 15*x + 6)*x + 17$
$((4*x + 15)*x+6)*x + 17$
This form of the polynomial requires three multiplications and 3 additions.
Notice how the number of multiplications is reduced.

In general,
Given a polynomial
$f(x) = a_n x^n + a_{n-1} x^{n-1} + a_{n-2} x^{n-2} + \ldots + a_2 x^2 + a_1 x + a_0$
We rewrite it in the Horner's method as follows:
$f(x) = ((\ldots ((a_n x + a_{n-1}) x + a_{n-2}) x + \ldots + a_2) x + a_1) x + a_0$

Coding Horner's method
To code Horner's method, we keep the coefficients of the polynomial in an array A[] of size n+1:

$A[] = \{a_n, a_{n-1}, a_{n-2}, \ldots, a_0\}$
Actual computation is just a simple loop:

```
int result = A[0];  // Initialize result with a_n
// evaluate value of polynomial using Horner's method
for(int i=n-1; i >= 0 ; --i)
      result = result * x + a[i];
return result;
```

11. More examples

1. Order the following function in the increasing order:
   $N \log N$, $\sqrt{N}$, 2^N, N, $N^{1.5}$, N^2, $N \log(N^2)$, $N^2 \log N$, N^3.
   Place your answer in the boxes below, with expressions with equal complexity in the same box.

Answer:

$\sqrt{N}$	N	N log N N log(N^2)	N log(N^2)	$N^{1.5}$	N^2	N^2 log N	N^3	2^N

Notice that $N \log(N^2) = 2N\log(N)$. So $O(N \log N) = O(N \log(N^2))$

---

2. Express each of the following functions in the big-O notation:

   (a) $\frac{n^3+2n^2+14n+100}{3n^2+15n+10}$

   (b) 20nlogn + √n + 30

   (c) n^2logn + nlog(n^2) + 30n + 500

   (d) $1^2 + 2^2 + 3^2 + 4^2 + \ldots + n^2$

   (e) $\frac{n\sqrt{n}+10n+10}{4\sqrt{n}+100}$

   Answers:

   (a) $\frac{n^3+2n^2+14n+100}{3n^2+15n+10} \approx \frac{n^3}{n^2} = O(n)$

   (b) O(nlogn)

   (c) O(n^2logn)

   (d) A formula for the given sum is (see http://www.trans4mind.com/personal_development/mathematics/series/sumNaturalSquares.htm)

$1^2 + 2^2 + 3^2 + 4^2 + \ldots + n^2 = \frac{n(n+1)(2n+1)}{6}$

$= O(n^3)$

(e) $\frac{n\sqrt{n}+10n+10}{4\sqrt{n}+100} \approx \frac{n\sqrt{n}}{\sqrt{n}} = (n)$

3. In each of the following pairs of functions, indicate which runs faster (that is lower running time):
   (a) (i) $\log^2 n$ and (ii) $\log n^2$
   (b) (i) $\sqrt{n}\ log\sqrt{n}$ and (ii) nlogn
   (c) $n\sqrt{nlogn}$ and (ii) $n\log n^2$

   Answers
   (a) $\log^2 n$ (Note: $\log n^2 = 2\log n$
   (b) $\sqrt{n}\ log\sqrt{n}$ (Note: $\sqrt{n}\ log\sqrt{n} = \frac{1}{2}\sqrt{n}\ logn$)
   (c) $n\log n^2$ (**Note**: $n\log n^2 = 2n\log n$)

4. Assume $T_1(n) = O(f(n))$ and $T_2(n) = O(f(n))$.
   True or false: $\frac{T_1(N)}{T_2(N)} = O(1)$.

   Answer: False. Take for example,
   $T_1(n) = n^2 + 5n + 15 = O(n^2)$
   $T2(n) = 5n + 15 = O(n)$

   Even though $T_2(n) = O(n)$ is the right answer, we can also say $T2(n) = O(n^2)$.
   Technically speaking, $T2(n) = O(n^2)$.
   Thus we have $T_1(n) = O(n^2)$ and $T_2(n) = O(n^2)$
   Now consider: $\frac{T_1(N)}{T_2(N)} = \frac{n2 + 5n + 15}{5n + 15} = O(n)$.

---

5. Write a running time expression for the following code segment:

```
sum = 0;
for (i=0; i<n; i++)
        sum++;
```

Answer: This has a loop and the number of iterations is n.
The running time = O(n).

---

6. Write a running time expression for the following code segment:

```
sum = 0;
for (i=0; i<n; i++)
  for(j=0;j<n;j++)
        sum++;
```

Answer: This has a nested loop. Sizes n and n.
The running time = $O(n^2)$.

---

7. Write a running time expression for the following code segment:

```
sum = 0;
for (i=0; i<n; i++)
  for(j=0;j<n*n; j++)
        sum++;
```

Answer: This has a nested loop. Size of the outer loop is n. Notice that the inner loop has size n^2.
The running time = $O(n^3)$.

---

8. Write a running time expression for the following code segment:

```
sum = 0;
for (i=1; i<=n; i++)
  for(j=1;j<=i; j++)
        sum++;
```

Answer: This has a nested loop. Size of the outer loop is n. Notice that the inner loop has the condition j <= i.

Let us understand how this works.

i in outer loop	Inner loop	Number of times inner loop is executed
i = 1	The inner loop goes from j=1 and ends when j = 2. The loop is executed 1 time.	1
i = 2	The inner loop executes for j=1, j=2 and stops when j=3.	2
i = 3	The inner loop executes for j=1, j=2,j=3 and stops when j=4.	3
i = 4	The inner loop executes for j=1, j=2, j=3, j=4 and stops when j=5.	4
...and so on		
i = n This is the last iteration for outer loop	The inner loop executes for j=1, j=2,...,j=n and stops when j=n+1.	n

Thus the total number of times the statement sum++ is executed is,

$1 + 2 + 3 + ... + n = \frac{n(n+1)}{2} = O(n^2)$

The running time = $O(n^2)$.

---

9. Write a running time expression for the following code segment:

```
sum = 0;

for( i = 0; i < n; i++)

        for( j = 0; j < i*i; j++)

                for( k = 0; k < j; k++)

                        sum++;
```

The outer most loop iterates n times.
The second loop iterates at most n^2 times.
The outer most loop runs n^2 times.
Thus the overall complexity would be $O(n^5)$

---

10. In the earlier examples, we assumed that the statement in a loop was a simple statement with running time of 1 unit. If the statement in a loop is a more complex statement, such as a function call, with more than one unit of running time, then we have to multiply by that factor.
    Consider the following code. Assume the function call fun(i) has a running time of logn.

```
for(i=0; i<n; i++){
   fun(i)
}
```

    Total running time = n (for the loop) * logn (for function call) = nlogn.

11. Write a running time expression for the following code segment:

```
sum = 0; i=1; m=1;
while(m<=n){
  i++;
  m=m+i;
  sum++;
}
```

    Answer:
    We like to estimate how many times the loop is going to be executed.
    The loop is controlled by the value of m. Let us list the values of m in each iteration of the loop:
    Values of i and m before loop starts: i = 1 and m = 1
    To see the pattern, let us focus on the values of m.

Iteration #	**Value of i after i++**	**Value of m After m = m+i**
1	2	1+2
2	3	(1+2)+3
3	4	(1+2+3)+4
4	5	(1+2+3+4)+5
And so on… after k steps		
k	k+1	(1+2+3+…+k)+(k+1)

Let us make the kth step the last step of the loop, that is

1+2+3+….+(k+1) > n

We need to find an expression for k in terms of n:

$\frac{(k+1)(k+2)}{2} > n$

Thus, k = O($\sqrt{n}$)

EXERCISES

1. Order the following function in the increasing order:
   n, logn, 4, 10n, 2n2, n3, n2logn, n2, nlogn
   Place your answer in the boxes below, with expressions with equal complexity in the same box.

2. Order the following function in the increasing order:
   n, $\sqrt{n}, \sqrt{n}\ logn, nlog\sqrt{n},$ nlogn^2, n^{2}logn^2, logn,
   Place your answer in the boxes below, with expressions with equal complexity in the same box.

3. Order the following function in the increasing order:
   n^3, $n\sqrt{n}, \sqrt{n}\ log\sqrt{n}, nlog\sqrt{n},$ nlogn^2, n^{2}logn, loglogn,
   Place your answer in the boxes below, with expressions with equal complexity in the same box.

4. Express each of the following functions in the big-O notation:

   (a) $\frac{n^4+2n^3+14n^2+200n+100}{15n+100}$

   (b) 20n^{2}logn + $\sqrt{n}$ logn + 30n^2

(d) $n^2\log n + n\log(n^2) + 30n + 500$

(e) $\frac{n\sqrt{n}+10n+10}{4n+100}$

(d) $1 + 2 + 3 + 4 + \ldots + n$

(f) $1^3 + 2^3 + 3^3 + 4^3 + \ldots + n^3$
(Hint: Use sum of cubes formula: http://math.stackexchange.com/questions/973242/sum-of-cubes-proof)

5. In each of the following pairs of functions, indicate which runs faster (that is lower running time):
(a) (i) $\log^2 n$ and (ii) $n\log n^2$

(b) (i) $\sqrt{n}\ log\sqrt{n}$ and (ii) $n^2\log n$

(c) (i) $n\sqrt{nlogn}$ and (ii) $n\log n^2$

6. For each of the pairs of functions determine the smallest value of n for which the second function becomes smaller that the first:
(a) n^2, 100n
(b) 2^n, 100n
(c) $\sqrt{n}$, nlogn

7. Write a running time expression for the following code segment:

```
sum = 0;
for (i=0;  i<n;  i++)
  for(j=0; j<n; j++)
     for(k=0; k<n; k++)
          sum++;
```

8. Write a running time expression for the following code segment:

```
sum = 0;
for (i=0; i<n*n; i++)
  for(j=0;j<n*n; j++)
          sum++;
```

9. Wrte a running time expression for the following code segment:

```
sum = 0;
for (i=0; i<√n ; i++)
  for(j=0;j<n; j++)
        sum++;
```

10. Write a running time expression for the following code segment:

```
sum = 0;
for( i = 0; i < n; i++)
        for( j = 0; j < i*i; j++)
                for( k = 0; k < j*j; k++)
                        sum++;
```

11. Write a running time expression for the following code segment. Assume the function call fun(i) has a running time of n^2.

```
for(i=0; i<n; i++){
   fun(i)
}
```

12. Express each of the following polynomials using Horner's method

    (a) $5x^4 + 3x^3 - 7x^2 + 17x - 200$

    (b) $15x^5 - 80x^4 + 22n + 50$