Data Structures and Algorithms

Complexity

Dr. Lina Xu
Lina.xu@ucd.ie

School of Computer Science,
University College Dublin

September 10, 2018

Learning outcomes

After this lecture and the related practical students should...

- understand the three main methods of determining complexity
- be able to time the execution of an algorithm
- be able to estimate the execution time of an algorithm
- be able to understand the different classes of algorithms
- be able to order the different classes of algorithms in terms of running time

Table of Contents

Complexity

We are going to start off by only looking at the first question:

- How fast is my code?

But we are going to look at it another way,

- how long will my code take to execute?

Determining Complexity

The **time complexity** is the **computational complexity** that describes the amount of time it takes to run an algorithm.
There are different ways to find out how long it will take to run a piece of code. More specifically we are really looking at the complexity of this code. These methods are:

- Run it and find out
- Read the code and guess
- Analyse the code and its structure

Table of Contents

Timing your code

Most programming languages include some functionality for measuring time.

- This will often require us to find out the time before and after something has executed
- In C this functionality can be accessed by including the library `sys/time.h`
- The function `gettimeofday` can be used to get a precise measurement of time

Timing your code
gettimeofday

The gettimeofday function gets a precise measurement of the current time

- The return type of the function is a struct named timeval
- In order to correctly find the amount of time that passes while a section of code is executed we must know the time value before it is executed
- We then subtract this value from the time value after the code is executed

timeval

The timeval struct has two data members

- tv_sec - the number of whole seconds (long int)
- tv_usec - the number of microseconds (long int)

Timing your code

gettimeofday

- To get a value in seconds for how long your code took to execute, we need to subtract the start time from the end time
- This requires two steps because there are two parts to the time `tv_sec` and `tv_usec`
- For seconds e.g. `double time = end.tv_sec - start.tv_sec`
- For microseconds e.g. `time = time + (end.tv_usec - start.tv_usec) / 1000000.0`

Timing your code

Example

```
#include <stdio.h>
#include <sys/time.h>

int  main(){
   struct timeval start, end;
   gettimeofday(&start, NULL);
   functionToTime();
   gettimeofday(&end, NULL);
   double seconds = (end.tv_sec -
    start.tv_sec);
   seconds = seconds + (end.tv_usec -
    start.tv_usec)/ 1000000.0;

   printf("Code took %f s \n" ,  seconds );
}
```

Comparing Code

We will write two different functions to calculate the sum to `N`

- 1 + 2 + 3 + ... + N - 1 + N

We will then time both of these functions and see which is faster

Comparing code

First function

This function will loop all the way up to N, summing the values all the way

```
long long sumN1( long long  n){
  long long s = 0;
  long long j = 0;
  while (j < n){
    s = s + (j + 1);
    j = j + 1;
  }
  return  s;
}
```

Comparing code

Second function

This function will exploit a mathematical trick to calculate the answer

```
long long sumN2( long long  n){
  long long s = n * (n + 1)/2;
  return  s;
}
```

Comparing code

Comparison code

```
int  main(){
  struct timeval  tv1, tv2, tv3;
  long long number;
  scanf("%llu", &number);
  gettimeofday(&tv1, NULL);
  sumN1(number);
  gettimeofday(&tv2, NULL);
  sumN2(number);
  gettimeofday(&tv3, NULL);
  double time1 = ((tv2.tv_usec - tv1.tv_usec) /
   1000000.0 + (double) (tv2.tv_sec - tv1.tv_sec));
  double time2 = ((tv3.tv_usec - tv2.tv_usec) /
   1000000.0 + (double) (tv3.tv_sec - tv2.tv_sec));
  printf ("sumN1 = %f seconds\n", time1);
  printf ("sumN2 = %f seconds\n", time2);
}
```

Results

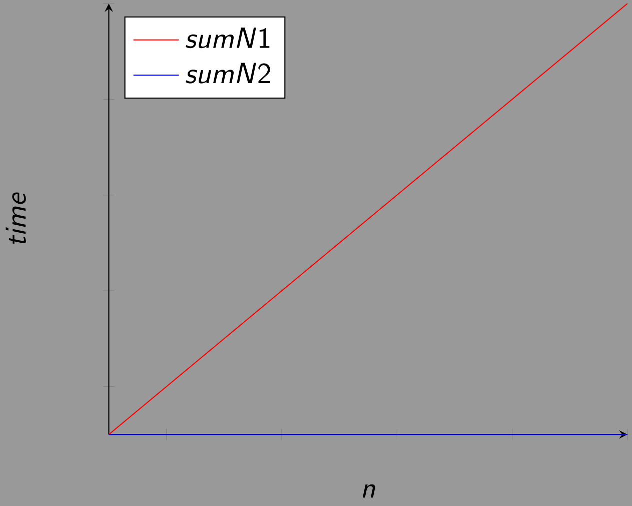


Table of Contents

Problem with timing code

There are quite a few problems with timing code

- Different computers execute code at different speeds
- The operating system may execute other programs at the same time, which can change the result

Overall this is not a very good system when we want to compare code execution times

Estimating running time

To solve these issues we are instead going to estimate the execution time of code.

- We are going to calculate the time it will take to execute on an imaginary computer
- We can decide the amount of time that each operation take to complete
- Then we can calculate the estimated running time

This is not very good for the real world, but excellent for comparing code

Estimating running time

This is just one example....

Estimations will be based on calculations in nano seconds:

Operator	Time (ns)	Example
Mathematical operators	10 ns	`+, -, *, /, %`
Comparison operators	10 ns	>, <, ==, >=, <=, !=
Assignment	10 ns	`=`
Calling function	50 ns	`func()`
Function with parameter	$50 + (10 * p)$[1]	`func(1,2)`
Returning a value	50 ns	`return 50;`
Creating an array	100 ns	`int array[60];`
Accessing an array	50 ns	`array[6]`
Boolean operators	10 ns	`&&, ||, !`

They are all **Constant Values**!

The important thing is to find out **How Many Times** they will be executed.

[1]Where p = number of parameters

Calculated running time

Code	Line cost	Total cost	Time Complexity
`x = x + 1`	$10 + 10$	20	Constant
`x == 2 || x == 5`	$10 + 10 + 10$	30	Constant
`x = x + y`	$10 + 10$	20	Constant
`y = x - y`	$10 + 10$	20	Constant
`z = x + y`	$10 + 10$	20	Constant

Calculated running time

Code	Line cost	Total cost
`long long sumN1(long long n){`	50 + 10	150
`  long long s=n*(n+1)/2;`	10 + 10 + 10 + 10	
`  return s;`	50	
`}`		

Table of Contents

Conditional Statements

When using conditional statements like `if` and `while`, the amount of time take to execute code may change

- When dealing with this we always assume that the worst case will happen
- If there are two options we choose the longest one

Conditional Statements

IF Statement

If we have an if statement like this:

```
if(booleanExpression){
  statement1;
} else {
  statement2;
}
```

Then the total cost of execution is the cost of boolean Expression + the largest cost of statement1 or statement2.

Conditional Statements

IF Statement

To calculate the cost of the following code:

```
if(x > 0){
  x = 1;
} else {
  x = x * x;
}
```

Then the total cost of execution is the cost of `x > 0` (10) + the largest cost of `x = 1` (10) or `x = x * x` (20).

This gives us a total cost of 30.

Conditional Exercises

Code	Line cost	Total cost
`if(f==0 || g==1){`	$10 + 10 + 10$	50
`  g = 55 * g;`	$10 + 10$	
`}`		
`if((f+1) > 0){`	$10 + 10$	$20 + \max(20, 10)$
`  g = 55 * g;`	$10 + 10$	$= 40$
`} else {`		
`  f = 1;`	10	
`}`		

Table of Contents

Loop Statements

The cost of executing loop statements is the sum of the following

- The cost of evaluating the loop condition multiplied by `(n + 1)`
- The cost of executing the statements in the body of the loop multiplied by `n`

Here `n` is the number of times that the loop will be executed (also called the number of iterations)

Loop Statements

If we have a while loop like this:

```
while(booleanCondition){
  statements;
}
```

Then the total cost of executing the loop is

- the cost of evaluating booleanExpression multiplied by `(n + 1)`
- added to the cost of executing statements multiplied by `n`

Loop Example

To calculate the cost of this code:

```
int i = 0;
while(i < 5){
  x = x * x;
  i = i + 1;
}
```

- First we calculate the cost of the loop condition 10
- Then we calculate the cost of the loop body 40
- The cost of the loop condition is then multiplied by `n + 1` and the cost of the loop body is multiplied by `n` `10*(n+1)+40*n`
- Finally if we know the value for `n` we can calculate the correct time `10 * (5 + 1) + 40 * 5 = 260`

Loop Exercise

Code	Line cost	Total cost
`long s = 0;`	10	$60 * n + 30$
`int j = 0;`	10	$= 60 * 100 + 30$
`while(j < 100){`	$10 * (n + 1)$	$= 6030$
`  s = s + (j+1);`	$n * 30$	
`  j = j + 1;`	$n * 20$	
`}`		

Here we can see that the loop will execute 100 times

Function Exercise

Code	Line cost
`long long sumN2(long long n){`	50 + 10
`   long long sum = 0;`	10
`   long long j = 0;`	10
`   while(j < n) {`	(n + 1) * 10
`      sum = sum + j + 1;`	n * 30
`      j = j + 1;`	n * 20
`   }`	
`   return sum;`	50
`}`	

Here we can not tell how many times that the loop will be executed.
It will depend on the value passed into the function
Therefore our answer will have to have n in it
`= 70 * n + 140`

Changing Values

In the previous example our final answer still had `n` in it. When this is the case the running time will change depending on the size of `n`

- When comparing algorithms, it is often helpful to use a graph to show the different running times for different values
- Lets compare the running times for the functions `sumN1` and `sumN2`

Comparing sumN1 and sumN2

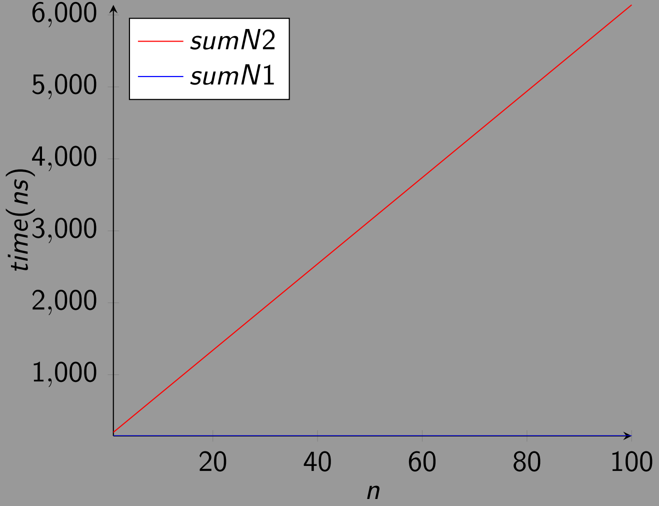


Table of Contents

Nested Loops

- A nested loop is when we have one loop inside another
- If we have two loops, an outer loop and an inner loop (the inner loop is inside the outer loop), then the inner loop will be executed fully ever time the outer loop is executed once
- There are two steps to calculating the running time of nested loops
    1. Use a constant to represent the inner loop and calculate the running time of the outer loop
    2. Calculate the actual running time of the inner loop and replace the constant with our result

Nested Loop Exercise

Code	Line cost
`int n = 10;`	10
`int f[n][n];`	100
`int a = 0;`	10
`while(a < n){`	$(n+1) * 10$
`    int b = 0;`	$n * 10$
`    while(b < n){`	**n * k (whole loop)** $(n+1) * 10$
`      f[a][b] = 1;`	$n * 60$
`      b = b + 1;`	$n * 20$
`    }`	
`    a = a + 1;`	$n * 20$
`}`	

Nested Loop Exercise

Total running time

- The overall running time of this code was `time = 10+100+10+(n+1)*10+n*10+n*k+n*20`
- `time = k * n + 40 * n + 130`
- The running time of the inside loop was `k = 90 * n + 10`

Nested Loop Exercise

Total running time

- When we replace k with the value we calculated we get
  - Running time = `40 * n + (90 * n + 10) * n + 130`
  - `= 40 * n + 90 * n * n + 10 * n + 130`
  - $= 90 * n^2 + 50 * n + 130$

Again the running time changed depending on the value of n

If `n = 10,000` the running time will be

- `= 90 * 100000000 + 50 * 10000 + 130`
- `= 9000500130 nano seconds`
- `= 9.00 seconds`

What is the running time if `n = 10,000,000`?

- `9000000500000130 nanoseconds`
- About 15 years...

Table of Contents

Classes of algorithms

When analysing the complexity of code, algorithms can be grouped together into classes where all the algorithms in that class have similar performance

- Constant time
  - This is where the running time never changes (the result does not contain n)
- Linear time
  - This is where the running time gets longer at the same time as n gets bigger (There is an n in our answer but there is no exponent)
- Quadratic time
  - This is when the running time gets bigger at a much faster rate than n gets bigger (specifically where the answer contains n^2)

Comparing classes of algorithms

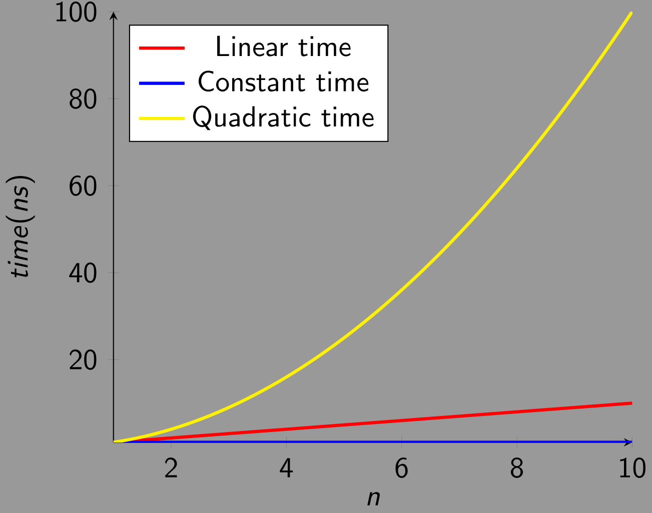


Table of Contents

Logarithmic time

The next class of algorithms we will look at is logarithmic time

- Logarithmic time algorithms are the hardest to analyse
- You have to really understand the source code to properly identify them

Logarithmic time

The running time of some algorithms change based on the value of `n`, but not in a linear way. In these cases, the running time is instead based on the log of `n`

- These algorithms will usually divide the problem in half with every step
- Binary search is an example of a logarithmic algorithm

Logarithmic algorithm

What is the cost of executing the following loop?

```
int k = 1024;
while(k > 1){
  // loop body here
  k = k/2;
}
```

- Every iteration of the loop `k` is divided by 2
- k starts at 1024
- After 1 Iteration: `k = 1024 / 2 = 512.` After 2 Iterations: `k = 512 / 2 = 256.` After 3 Iterations: `k = 256 / 2 = 128.` After 4 Iterations: `k = 128 / 2 = 64.` After 5 Iterations: `k = 64 / 2 = 32.` After 6 Iterations: `k = 32 / 2 = 16.` After 7 Iterations: `k = 16 / 2 = 8.` After 8 Iterations: `k = 8 / 2 = 4.` After 9 Iterations: `k = 4 / 2 = 2.` After 10 Iterations: `k = 2 / 2 = 1.`

The number of times a logarithmic algorithm executes can be calculated using a log e.g. $log_2 1024 = 10$

Calculating number of iterations of logarithmic algorithms

We have seen that when n is a power of 2 the number of times that a logarithmic algorithm will execute is $log_2 n$

- But what if n is not a power of 2?, e.g. n = 1056
- This can be calculated using the floor of the log of n

$$\lfloor Log_2(n) \rfloor$$

- We can use this to find the number of iterations for any logarithmic algorithm

$$\lfloor Log_2(2000) \rfloor = 10$$

What is n?

Figuring out what running time of an algorithm often means that we are dealing with some unknown variable `n`. How can we know what value represents `n`?

- This requires a good understanding of the code you are analysing
- Often `n` will be the value of a variable
- Mostly in this course when we talk about `n` we will be talking about the number of items stored in a data structure

Binary Search

Binary search is an excellent example of a logarithmic algorithm:

- We wish to search an array to see if it contains a value `x`
- The values stored in the array are sorted in ascending order
- Instead of checking from the start we can start in the middle
- If the number in the middle is bigger than `x` then `x` cannot be any higher than the middle (all of these numbers will be bigger than `x`)
- If the number in the middle is smaller than `x` then `x` cannot be any lower than the middle (all of these numbers will be smaller than `x`)
- This eliminates half of the numbers in the array every time

Binary Search

Code	Line cost
`int bSearch( ( int f[], int n, int x){`	$50 + 30$
`    int j = 0;`	10
`    int k = n;`	10
`    while (j + 1 != k){`	$(L + 1) * 20$
`       int i = (j + k)/2;`	$L * 30$
`       if ( x >= f[i]){`	$L * 60$
`          j = i;`	$L * 10$
`       } else {`	
`          k = i;`	$L * 10$
`       }`	
`    }`	
`    return (f[j]==x);`	$50 + 50 + 10$
`}`	

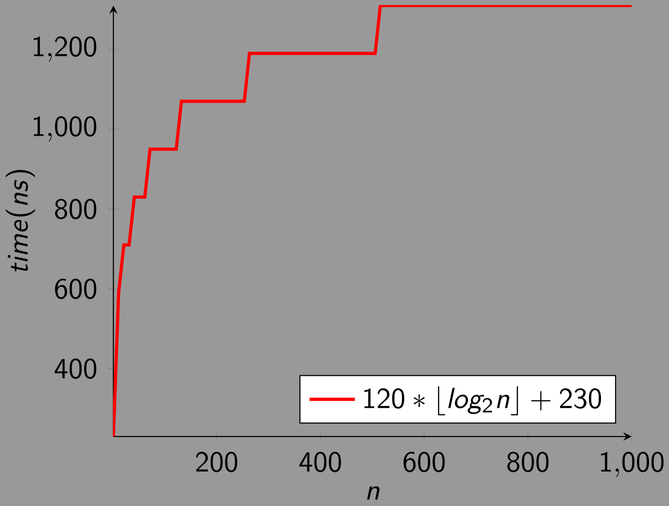
Here L is the number of times the loop executes

Binary Search

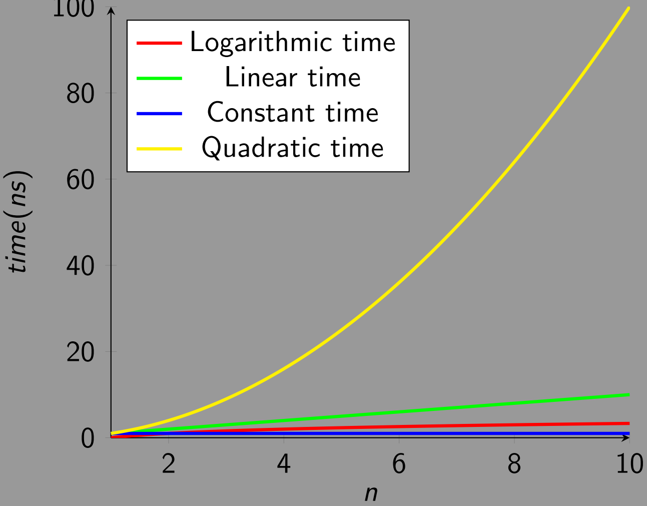
Result

Adding all of the lines together we get:
$60 * L + max(L * 10, L * 10) + L * 30 + (L + 1) * 20 + 210$
$= 120 * L + 230$
If we work out the value of L we get $L = \lfloor log_2 n \rfloor$
So we end up with $120 * \lfloor log_2 n \rfloor + 230$

Binary Search Running time

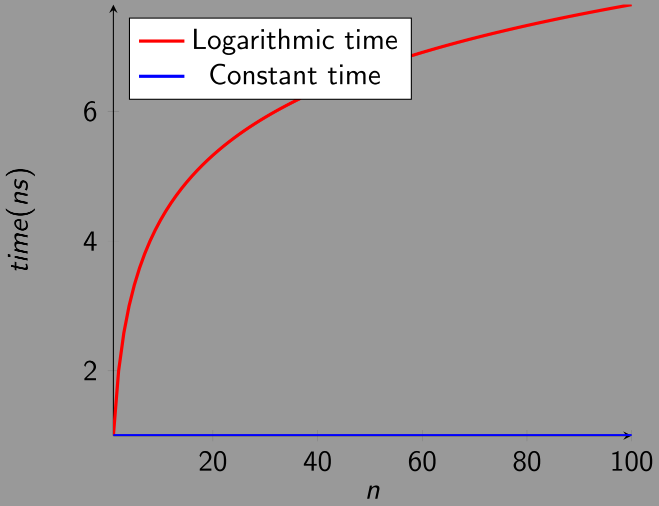


Comparing Algorithm Running times



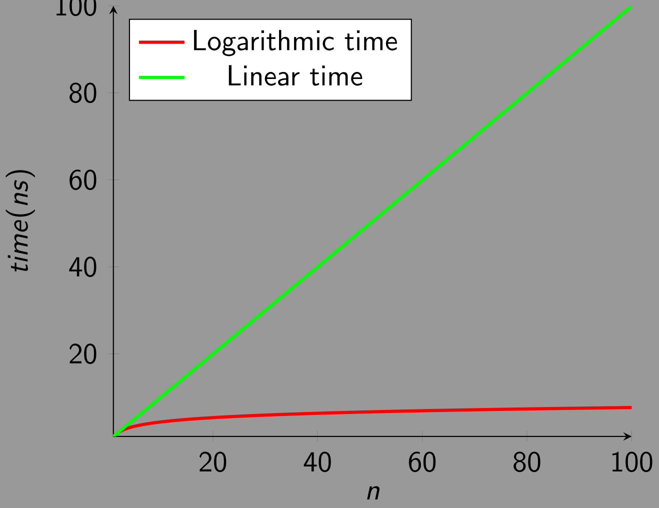
Comparing Algorithm Running times

Constant time and Logarithmic time



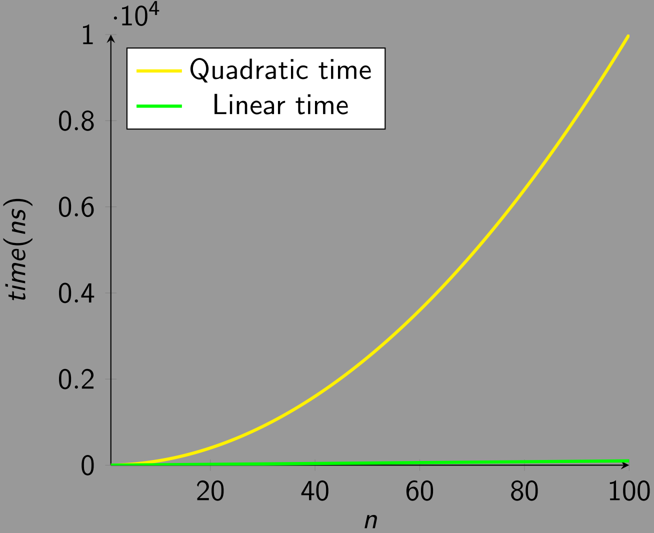
Comparing Algorithm Running times

Logarithmic time and Linear time



Comparing Algorithm Running times

Linear time and Quadratic time


$\cdot 10^4$
time(ns)
Quadratic time
Linear time
n