Algorithm Analysis

Time Complexity

Real Time:

To analyze the real time complexity of a program we need to determine two numbers for each statement in it:

- amount of time a single statement will take.
- No. of times it is executed.
- Product of these two, will be the total time taken by the statement.

First no. depends upon the machine and compiler used , hence the real time complexity is machine dependent.

Frequency count

- To make analysis machine independent
  it is assumed that every instruction takes the same constant amount of time for execution.

- Hence the determination of time complexity of a program is the matter of summing the frequency counts of all the statements.

Linear loop

```
1   i=1
2   Loop(I <= 1000)

        1       application code
        2       I = I  +  1
```

The body of the loop is repeated 1000 times.

```
1   I = 1
2   Loop (I <= 1000)
    1   Application code
    2   I = I + 2
```

For this the code time is proportional to n

Logarithm Loops

Multiply loops

1 I = 1

2 Loop (I <1000)

1 application code code

2 I = i*2

F(n) = [log n]

Divide loops

1 I = 1000

2 loop(i>= 1)

1 application

2 I=I/2

F(n) = [log n]

Nested loop- linear logarithmic

- **1 I = 1**
- **2 loop(I <= 10)**

1 j = 1

2 loop(j < = 10)

1 application code

2 j = j *2

3 I = I + 1

F(n) = [n log n]

Dependent Quadratic

```
 1    I = 1
2     loop ( I < = 10)
      1       j = 1
      2       loop( j  < =  i)
              1       application code
              2       j = j + 1
      3       I = I  +  1
```

no of iterations in the body of the inner loop is

1 + 2 + 3 + 4 +… + 10 = 55 I.e. n(n +1)/2

On an average = (n+1/)2

thus total no of iterations = n (n+1)/2

Quadratic

1 i=1

2 Loop (I < = 10)

1 j = 1

2 Loop(j < = 10)

1 application code

2 j = j+1

3 I = i+1

$$F(n) = n^2$$

Insertion Sort

	Freq.
`for j ← 2 to length[A]`	n
`do  key ← A[j]`	$n-1$
`* insert A[j] into sorted array`	
`  * sequence A[1….j-1]`	
`      I ←  j-1`	$n-1$
`     while I> 0 and A[I] > key`	
`          Do A[I+1] ← A[I]`	$\sum t_j$
`          I ←  I-1`	$\sum(t_j-1)$
`   A[I+1] ← key`	$n-1$

$$T(n) = c_1 n + c_2 (n-1) + c3(n-1)+ c_4 \sum t_j + c_5 \sum(t_j-1) + c_6 \sum(t_j-1) + c_7 (n-1)$$

line	pseudocode	cost	times
1	// At start, the singleton sequence A[1] is trivially sorted	$c_1 = 0$	
2	for j = 2 to n	c_2	n
3	// Insert A[j] into the sorted sequence A[1..(j 1)]	$c_3 = 0$	
4	do key = A[j]	c_4	$n – 1$
5	i = j - 1	c_5	$n – 1$

// let tj be the number of times the following //while-loop is tested for the value j		$c_6 = 0$
while (i > 0 and A[j] > key)	c_7	$\sum_{j=2}^{n} t_j$
do A[i+1] = A[i]	c_8	$\sum_{j=2}^{n} (t_j-1)$
i = i - 1	c_9	$\sum_{j=2}^{n} (t_j-1)$
A[i + 1] = key	c_{10}	$n - 1$

Worst/ best/average cases

- **Worst case is the longest running time for any input of size n**
- **O-notation represents upper bound I.e. an upper bound for worst case.**
- **Best case is the input data set that results in best possible performance.You cannot do better. This is lower bound.**
- **Average case is the average performance**

Big O notation

- *Informally,* **Time to solve a problem of size, n,**

$$T(n) \quad \text{is} \quad O(\log n)$$

$$\Leftarrow T(n) = c \log_2 n$$

- ***Formally:***
    - $O(\,g(n)\,)$ **is the set of functions,** f**, such that**

$$f(n) < c\,g(n)$$

for some constant, $c > 0$**, and** $n > N$

ie for sufficiently large n

- **Alternatively, we may write**

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} \leq c$$

and say

g ***is an upper bound for*** f

- A non negative function $T(n)$ is said to be $O(f(n))$ provided there exists a constant $c>0$ and an integer $n_0 > 0$ such that
- $T(n) \leq c.\ f(n)$ for all integer $n > n_0$
-
- Consider $1/3\ n^2 – 5n$
- The dominating term is n^2
- Therefore it should be of $O(n^2)$
- Given a positive constant c , a positive integer n_0 to be found such that

- $1/3\ n^2 \ - 5\ n \leq c\ n^2$

- Dividing the inequality throughout by n^2

- $1/3 - 5/n \leq c$

- Therefore if $n_0 \geq 1$, choosing $c \geq 1/3$ the inequality will never be violated
- Hence the expression is indeed of $O(n^2)$

- Suppose $T(n) = 1/3\ n^3 + ½\ n^2 + 1/6\ n$
- So $T(n)$ is of order$(1/3n^3)$ which can be proved as follows

- $1/3\ \ n^3 + ½\ \ n^2\ \ + 1/6\ n \leq c\ 1/3\ n^3$
- $1/3\ \ + 1/(2n) + 1/(6n^2) \leq\ \ c/3$
- Inequality is valid for case $c\ \ \geq 3$ and integers $n \geq 1$

- $O(g)$
  - the set of functions that grow no faster than g.
- $g(n)$ **describes the worst case behaviour of an algorithm that is** $O(g)$

- $O(g)$

  - the set of functions that grow no faster than g.
- $g(n)$ describes the worst case behaviour of an algorithm that is $O(g)$
- Two additional notations
- $\Omega(g)$

  - the set of functions, f, such that

$$f(n) > c\, g(n)$$

for some constant, c, and $n > N$

g is a lower bound for f

- $O(\,g\,)$

  - the set of functions that grow no faster than g.
- **$g(n)$ describes the worst case behaviour of an algorithm that is $O(\,g\,)$**
- **Two additional notations**
- $\Omega(\,g\,)$

  - the set of functions, f, such that

  $$f(n) > c\,g(n)$$

  for some constant, c, and $n > N$

  g is a lower bound for f
- $\Theta(\,g\,) = O(\,g\,) \cap \Omega(\,g\,)$

  Set of functions growing at the same rate as g

Properties of the O notation

- **Constant factors may be ignored**
  - $\forall\ k > 0$, kf is $O(f)$

Properties of the O notation

- **Constant factors may be ignored**
  - $\forall\ k > 0,\ kf$ is $O(f)$
- **Higher powers grow faster**
  - n^r is $O(n^s)$ if $0 \le r \le s$

Properties of the O notation

- **Constant factors may be ignored**
  - $\forall\ k > 0,\ kf$ is $O(f)$
- **Higher powers grow faster**
  - n^r is $O(n^s)$ if $0 \le r \le s$

← **Fastest growing term dominates a sum**

- If f is $O(g)$, then $f + g$ is $O(g)$

  eg $an^4 + bn^3$ is $O(n^4)$

Properties of the O notation

- **Constant factors may be ignored**
  - $\forall\ k > 0,\ kf$ **is** $O(f)$
- **Higher powers grow faster**
  - n^r **is** $O(n^s)$ **if** $0 \le r \le s$

← **Fastest growing term dominates a sum**

- **If** f **is** $O(g)$, **then** $f + g$ **is** $O(g)$

  $eg\quad an^4 + bn^3$ **is** $O(n^4)$

← **Polynomial's growth rate is determined by leading term**

- **If** f **is a polynomial of degree** d**,**
  then f **is** $O(n^d)$

Properties of the O notation

- f is $O(g)$ is transitive
  - If f is $O(g)$ and g is $O(h)$ then f is $O(h)$

Properties of the O notation

- f is $O(g)$ is transitive
  - If f is $O(g)$ and g is $O(h)$ then f is $O(h)$
- Product of upper bounds is upper bound for the product
  - If f is $O(g)$ and h is $O(r)$ then fh is $O(gr)$

Properties of the O notation

- f is $O(g)$ is transitive
  - If f is $O(g)$ and g is $O(h)$ then f is $O(h)$
- Product of upper bounds is upper bound for the product
  - If f is $O(g)$ and h is $O(r)$ then fh is $O(gr)$
- Exponential functions grow faster than powers
  - n^k is $O(b^n)$ $\forall$ $b > 1$ and $k \geq 0$
    eg n^{20} is $O(1.05^n)$

Properties of the *O* notation

- f is $O(g)$ is transitive
  - If f is $O(g)$ and g is $O(h)$ then f is $O(h)$
- Product of upper bounds is upper bound for the product
  - If f is $O(g)$ and h is $O(r)$ then fh is $O(gr)$
- Exponential functions grow faster than powers
  - n^k is $O(b^n)$ $\forall$ $b > 1$ and $k \geq 0$
    eg n^{20} is $O(1.05^n)$
- Logarithms grow more slowly than powers
  - $\log_b n$ is $O(n^k)$ $\forall$ $b > 1$ and $k > 0$
    eg $\log_2 n$ is $O(n^{0.5})$

Properties of the O notation

- f is $O(g)$ is transitive
  - If f is $O(g)$ and g is $O(h)$ then f is $O(h)$
- Product of upper bounds is upper bound for the product
  - If f is $O(g)$ and h is $O(r)$ then fh is $O(gr)$
- Exponential functions grow faster than powers
  - n^k is $O(b^n)$ $\forall$ $b > 1$ and $k \geq 0$
    eg n^{20} is $O(1.05^n)$
- Logarithms grow more slowly than powers
  - $\log_b n$ is $O(n^k)$ $\forall$ $b > 1$ and $k > 0$
    eg $\log_2 n$ is $O(n^{0.5})$

Important!

Properties of the O notation

- **All logarithms grow at the same rate**
  - $\log_b n$ **is** $O(\log_d n)$ $\forall$ $b, d > 1$

Properties of the O notation

- All logarithms grow at the same rate
  - $\log_b n$ is $O(\log_d n)$ $\forall$ $b, d > 1$

- Sum of first n r^{th} powers grows as the $(r+1)^{th}$ power
  - $\sum_{k=1}^{n} k^r$ is $\Theta(n^{r+1})$

$$eg \quad \sum_{k=1}^{n} i = \frac{n(n+1)}{2} \quad \text{is } \Theta(n^2)$$

Polynomial and Intractable Algorithms

- **Polynomial Time complexity**
  - **An algorithm is said to be polynomial if it is $O(n^d)$ for some integer d**
  - **Polynomial algorithms are said to be efficient**
    - **They solve problems in reasonable times!**

- **Intractable algorithms**
  - **Algorithms for which there is no *known* polynomial time algorithm**
  - ***We will come back to this important class later in the course***

Analysing an Algorithm

- **Simple statement sequence**

  ```
  s1; s2; …. ; sk
  ```

  - $O(1)$ as long as k is constant
- **Simple loops**

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

  where s is $O(1)$

  - Time complexity is $n\,O(1)$ or $O(n)$
- **Nested loops**

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

  - Complexity is $n\,O(n)$ or $O(n^2)$

This part is $O(n)$

Analysing an Algorithm

- **Loop index doesn't vary linearly**

```
h = 1;
while ( h <= n ) {
      s;
      h = 2 * h;
      }
```

  - **h takes values $1, 2, 4, \ldots$ until it exceeds n**
  - **There are $1 + \log_2 n$ iterations**
  - **Complexity $O(\log n)$**

Analysing an Algorithm

- **Loop index depends on outer loop index**

```
for(j=0;j<n;j++)
   for(k=0;k<j;k++){
      s;
   }
```

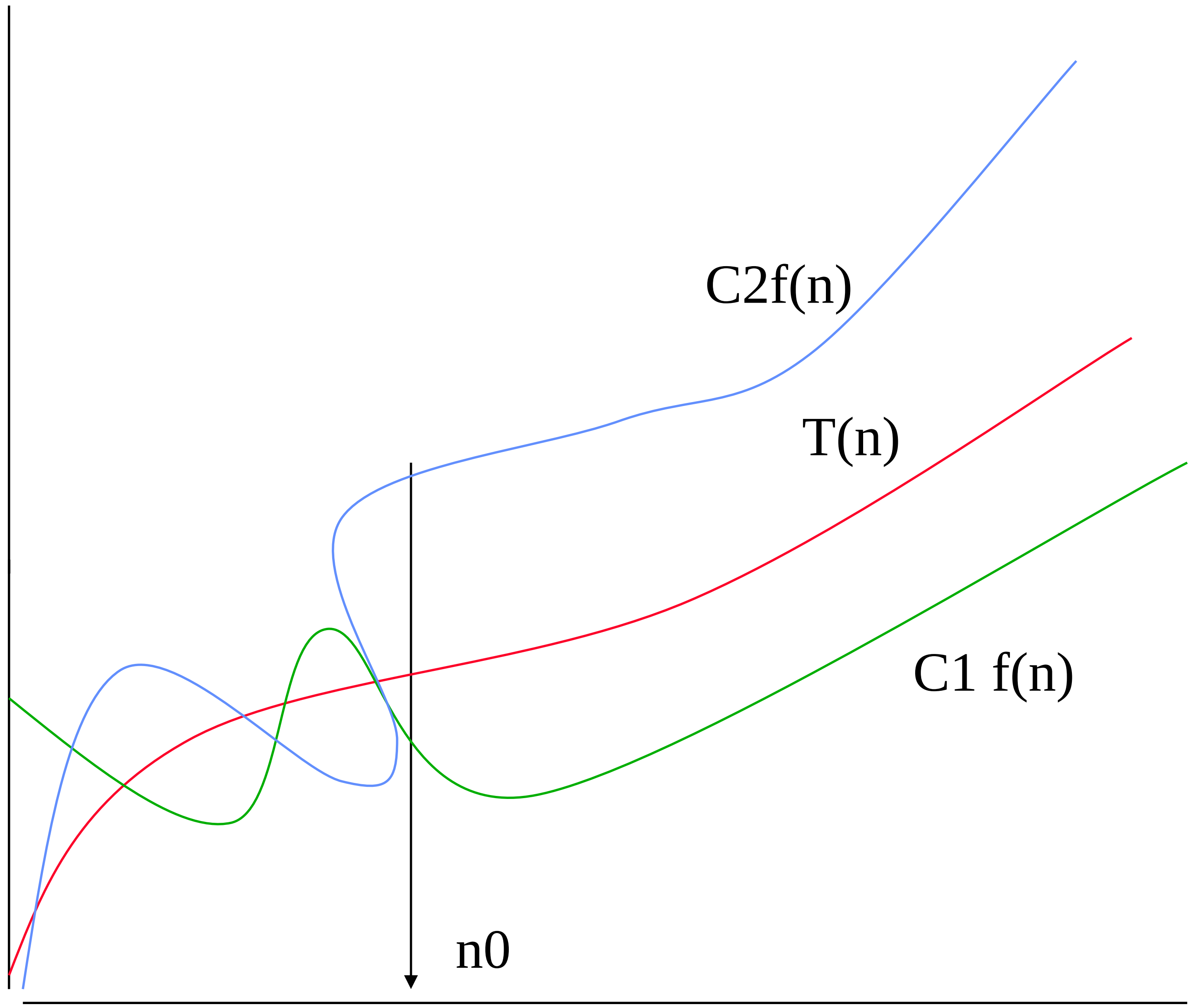
- **Inner loop executed**
  - **1, 2, 3, …., n times**

$$\sum_{i=1}^{} i = \frac{n(n+1)}{2}$$

$\therefore$ **Complexity** $O(n^2)$

u

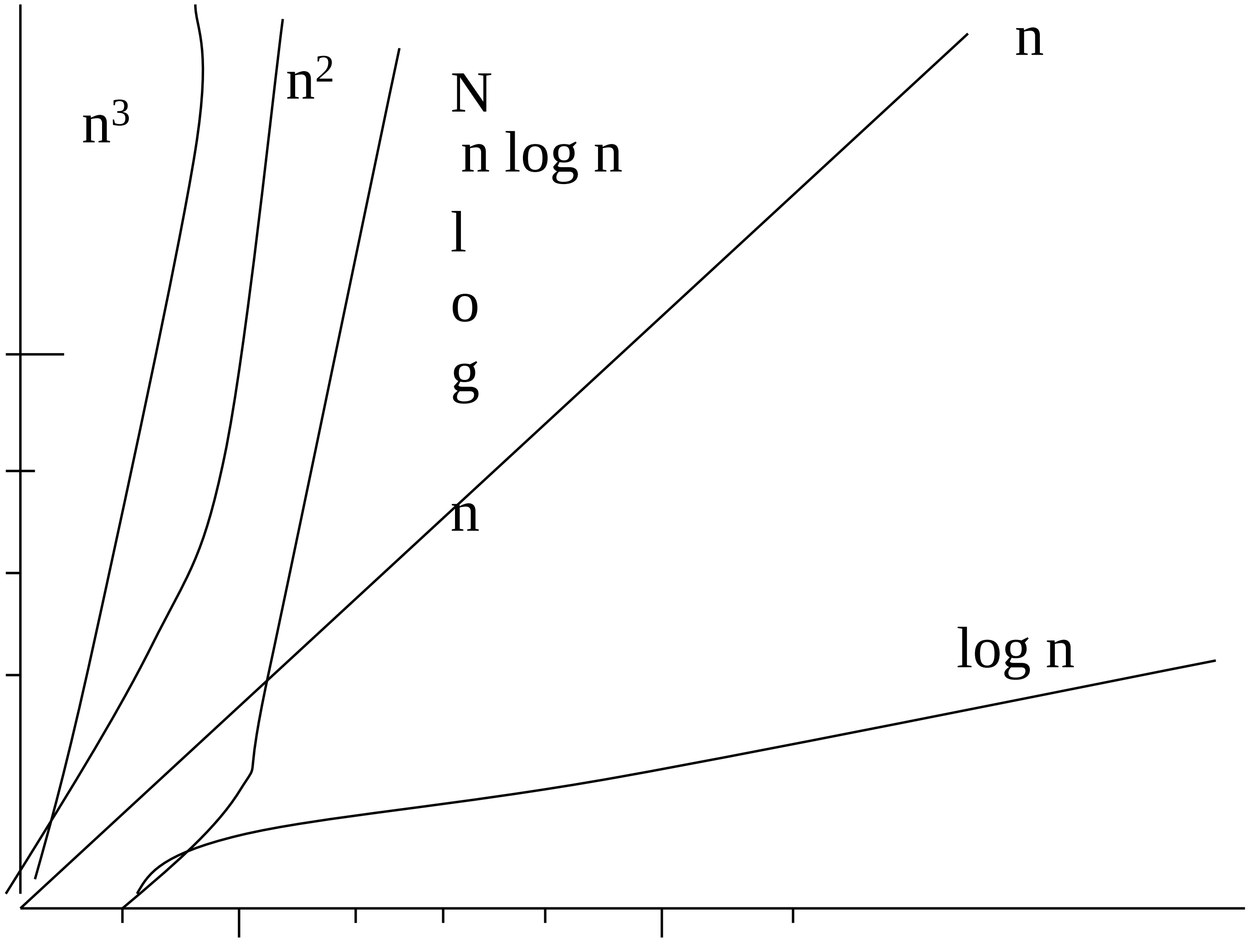
Distinguish this case - where the iteration count increases (decreases) by a constant ← $O(n^k)$ from the previous one - where it changes by a factor ← $O(\log n)$



C2f(n)
T(n)
C1 f(n)
n0

Seven categories of algorithm efficiency

efficiency	Big O
Logarithmic	$O(\log n)$
Linear	$O(n)$
Linear logarithmic	$O(n \log n)$
Quadratic	$O(n^2)$
Polynomial	$O(n^k)$
Exponential	$O(c^n)$
Factorial	$O(n!)$



n
n2
n3
N
n log n
l
o
g
n
log n

- Q1 Find functions f and g such that
- (i) f is (g)
- (ii) f is not Θ (g)
- (iii) f(n) > g(n) for infinitely many n.

- Let us take G(n) = ½ for all n
- F(n) be a function with values in(0,1) close to 1.
- F(n) = sin(n) or
- F(n) = 1/n if n is perfect square and 1 otherwisw.