Algorithmic Complexity - I

Atul Gupta

ES103: Course Summary

- Estimating algorithmic complexities
- Tools like Recursion, Iteration
- ADT
  - Linear data organizations like Arrays, Stacks, queues, Linked lists
  - Non-linear data organizations like hashing, trees (and its many variances), graphs (and many variances)
  - Examples
- Standard Problems: Searching and Sorting
- Standard Problem Solving Approaches

Algorithm

"An algorithm is a sequence of computational steps that transform the input into the output"

An Example: **Sorting Problem**
Input: A sequence of n numbers a_1 , a_2 , ..., a_n
Output: Output: A permutation (reordering) {a_1' , a_2' , ..., a_n'} of the input sequence such that $a_1' <= a_2' <=$, ..., $<=a_n'$

Algorithms as a tool

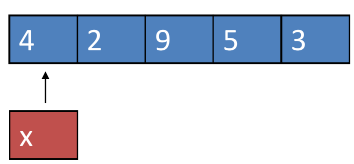
- What kinds of problems are solved by Algorithms?
  - Internet
  - Operating Systems
  - Machine Learning
  - Design and Manufacturing
  - E-Commerce
  - Workflows
  - …

Algorithm

- Can be expressed in many kind of notations, like
  – Natural languages,
  – Pseudocode,
  – Flowcharts,
  – Programming Languages

Search

Example: **Searching Problem**
Input: A sequence of n numbers a_1 , a_2 , ..., a_n and an element x to be search
Output: Output: return the index of the element x if present, else, return 'not found'



Sequential Search – C Code

```
int sequentialSearch(int c[ ], int s_element){

    int i;
    int array_ size = size_of_array(c);
    for (i=0; I < array_size  and s_element != c[i]; i++);

    if (i == array_size)
        return -1;
     else
        return i;
  }
```

```
int sequentialSearch(int c[ ], int s_element){

    int i;
    int array_ size = size_of_array(c);
    for (i=0; I < array_size; i++){
        if (s_element = c[i])
            break;
     }
     if (i == array_size)
         return -1;
        else
         return i;
  }
```

Sequential Search – C Code

```
int sequentialSearch(int c[ ], int s_element){

    int i;
    int array_ size = size_of_array(c);
    for (i=0; I < array_size  and s_element != c[i]; i++);
        if (i == array_size)
           return -1;
        else
           return i;
  }
```

Algorithm

```
int sequentialSearch(x, A)
int i;
for (i=0; i < length[A] and x <> A[i]; i++);
   if (i == length[A])
      return (-1)
   else
      return (i)
```

Sequential Search – Analysis

Code	Cost
`int sequentialSearch(x, A)`	1
`int i;`	1
`for (i=0; i < length[A]; i++) {`	1 - length[A]
`    if x == A[i]`	0 - length[A]
`        then break;`	1
`}`	
`if (i == length[A])`	1
`    return (-1)`	1
`else`	1
`    return (i)`	1

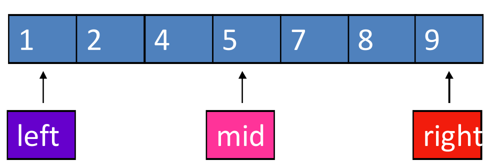
Best Case : when x is the first element (No of comparisons = 1) : O(1)

Worst Case : when x is absent (No of comparisons = n) : O(n)

Average Case : $\frac{1}{n}\sum_{i=1}^{n} i = \frac{(n+1)}{2}$ = O(n)

Binary Search

```
BinarySearch(x, A)
left = 0;
right = length(A) - 1;
while (left <= right) {
    mid = (left + right) / 2
    if (A[mid] == x)
        return mid;
    else if (A[mid] > value)
        right = mid - 1
    else
        left = mid + 1
}
return -1 // not found
```

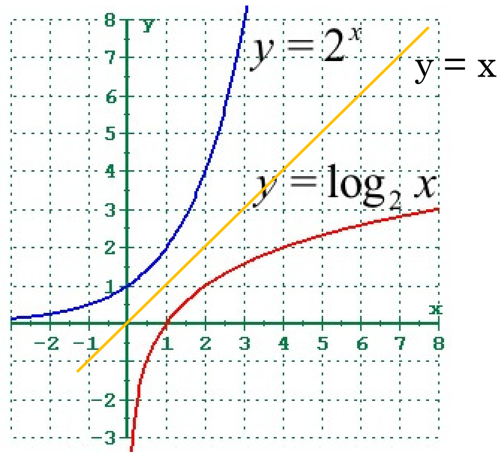


Best Case : when x is mid element (No of comparisons = 1) : O(1)

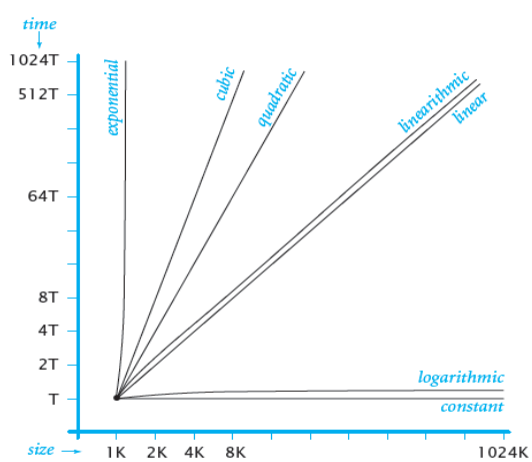
Worst Case : when x is absent (No of comparisons = n) : O(log n)

Average Case : O(log n)

Comparing Performance



Comparing Performance



Orders of growth (log-log plot)

Finding Minimum of a Collection

Code	Time	Space
int findMin(int[] x) {		n
int k = 0; int n = x.length;		1+1
for (int i = 1; i < n; i++) {	n	
if (x[i] < x[k]) {	1	
k = i;	1	
} }		
return k;	1	
}	n+3	n+2
	O(n)	O(n)

C Code for Binary Search

```
int Bin-Search(){
first = 0;
  last = n - 1;
  middle = (first+last)/2;

  while( first <= last )
  {
    if ( array[middle] < search )
      first = middle + 1;
    else if ( array[middle] == search )
    {
      printf("%d found at location %d.\n", search, middle+1);
      break;
    }
    else
      last = middle - 1;

    middle = (first + last)/2;
  }
  if ( first > last )
    printf("Not found! %d is not present in the list.\n", search);

  return 0;
}
```

How good is your program?

- Sequence of computational steps
- Efficient
  - Speed (Time)
  - Space (Memory)
- Evaluation of "Goodness" should be independent of
  - Language
  - Processor
- Algorithmic Evaluation

How good is your program?

- Running Time
  - Types of operations
  - H/W
  - Compiler
  - Inputs
- Number of Basic Operations
  - Different types
  - Inputs and size
  - Complex

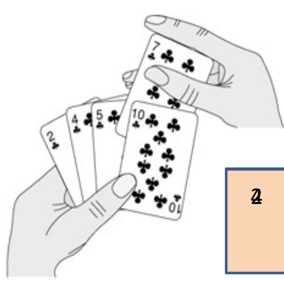
```
scanf("%d", &x);
if (x < 10)
   x++;
else
   for (i =0, i<10, i++);
```

How good is your program?

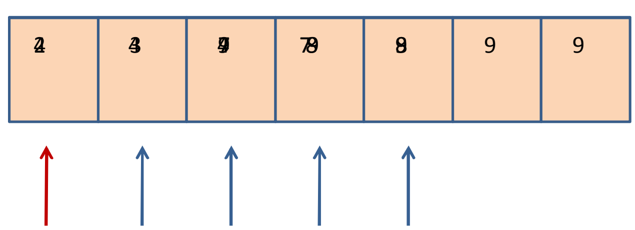
- Running time as a function of input size
  - Input type and size is known
  - Compute number of times each statement is going to be executed (expressed in terms of input size)
  - Replace operation times with constants
    - Different operations take different (but constant) time
  - Compute

    $T(n) = \Sigma$ time taken by i^{th} statement

An Example: Insertion Sort



Sequence is 4, 2, 9, 7, 8, 3, 5



The figure is taken from the Book by Cormen et. al. – Introduction to Algorithms

Algorithm: Insertion sort

```
void insertion-sort(A)
1  for (i ← 2; i < length[A]; i++) {
2        key ← A[i];
         //Insert A[j] into the sorted sequence A[1 ] to A[j - 1].
3        j ← i – 1;
4        while (j > 0 and A[j] > key) {
5              A[j + 1] ← A[j];
6              j ← j – 1;
         }
7        A[j + 1] ← key;
   }
```

Insertion sort: Time Taken

S. No.	Steps	cost	times
1	for (i ← 2; i< length[A]; i++)	c1	n-1
2	key ← A[i];	c2	n-1
	//Insert A[j] into the sorted sequence A[1] to A[j - 1].		
3	j ← i - 1	c3	n-1
4	while (j > 0 and A[j] > key)	c4	$\sum_{j=2}^{n} t_j$
5	A[j + 1] ← A[j];	c5	$\sum_{j=2}^{n} (t_j-1)$
6	j ← j– 1;	c6	$\sum_{j=2}^{n} (t_j-1)$
7	A[j + 1] ← key	c7	n-1

$$T(n) = c1(n-1) + c2(n-1) + c3(n-1) + c4 \sum_{j=2}^{n} t_j + c5 \sum_{j=2}^{n}(t_j-1) + c6 \sum_{j=2}^{n}(t_j-1) + c7(n-1)$$

Insertion Sort: Time Taken

- Best Case
- Worse Case
- Average Case

Why Worse Case?

- Act as the "Upper Bound"
- Worse case do occur in practice
  - Ex: a record not found in the database
- Average case is often (roughly) as bad as the worse case

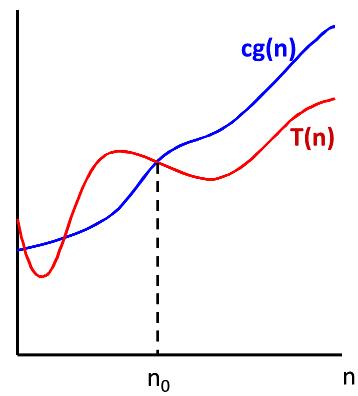
Algorithmic Complexity

- A measure of the performance of an algorithm with respect to input size
  - Space complexity
  - Time complexity (Mostly used)
- Expressed in terms of Asymptotic Notations
  - Bog-Oh (O), Big-Theta (Θ), Big-Omega (Ω)
- What to measure
  - Worst case performance
  - Average case
  - Best case

Big-Oh (O)

- An indicator of the worst case performance (upper bound)
- Defined as a function T(n) which is said to be of O(g(n)) if there exist positive constants c_0 and n_0 such that for all $n >= n_0$, we have

  $T(n) < = c_0 \, g(n)$
- T(n) is asymptotically smaller than or equal to g(n)



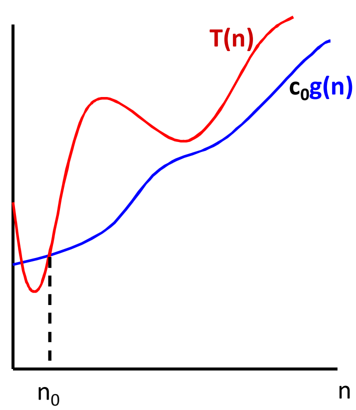
$T(n) <= c_0 \, g(n)$

Big-Oh (O): Examples

- $T(n) = 3n^2+4n = O(n^2)$
- $T(n) = 3n^2+4n = O(n^3)$
- $T(n) = 3n^2+4n \neq O(n)$
- $T(n) = (n+1)^2 = O(n^2)$

Big-Omega (Ω)

- An indicator of the best case performance (lower bound)
- Defined as a function $T(n)$ which is said to be of $\Omega(g(n))$ if there exist positive constants c_0 and n_0 such that for all $n >= n_0$, we have
  $T(n) >= c_0\, g(n)$
- $T(n)$ is asymptotically greater than or equal to $g(n)$



$T(n) >= c_0\, g(n)$

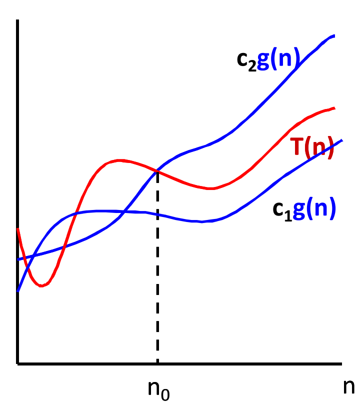
Big-Omega (Ω): Examples

- $T(n) = 3n^2+4n = \Omega(n^2)$
- $T(n) = 3n^2+4n = \Omega(n)$
- $Tn = 4n+2 = \Omega(n)$
- $Tn = 4n+2 = \Omega(1)$

Big-Theta (Θ)

- An indicator of the worse case and best case performance (asymptotically tight bound)
- Defined as a function $T(n)$ which is said to be of $\Theta(g(n))$ if there exist positive constants c_1, c_2 and n_0 such that for all $n >= n_0$, we have

  $c_1\, g(n) <= T(n) <= c_2\, g(n)$
- $T(n)$ is asymptotically equal to $g(n)$



$T(n) <= c_0\, g(n)$

Big-Theta (Θ): Examples

- $T(n) = 3n^2+4n = \Theta(n^2)$
- $T(n) = 3n^2+4n \neq \Theta(n)$
- $Tn = 4n+2 = \Theta(n)$
- $Tn = 4n+2 \neq \Theta(1)$

Order of Growth

Analysis Type	Mathematical Expression	Relative Rates of Growth
Big O	$T(N) = O(F(N))$	$T(N) \leq F(N)$
Big Ω	$T(N) = \Omega(F(N))$	$T(N) \geq F(N)$
Big θ	$T(N) = \theta(F(N))$	$T(N) = F(N)$

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

Name Convention

- O(1) Constant
- O($\log_c n$) Logarithmic (same order $\forall c$)
- O($\log^c n$) Polylogarithmic
- O(n) Linear
- O(n^c) Polynomial
- O(c^n), c>1 Exponential
- O(n!) Factorial

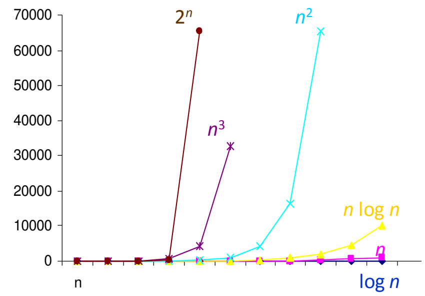
Complexity and Tractability

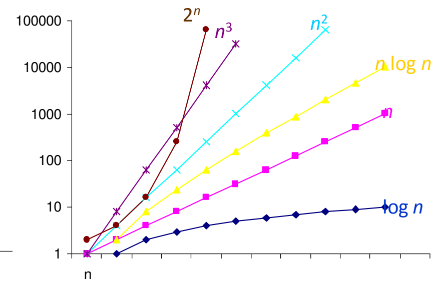
	$T(n)$						
n	n	$n \log n$	n^2	n^3	n^4	n^{10}	2^n
10	.01μs	.03μs	.1μs	1μs	10μs	1s	1μs
20	.02μs	.09μs	.4μs	8μs	160μs	2.84h	1ms
30	.03μs	.15μs	.9μs	27μs	810μs	6.83d	1s
40	.04μs	.21μs	1.6μs	64μs	2.56ms	121d	18m
50	.05μs	.28μs	2.5μs	125μs	6.25ms	3.1y	13d
100	.1μs	.66μs	10μs	1ms	100ms	3171y	4×10^{13}y
10^3	1μs	9.96μs	1ms	1s	16.67m	3.17×10^{13}y	32×10^{283}y
10^4	10μs	130μs	100ms	16.67m	115.7d	3.17×10^{23}y	
10^5	100μs	1.66ms	10s	11.57d	3171y	3.17×10^{33}y	
10^6	1ms	19.92ms	16.67m	31.71y	3.17×10^{7}y	3.17×10^{43}y	

Assume the computer does 1 billion ops per sec.

32

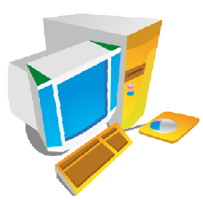
$\log n$	n	$n \log n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65,536
5	32	160	1,024	32,768	4,294,967,296

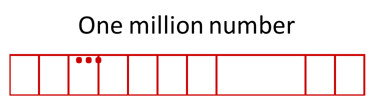




33

Time Complexity – How much it matters?





Computer - A
10^9 instructions/sec
Insertion sort
($2n^2$)

Computer - B
10^7 instructions/sec
Merge sort
50 n log n

2000 Seconds

100 Seconds

Summary

- Algorithms are omnipresent in computational domain
- Time complexity of algorithm does matter
- Time complexity of an algorithm is measured as rate of growth of the running time wrt input size
- Time complexity is normally expressed
  - Big-Oh (O) ----- upper bound (most Popular)
  - Big-theta (Θ) ----- tight (upper & lower) bound
  - Big-omega (Ω) ----- lower bound
- Can you measure time complexity of a given algorithm?

To-Do

- Do assignment #1