Data Structures

R. K. Ghosh

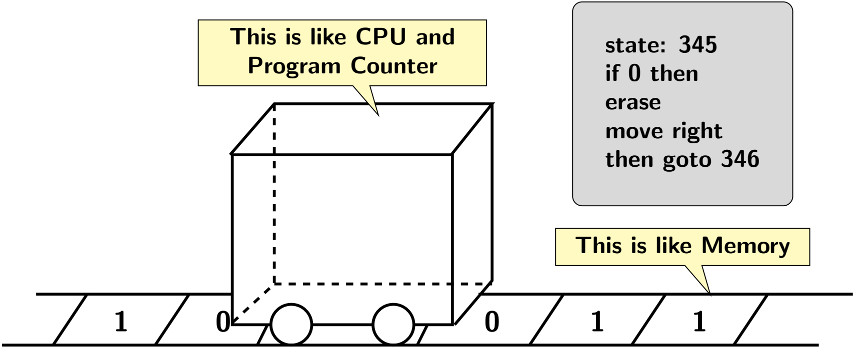
IIT Kanpur

Data structures: Computational Model

Turing Machine

- Turing proposed a generic definition of a computation.
- He viewed that any information can be written in coded form (a string of 0s and 1s).
- An infinite tape divided into cells, each holding a 0 or an 1 or some spaces.
- There is an automaton which has a knowledge of its current state.
- The automaton examines each cell on the tape at a time.
- Looks at a program book which tells it what to do in the current state.

Turing Machine



- After examining current input, TM either moves left or right.
- Changes its state as specified by the program.

Turing Machine

- Initial conditions: entire input string w is present on the tape surrounded by infinite number of blanks.
- Final state: if TM halts in final state then it accepts w
- TM halts in a non final state: it rejects w
- In general a transition is expressed as: $\delta(q, X) = (p, Y, D)$,
  - q: current state,
  - X: TM's RW-head at tape symbol X
  - Y: Output symbol, RW-head erases X and replaces it by Y.
  - p: New state
  - D: could be R or L specifying movement of RW-head

Computation versus Language

- Calculation: Takes an input value and outputs a value.
- Language: A set of string meeting certain criteria.
- So, language for a calculation basically a set of strings of the form "$<$input, output$>$", where output correspond to value calculated from the input.

Computation versus Language

L_{add} could consists of strings

$<$0+0, 0$>$	$<$0+1,1$>$	$<$0+2,2$>$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$<$5+7, 12$>$	$<$5+8,13$>$	$<$5+9,14$>$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$

Membership question: Verifying a solution $<$13+12, 25$>$ belongs to L_{add} or not?

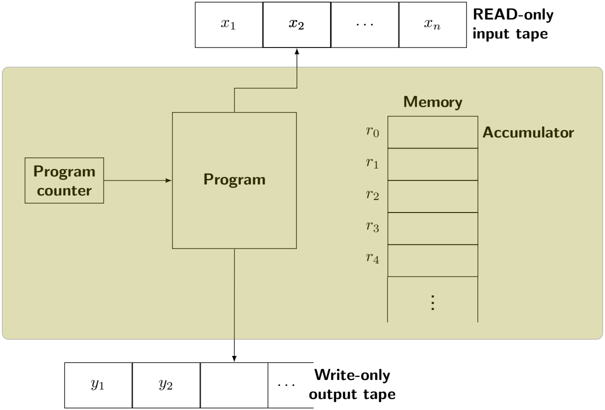
Random Access Machine

- Disconnect between a TM and real computer is sequential tape vs random access memory.
- A RAM is a simplified abstraction of real world computer
- It has an unbounded memory and capable of storing an arbitrarily large integer in each memory cell.
- RAM can access content of any random memory cell.
- However, to access a random cell, RAM needs to read the address for the cell in a different register.
- For description of algorithms it is practical to use RAM, since it is closest to a real program.

RAM Model

- Instructions are executed sequentially.
- Impractical to define instructions of each machine, and their corresponding costs.
- Therefore, a set of commonly found instructions in a computer are assumed:
  - **Arithmetic**: ADD, SUB, MULTI, DIV,
  - **Data movement**: LOAD, STORE, WRITE, READ
  - **Control**: JUMP, JGTZ, JZERO, HALT.
- Assume each instruction takes one unit of time.
- A RAM program is not stored in memory of RAM, so instructions cannot be modified.

RAM Model


x_1
x_2
$\cdots$
x_n
READ-only input tape
Memory
r_0
Accumulator
r_1
r_2
r_3
r_4
$\vdots$
Program counter
Program
y_1
y_2
$\cdots$
Write-only output tape

RAM Model

- Programs of RAM not stored in the memory, so cannot be modified.
- All computation take place in register r_0 (accumulator)
- An operand can be one of the following type:
  - Immdiate Addressing ($=i$): integer i itself.
  - Direct Addressing (i): $c(i)$ contents of register r_i.
  - Indirect Addressing ($*i$): $c(c(i))$, if $c(c(i)) < 0$, machine halts.
- Initially $c(i) = 0$ for all $i \geq 0$.
- LC (PC) is set to first instruction of program P.
- After execution of k instruction LC = $k + 1$, automatically unless k instruction is JUMP, JGTZ, or JZERO.

Meaning of an Instruction & Program

- Value $v(a)$ of an operand a is defined as follows:
  - $v(=i) = i$, $v(i) = c(i)$, $v(*i) = c(c(i))$.
- Program is essentially defines a mapping of input tape to output tape.
- Since, program may not halt for some input, the mapping is only partial.

An Example

Pseudo Code

begin
    $d = 0$;
    read x;
    while $x \neq 0$ **do begin**
        if $x \neq 1$ **then** $d = d - 1$;
        else $d = d + 1$;
        read x;
    end;
    if $d == 0$ then **write** 1
end

Accepts all strings with same number of 1s and 2s, 0 is end marker.

An Example

RAM Program

Label	Instruction	Operand	Comment
	LOAD	=0	$d = 0$
	STORE	2	
	READ	1	read x
while:	LOAD	1	while $x \neq 0$ do
	JZERO	endwhile	
	LOAD	1	if $x \neq 1$
	SUB	=1	
	JZERO	one	
	LOAD	2	then $d = d - 1$
	SUB	=1	
	STORE	2	
	JUMP	endif	
one:	LOAD	2	else $d = d + 1$
	ADD	=1	
	STORE	2	
endif:	READ	1	read x
	JUMP	while	
endwhile:	LOAD	2	if $d = 0$ then write 1
	JZERO	output	
	HALT		
output:	WRITE	=1	
	HALT		

Assignment #2

Questions (Full Marks 35)

In each case you have to provide the theoretical solution in LaTeX. All programs should be submitted as per instructions provided in the course website.

1. Give a RAM Program for computing n^k, using squaring each time. [15]
2. Write a TM program for doubling of an input consisting of k consecutive 1s. Replace the input with 2k consecutive 1s. [10]
3. Write a TM program that accepts a binary number if it is divisible by 3. [10]

Complexity

- Two important measures for an algorithm: Running time and Space requirement.
- Worst case time complexity: For a given input size, the complexity is measured as the maximum of time taken over all possible inputs that size.
- Average case time complexity: Equals to average of the time complexity over all input of a given size.
- Average case complexity is difficult to determine because, it requires certain assumptions about distribution of inputs. These assumption may at times won't be mathematically tractable.

Notion of Running Time

- Sorting 1000 elements takes more time than sorting 3 elements.
- A given algorithm take different amount of times for same input size.
  - Data **shifting** is not required in insertion sort for a **sorted** sequence.
  - But required for a **reverse sorted** sequence.

Example

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

Time Complexity

Description	Times executed
Initialization step	1
Comparison step	$n + 1$
Addition and assignment steps together	$2n$
Increment step	n
Total	$4n + 2$.

Example

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

Time Complexity

Description	Times executed
Initialization	1 + n (1 for i, n for j)
Comparison step	$(n+1)$ + $n(n+1)$
Addition and assignment steps together	$2n \times n$
Increment step for first loop	n
Increment step for second loop	n^2
Total	$4n^2 + 4n + 2$

Example

```
for (i = 0; i < n; i++)
    for (j = i + 1; j < n; j++)
        if (a[i] < a[j])
            swap(a[i], a[j]);
```

Time Complexity

- The input is n-element array, so code will result in:
  - $n-1$ comparisons for a[0]
  - $n-2$ comparisons for a[1], and so on.
  - In general, $n-i-1$ comparisons for a[i]
- Therefor, total number of comparisons = $\sum_{i=0}^{n-1}(n-i-1)$ or $\sum_{i=1}^{n-1} i = n(n-1)/2$

Relative Comparison of Execution Speeds

- Suppose an algorithm A takes time $5000n$ and another algorithm B takes time 1.1^n

Execution Speeds

Input	Program A	Program B
n	$5000n$	1.1^n
10	50000	5500
100	500,000	13,781
1000	5,000,000	2.5×10^{41}
1,000,000	5.10^9	$4.8.10^{41392}$

Largest Solvable Input Size

What is the largest problem size n that can be solved by in 1 minute by an algorithm which has running time, in microseconds
(a) $\log n$ (b) $\sqrt{n}$ (c) n (d) n^2 (e) 2^n

Solution

(a) $\log n = 6 \times 10^7$, so $n = 2^{6*10^7}$
(b) $\sqrt{n} = 6 \times 10^7$, so $n = 36 * 10^{14}$
(c) $n = 6 \times 10^7$, so nothing to solve here.
(d) $n^2 = 6 \times 10^7$, so $n = \sqrt{n^2} = 7745$
(e) $2^n = 6 \times 10^7$, so $n = \log 6 \times 10^7 = 58$

Influence of Machine Speeds

Machine Speeds

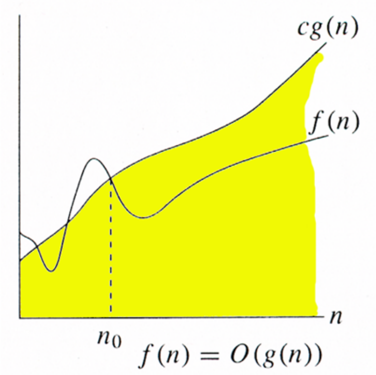
Function	Size of Large Problem Instance in 1 hour		
	With M1	**With M2**	**With M3**
n	N1	100N1	1000N1
n^2	N2	10 N2	31.6 N2
n^3	N3	4.64 N2	10 N2
2^n	N4	N4 + 6.64	N4 + 9.97
3^n	N5	N5 + 4.19	N5 + 6.29

- For 2^n case, in 1hr = $N4$ with slow computer
- For fast computer $100 \times 2^{N4} = 2^{Nx}$,
  $Nx = N4 + (\log 100 / \log 2) = N4 + 6.64$

Efficiency is Measured by Growth Rate

- An important measure of efficiency of a program is how the number of steps (time) grows as the input size grows.
- Growth function is defined by Big-O notation.

Big Oh Notation



Definition of Big Oh

- Let $g : R \to R$ be a function.
- A function $f : R \to R$ is **big-Oh** of $g(n)$, if there exist positive constants $c > 0$, and $n_0 \in N$ such that

$$f(n) \leq cg(n), \text{ for all } n \geq n_0$$

Big Oh is Upper Bound

- $f(n)$ is bounded above by $g(n)$ from some point onwards. where
  - $g(n)$ is formulated as a simpler function.
  - $g(n)$ exhibits same trend in growth as $f(n)$.
- Since we are interested for large n, it is alright if $f(n) \leq cg(n)$ for $n > n_0$.

Example

$$\begin{aligned} f(n) = n^2 + 2n + 1 &\leq n^2 + 2n^2, \text{ if } n \geq 2 \\ &= 3n^2 \end{aligned}$$

Therefore, for $c = 3$ and $n_0 = 2$, $f(n) \leq cn^2$, whenever $n \geq n_0$.

Smallest Simple Function for Big Oh

- If $f(n)$ is O(n^2), is it also O(n^3)?
  - Since O(n^3) grows faster than O(n^2), it is true.
  - However, O(n^3) over estimates a O(n^2) function.
- So, our attempt will be to find smallest simple function for which $f(n)$ is O($g(n)$).
- Some well known growth functions in order of growth:
  - 1, $\log n$, n, $n \log n$, n^2, n^3, 2^n, etc.
- Notice that only +ve integral values of n are of interest.

Computing Big Oh

- Find the dominant term of the function and find its order.
  - Any exponential function dominates any polynomial function.
  - Any polynomial function dominates any logarithmic function.
  - Any logarithmic function dominates any constant.
  - Any polynomial of degree k dominates lower degree polynomial
- Premise here is:
  - The dominant term grows more rapidly compared to others.
  - It will quickly outgrow non-dominant terms.

Other Simple Rules

- If $T_1(n) = O(f_1(n))$, and $T_2(n) = O(f_2(n))$, then
  - $T_1(n) + T_2(n) = \max\{O(f_1(n)), O(f_2(n))\}$
  - $T_1(n) * T_2(n) = O(f_1(n) * f_2(n))$
- If $T(n)$ is a polynomial of k then $T(n) = \Theta(n^k)$
- $\log^k n = O(n)$ for any constant k
- For checking whether $g(n)$ and $f(n)$ are comparable find $\lim \frac{f(n)}{g(n)} \to 1$?
- E.g.: $\lim \frac{n^2}{n^2+6} = \lim \frac{2n}{2n} = 1$
- $\lim \frac{\log n}{\log n^2} = \lim \frac{(1/n)}{2(1/n)}$ = 1/2.

Some Examples

- Examples of O(n^2) functions: n^2, n^2+n, $n^2+1000n$, $100n^2$+ 1000n, n, $n/100$, $n^{1.99999}$, $n^2/(\log\log\log n)$
- $\log n! = O(n\log n)$:

$$\begin{aligned}\log n! &= \log 1 + \log 2 + \ldots + \log n\\ &\leq \log n + \log n + \ldots + \log n = n\log n\end{aligned}$$

- $2^{n+1} = 2.2^n$ for all n.
  - So with $c = 2, n_0 = 1$, $2^{n+1} = O(2^n)$.
- But $2^{2^n} \neq O(2^n)$ can be proved by contradiction.
  - Suppose $0 \leq 2^{2^n} = 2^n.2^n \leq c.2^n$, then $2^n \leq c$.
  - But no constant is greater than 2^n.

Some Proofs for Big Oh

Theorem

Prove that $n^3 + 20n + 1$ is not O(n^2).

Proof

- By definition we should have $n^3 + 20n + 1 \leq c.n^2$.
- So $n + \frac{20}{n} + \frac{1}{n} \leq c$.
- Since left side grows with n, c cannot be a constant.

Some Proofs for Big Oh

Theorem

$f(n) = \frac{n^2+5\log n}{2n+1}$ is O(n)

Proof

- $5\log n < 5n < 5n^2$, for all $n > 1$
- $2n + 1 > 2n$, so $\frac{1}{2n+1} < \frac{1}{2n}$ for all $n > 0$
- Thus $\frac{n^2+5\log n}{2n+1} \leq \frac{n^2+5n^2}{2n} = 3n$ for all $n > 1$.
- So, with $c = 3$ and $n_0 = 1$ we have $f(n) < c.n$

Some Proofs for Big Oh

Theorem

Let $f(n) = n^k$, and $m > k$, then $f(n) = O(n^{m-\epsilon})$, where $\epsilon > 0$

Proof

- Set $\epsilon = (m-k)/2$, so $m - \epsilon = (m+k)/2 > k$.
- Hence, $n^{(m-\epsilon)}$ dominates n^k.

Theorem

Let $f(n) = n^k$, and $m < k$, then $f(n) = \Omega(n^{m+\epsilon})$, where $\epsilon > 0$

Proof

- Set $\epsilon = (k - m)/2$, so $m + \epsilon = (m + k)/2 < k$.
- Hence, $n^{(m+\epsilon)}$ is dominated by n^k.

Some Proofs for Big Oh

Theorem

Show $f(n) = n^k$ is of O($n^{\log\log n}$) for any $k > 0$

Proof

- $n^k < n^{\log\log n}$ iff $k < \log\log n$, i.e., $n > 2^{2^k}$.
- Setting $n_0 = 2^{2^k}$, we have $n^k = O(n^{\log\log n})$.

Methods of Computing Big Oh

- Single loops: **for**, **while**, **do-while**, **repeat until**
  - Number of operations is equal to number of iterations times the operations in each statement inside loop.
- Nested loops:
  - Number of statements in all loops times the product of the loop sizes.
- Consecutive statements:
  - Use addition rule: O($f(n)$) + O($g(n)$) = max($g(n)$, $f(n)$)
- If/else and if/else if statement:
  - Number of operations is equal to running time of conditional evaluation and maximum of running time of **if** and **else** clauses.

Methods for Computing Big Oh

- **Switch** statements:
  - Take the complexity of the most expensive case (with the highest number of operations).
- Function calls:
  - First, evaluate the complexity of the method being called.
- Recursive calls:
  - Write down recurrence relation of running time.
  - Solution mostly possible by observing pattern of growth and prove the same on the basis of induction from base case.

Big Oh for Recursive Algorithms

- Most of the time recursive have algorithms have a general form
$$T(n) = aT(n/b) + O(n^k)$$
- On each recursive call the size of the problem is reduced by a fraction $1/b$ of the current size.
- Also a number of calls will be necessary for solution.
- Furthermore, on each call O(n^k) work is done.
- It has solutions as follows:
  - If $a > b^k$ then complexity is O($n^{\log_b a}$)
  - If $a = b^k$ then complexity is O($n^k \log n$)
  - If $a < b^k$ then complexity is O(n^k)

Example

Analysis of for Loops

```
for (i = 0; i < n; i++)
    a[i] = 0;
    for (j = 0; j < n; j++) {
        sum = i + j;
        size++;
}
```

- First for loop: n times
- Nested for loops: n^2 times
- Total: O($n + n^2$) = O(n^2)

Example

Switch Case Statement

```
char key;
int X[5], Y[5][5], i, j;
.........
switch(key) {
  case 'a' :
       for (i = 0; i < sizeof(X)/sizeof(X[0]); i++)
            sum = sum + X[i];                => O(n)
       break;
  case 'b' :
       for (i = 0; i < sizeof(Y)/sizeof(Y[0]); i++)
            for (j = 0; j < sizeof(Y[0])/sizeof(Y[0][0]); j++)
                 sum = sum + Y[i][j];        => O(n^2)
       break;
} // End of switch block
```

- So using switch statement rule: O(n^2)

Example

For & if else

```
char key;
int A[5][5], B[5][5], C[5][5];
........
if(key == '+') {
  for(i = 0; i < n; i++)
    for(j = 0; j < n; j++)
       C[i][j] = A[i][j] + B[i][j];
} // End of if block          => O(n^2)
else if(key == 'x')
   C = matrixMult(A, B);      => O(n^3)
else
   printf("Error! Enter '+' or 'x'! :");  => O(1)
```

- Overall complexity is: O(n^3).

Exponential Algorithm are Expensive

Theorem

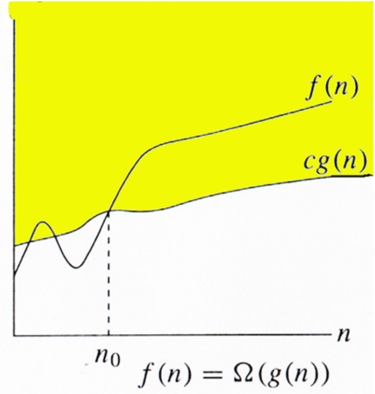
Let us first prove $n^k = O(b^n)$ whenever $0 < k \leq c$,

Proof

$$\lim \frac{n^k}{b^n} = \lim \frac{kn^{k-1}}{\ln b.b^n}$$

- The numerator's exponent decremented after each application of L Hospital's rule.
- So, b^n dominates n^k for any finite k.

Big Omega



Definition Of Big Omega

- Let $f(n)$ and $g(n)$ be functions defined over positive integers.
- $f(n)$ is $\Omega(g(n))$, if $\exists c > 0$, and $n_0 > 1$ such that

$$f(n) \geq c.g(n)$$

for all values of $n \geq n_0$.

Big Omega

Theorem

Prove $f(n) = n^3 + 20n$ is $\Omega(n^2)$

Proof

- Find $c > 0$, and $n_0 > 0$ such that $n^3 + 20n \geq c.n^2$
- Or, $c \leq n + \frac{20}{n}$.
- RHS of above expression is minimum, when $n = \sqrt{20}$
- So, with $n_0 = 5$ and $c \leq 9$ $f(n) \geq c.n^2$ for $n \geq n_0$.
- Note this is same as saying n^2 is O($n^3 + 20n$).

Theorem

Prove $f(n) = n^3 + 20n$ is $\Omega(n^3)$

Proof

- Find $c > 0$, and $n_0 > 0$ such that $n^3 + 20n \geq c.n^3$
- I.e., $c \leq 1 + \frac{20}{n^2}$,
- Let $c = 1$ and $n_0 = 1$, then $f(n) \geq c.n^3$ for $n \geq n_0$.

Big Omega

Theorem

Prove that $f(n)$ is $\Omega(g(n))$ iff $g(n) = O(f(n))$.

Proof

- If $f(n) = \Omega(g(n))$ then $\exists c > 0$ and $n_0 \geq 1$ such that $f(n) \geq c.g(n)$.
- It implies $g(n) \leq \frac{1}{c} f(n)$.
- Let $\frac{1}{c} = c_1$. Since $c > 0$, $c_1 > 0$.
- So, we have $g(n) \leq c_1 f(n)$ for a $c_1 > 0$, and $n > n_0 \geq 1$,
- Converse part can be proved likewise.

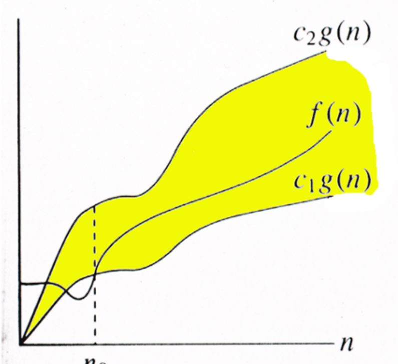
Big Omega

Example

Prove that $n^2 - 2n + 1$ is $\Omega(n^2)$

Proof

- Eliminate lowest order term $1 > 0$, $f(n) > n^2 - 2n$
- If $n > 10$, then $-10 > -n$, implies $-2 > 0.2n$
- Now $-2 > -0.2n$ implies $-2n > -0.2n^2$
- So, $n^2 - 2n > n^2 - 0.2n^2 = 0.8n^2$
- Furthermore, $n > 10$ implies $.8n^2 > n^2/2$
- Therefore, $n^2 - 2n + 1 > n^2/2$ for $n > n_0 = 10$.



$f(n) = \Theta(g(n))$

Definition Of Big Theta

- Let $f(n)$ and $g(n)$ be functions defined over positive integers.
- $f(n)$ is $\Theta(g(n))$, if $\exists c_1 > 0$, $c_2 > 0$ and $n_0 > 1$ such that

$$c_1.g(n) \leq f(n) \geq c_2.g(n)$$

for all values of $n \geq n_0$.

Big Theta

Example

Show that $f(n) = 3n^2 + 8n \log n$ is $\Theta(n^2)$.

Proof

- For $n > 1$ $0 \leq 8n \log n \leq 8n^2$. Therefore, $3n^2 + 8n \log n \leq 11n^2$
- Also n^2 is O($3n^2 + 8n \log n$).
- Hence, $3n^2 + 8n \log n = \Theta(n^2)$.

Use of Limits

- A quick way to determine if $f(n)$ is O($g(n)$) is to find if

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

  exists and finite.
- Similarly, if the above limit is not equal to zero. then $f(n)$ is $\Theta(g(n))$.
- If above limit is some value c where $0 < c \leq \infty$ then $f(n)$ is $\Omega(g(n))$.

Use of Limits

- There are two other asymptotic bounds called little ω and little o.
- These bounds are loose bounds.
- If $\lim_{n\to\infty} \frac{|f(n)|}{|g(n)|} = 0$ then $f(n)$ is $o(g(n))$
- If $\lim_{n\to\infty} \frac{|f(n)|}{|g(n)|} = \infty$ then $f(n)$ is $\omega(g(n))$

Use of Limits

Example

Prove $f(n) = 7n + 8$, is $o(n^2)$.

Proof

$$\lim_{n\to\infty} \frac{7n+8}{n^2} = \lim_{n\to\infty} \frac{7}{n}, \text{ by l'Hospital}$$
$$= 0$$

Upper & Lower Bounds

- Upper and lower bounds give only incomplete information.
- Bounds are interesting when we have incomplete knowledge of execution time.
- Upper (lower) bound is not the same as worst (best) case
- Best and worst cases are not tied to input sizes.
  - They express the distribution of input elements, so that for a given size maximum (minimum) execution time is spent.

Upper & Lower Bounds

Upper bound

Lowered by better algorithm

UB implies we know the best algorithmic solution.

LB represent the best theoretical solution

Lower bound

Raised by better proof

Upper & Lower Bounds

Upper bound **Closed problems have identical bounds**

Lower bound

Upper bound

LB & UB differ: Unknown space

Lower bound

- For closed problems, better algorithms are possible: it does not change big-Oh but reduces hidden constant.

Tractable and Intractable Problems

	Problems	Algorithms
Polynomial	**Tractable**	**Reasonable**
Exponential	**Intractable**	**Unreasonable**

Definition (Tractable)

If upper and lower bounds have only polynomial factors.

Definition (Intractable)

If both upper and lower bounds have an exponential factor.

Assignment #3

T

his assignment is a written assignment to be submitted on or before the due date as indicated in the assignment sheet.

Summary

Concepts

- Introduced theoretical models of computation: TM and RAM
- Notion of running time
- Big Oh, Big Omega, Big Theta, little oh and little omega.
- Some worked out examples.
- Upper bound and lower bounds.