# Data Structures

R. K. Ghosh

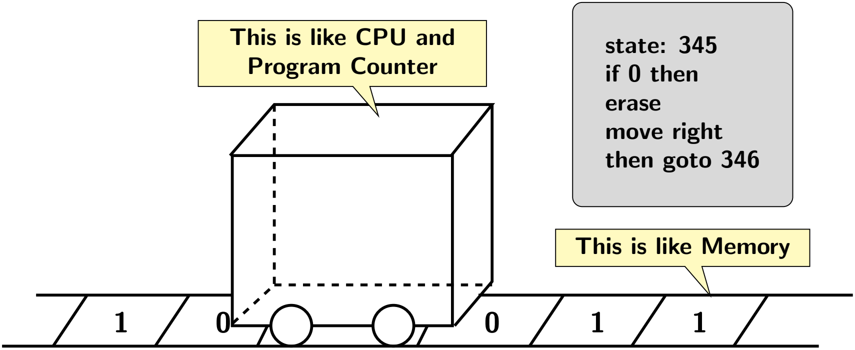
IIT Bhilai

Data structures: Computational Model

# Turing Machine

- A. M. Turing gave an abstract definition of a computation using abstract machine called TM.
- A TM manipulates a string of 0s, 1s and spaces on a strip of tape according a table of rule (program book).
- There is control (automaton):
  - It has a knowledge of its current state.
  - It examines each cell on the tape at a time.
  - It consults a program book which tells it what to do in the current state.

# Turing Machine




- After examining current input, RW-head of 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 $w$ is rejected.
- 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?

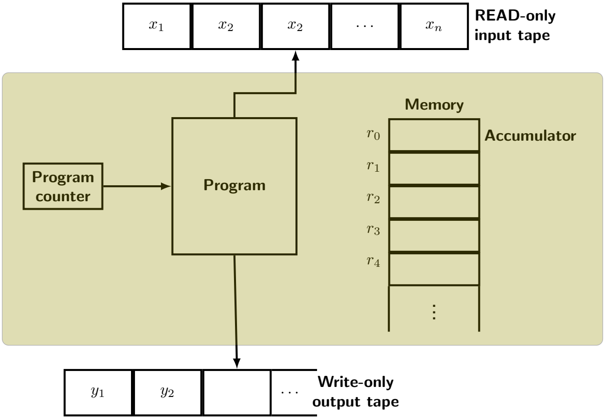
# Random Access Machine

- 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.
  - A 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



$x_1$
$x_2$
$x_2$
$\cdots$
$x_n$
READ-only input tape
Memory
$r_0$
$r_1$
$r_2$
$r_3$
$r_4$
Accumulator
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**

For algorithm that accepts strings (with end marker 0) having equal number of 1s and 2s.

```
begin
    d = 0;
    read x;
    while x ≠ 0 do begin  // 0 is used as endmarker
        if x ≠ 1 then d = d − 1;
        else d = d + 1;
        read x;
    end;
    if d == 0 then write 1
end
```

# An Example

## RAM Program

| | | | |
|---|---|---|---|
| | LOAD | =0 | |
| | STORE | 2 | $d = 0$ |
| | READ | 1 | read $x$ |
| while: | LOAD | 1 | while $x \neq 0$ do |
| | JZERO | endwhile | |
| | LOAD | 1 | |
| | SUB | =1 | if $x \neq 1$ |
| | JZERO | one | |
| | LOAD | 2 | |
| | SUB | =1 | then $d = d - 1$ |
| | STORE | 2 | |
| | JUMP | endif | |
| one: | LOAD | 2 | |
| | ADD | =1 | else $d = d + 1$ |
| | STORE | 2 | |
| endif: | READ | 1 | read $x$ |
| | JUMP | while | |
| endwhile: | LOAD | 2 | |
| | JZERO | output | if $d = 0$ |
| | HALT | | then write 1 |
| 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 of 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 of 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.
  - It requires assumptions about distribution of inputs.
  - These assumption may at times won't be mathematically tractable.

# Notion of Running Time

- Sorting of 1000 elements takes more time than sorting of 3 elements.
- Even the same algorithm may take different amounts of time for the different inputs of same 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 | $2n$ |
| 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 | $2n \times n$ |
| 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 $\sum_{i=1}^{n-1} i = n(n-1)/2$

# Comparison of Relative Execution Speeds

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

**Execution Speeds**

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

# Comparing Algorithms

## The largest Input Size for a Problem

Find the largest problem size $n$ that can be solved in 1 minute by each of the four algorithms with different running times (in microsconds) : (a) $\log n$, (b) $\sqrt{n}$, (c) $n$, (d) $n^2$, and (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

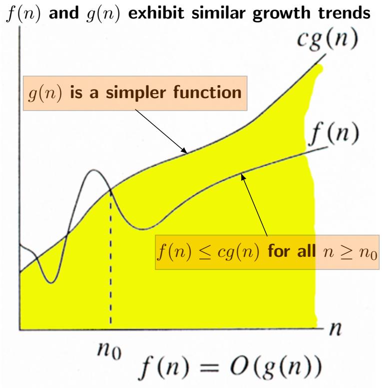
## Execution Speeds

| Complexity | Size of the Largest 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 $100 \times 2^{N4} = 2^{Nx}$,
  $Nx = N4 + (\log 100 / \log 2) = N4 + 6.64$

# Comparing Running Times

- Algorithms with exponential running times are inefficient and can solve only small problems of small sizes.
- Constant and logarithmic running times are most efficient.
- Sublinear and linear runing time are also very good.
- In general, algorithm with running times in polynomial of input size are considered efficient.
- Increasing machine speed does not help in scaling up.
- Therefore, the important measure of efficiency of a program is:
  **How the number of steps (time) grows with the input size?**




## Definition of Big Oh

- Growth function is defined by Big-O notation.
- $f(n)$ is **big-Oh** of $g(n)$, if
  - $g : R \to R$
  - $f : R \to R$,

  and 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 an O($n^2$) function.
- So, our attempt will be to find the 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.

# Guidelines for Computing Big Oh

- Find the dominant term of the function and find its order.
  - A logarithmic function dominates all constants.
  - A polynomial function dominates all logarithmic functions.
  - A polynomial of degree $k$ dominates all lower degree polynomials.
  - An exponential function dominates all polynomial functions.
- Basis here is that:
  - The dominant term grows more rapidly compared to others.
  - It will quickly outgrow non-dominant terms.

# Other Simple Rules

- If $\mathsf{T}_1(n) = \mathsf{O}(f_1(n))$, and $\mathsf{T}_2(n) = \mathsf{O}(f_2(n))$, then
  - $\mathsf{T}_1(n) + \mathsf{T}_2(n) = \max\{\, \mathsf{O}(f_1(n)), \mathsf{O}(f_2(n))\}$
  - $\mathsf{T}_1(n) * \mathsf{T}_2(n) = \mathsf{O}(f_1(n) * f_2(n))$
- If $\mathsf{T}(n)$ is a polynomial of $k$ then $\mathsf{T}(n) = \Theta(n^k)$[1]
- $\log^k n = O(n)$ for any constant $k$
- For checking whether $g(n)$ and $f(n)$ are comparable find $\lim \frac{f(n)}{g(n)} \leq k$, where $k > 0$ is a constant?
- E.g.: $\lim \frac{n^2}{n^2+6} = \lim \frac{2n}{2n} = 1$
- $\lim \frac{\log n}{\log n^2} = \lim \frac{(1/n)}{2(1/n)} = 1/2.$

[1] Not defined yet

# Some Examples

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

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

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

# Some Proofs for Big Oh

**Exercise 1**

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

**Solution**

- Assume that $n^3 + 20n + 1$ is O($n^2$).
- By definition of big-Oh it implies $n^3 + 20n + 1 \leq c.n^2$.
- Divide both side of the inequality by $n^2$.
- 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

**Exercise 2**

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

**Solution**

- $5\log n < 5n < 5n^2$, 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

**Exercise 3**

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

**Solution**

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

# Some Proofs for Big Oh

**Exercise 4**

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

[a] $\Omega$ not defined yet

**Solution**

- 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

**Exercise 5**

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

**Solution**

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

# Computing Big Oh of Programs

- 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($g(n), f(n)$)
- Conditional statement:
  - Number of operations is equal to running time of conditional evaluation and the maximum of running time of **if** and **else** clauses.

# Computing Big Oh of Programs

- **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 the base case.
  - For divide and conquer algorithms Master Theorem can be used.

# 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($n^2$)

# 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($n^2$)

# 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($n^3$).

**Exercise 6**

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

**Solution**

$$\lim \frac{n^k}{b^n} = \lim \frac{kn^{k-1}}{\ln b . b^n} \text{ (set } b^n = e^{n \ln b})$$

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

# Big Oh for Recursive Algorithms

```
procedure T(n: size of the problem) {
    if (n < 1)
        exit()
    Do work of amount n^k

    T(n/b) // Repeat for a times
    T(n/b)
    ...
    T(n/b)
}
```

- The original problem is recursively divided into $a$ subproblems of $n/b$.
- In each recursive call O($n^k$) work is done.

# Big Oh for Recursive Algorithms

## Master Theorem

- The expression for time complexity is

  $T(n) = aT(n/b) + O(n^k)$, where $a > 0, b > 1$ and $k \geq 0$

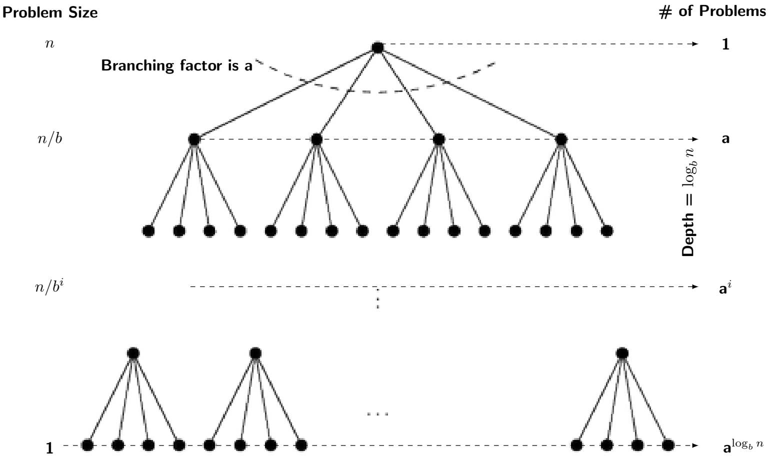
- The time complexity for recursive algorithms is given by:

$$T(n) = \begin{cases} O(n^k) \text{ if } a < b^k \\ O(n^k \log n) \text{ if } a = b^k \\ O(n^{\log_b a}) \text{ if } a > b^k \end{cases}$$

# Recursion Tree

- Before solving, let us take a look at recursion tree.
- $n$ is assumed to be a power of $b$, if not pad $n$ to be larger.
- It requires more than $b$ to be added to $n$.
- At level 0, when we start the problem size is $n$.
- At level 1, we have $a$ problems of size $n/b$ each.
- In general, at level $i$, we have $a^i$ problems of size $n/b^i$ each.

# Recursion Tree



# Solution of Master's Theorem

- First let us unfold the recurrence relation:

$$\begin{aligned}
T(n) &= aT\left(\frac{n}{b}\right) + n^k \\
&= a\left(aT\left(\frac{n}{b^2}\right) + \frac{n^k}{b^k}\right) + n^k \\
&\vdots \\
&= n^k + \frac{a}{b^k}n^k + \frac{a^2}{(b^k)^2}n^k + \cdots + \frac{a^L}{(b^k)^L}n^k \\
&= n^k\left(1 + \frac{a}{b^k} + \left(\frac{a}{b^k}\right)^2 + \left(\frac{a}{b^k}\right)^3 + \cdots + \left(\frac{a}{b^k}\right)^L\right)
\end{aligned}$$

- Here, $L = \log_b n$.

# Solution of Master's Theorem: Case I

- The expression within brackets is a GP, of the form

$$1 + r + r^2 + r^3 + \cdots + r^L, \text{ where } r = \frac{a}{b^k} \text{ and } L = \log_b n$$

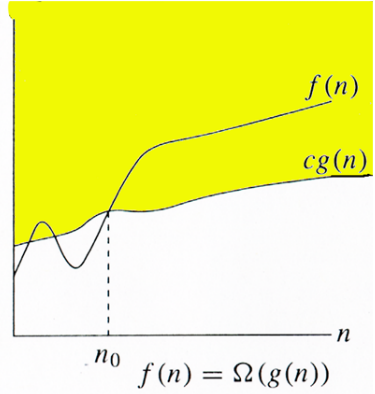
- In this case $a < b^k$, $r = \frac{a}{b^k} < 1$
- Therefore, the first term dominates the running time.
- In other words, the level 0 of the recursion dominates the runtime.
- Hence, the solution in this case will be O($n^k$).

# Solution of Master's Theorem: Case II

- In this case $a = b^k$, or $r = 1$ in the expression for the running time.
- In this case, equal work (=$n^k$) is done at every level of the recursion.
- Since depth of recursion is $1 + \log n$, the running time in this case is O($n^k \log n$).

# Solution of Master's Theorem: Case III

- Here, $a > b^k$, which implies $\frac{a}{b^k} > 1$.
- This means the last term in the sum dominates the runtime.
- So, the runtime should be O($n^k \left(\frac{a}{b^k}\right)^L$) = O($a^L$), as $(b^k)^L = (b^L)^k = n^k$
- Now replace $L$ by $\log_b n$ to get O($a^{\log_b n}$)
- $a^L = (b^{\log_b a})^{\log_b n} = (b^{\log_b n})^{\log_b a}$ = $n^{\log_b a}$.




**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($n^3 + 20n$).

# Big Omega

### 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

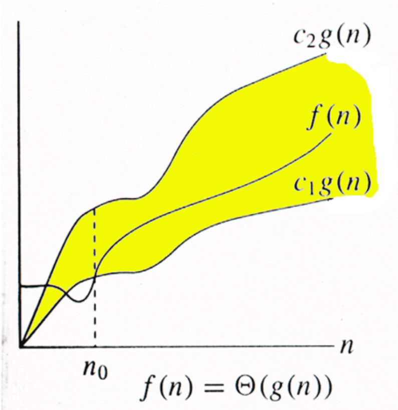
**Example**

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 $n > n_0 = 10$.

# Big Theta




**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 $n \geq n_0$.

## Example

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

## Proof

- For $n > 1$, since $0 \leq 8n \log n \leq 8n^2$, we have $3n^2 + 8n \log n \leq 11n^2$
- 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 constant $c$, where $0 < c < \infty$ then $f(n)$ is $\Omega(g(n))$.

# Little oh and Little omega

- There are two other asymptotic bounds called little $\omega$ 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 $\omega(g(n))$

**Exercise**

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 important when we have incomplete knowledge of execution time.
- Upper (or lower) bound is not the same as the worst (the best) case input size.
- The best and the worst cases are not tied to input sizes.
  - They express the distribution of the input elements, so that for a given size what would be the maximum (or the minimum) execution time.

# 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

### Assignment on Running Time

It will be a theoretical assignment which will be posted soon. Due date for the assignment will be as indicated in the sheet.

# Summary

## Computational 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.