L1: Motivation and Fundamentals

Welcome to the course of Advance Data Structures PCS101 3L 1T 2P Credit: 4.5
My name is Dr Deepak Garg and This Course has been assigned to me.

Motivation for the Course

- ✓ It is the study of data structures and algorithms
- ✓ Keep in mind that objective is to learn how to evolve a solution of a problem by choosing a good data structure with a good algorithm design
- ✓ From all perspectives it is the most important course in CS&E to be a SE and will be required throughout your studies and career.
- ✓ A Must for getting any placement

Book

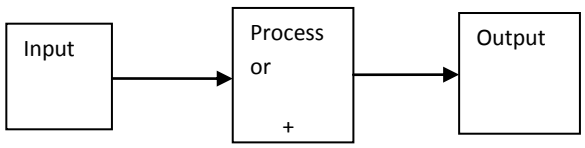
Introduction to Algorithms from PHI Third Edition by Cormen, Leiserson, Rivest and Stein
Algorithm Design from Wiley by Goodrich and Tamassia
Algorithm Design from Addison Wesley by Jon Klienberg and Eva Tardos

However this does not mean that you should not consult other books or evaluation has to be from this book
Class: If you are not on time or absent, then it is up to you to be ready for understanding the remaining part

Evaluation:

- 4 evaluations on 1st tutorial of every month for 4 marks each: 16 Marks
- 4 Assignments, one every month for 4 marks each: 16 Marks
- 8 marks for Viva at the end of the course
- 20 marks Mid Semester Test
- 40 marks End Semester Test
- Total 100
- Problems: Any administrative problems through the CR only
- Questions are encouraged any time during the class/Tutorial and lab hours

Computer



- ∑ Is today's computer sufficiently described by this model (yes/No?)
- ∑ Now Computer not only means Compute or processing of numbers but also communication/ storage/ analysis/ designing.
- ∑ Now it is not the privilege of a few big organization but a must have device for the masses.

Memory Hierarchy

- ➢ Registers
- ➢ L1 Cache (Internal cache)-SRAM
- ➢ L2/L3 Cache (External Cache)-SRAM
- ➢ DRAM-Main Memory
- ➢ Hard Disk – Secondary Storage

- External Storage – Floppy, CD, DVD, Flash Drive
- Size is increasing(individually and hierarchically)
- Cost/byte is decreasing
- Speed is decreasing

Evolution in various things

- Vacuum Tubes-> Transistors-> ICs->LSICs->VLSIs->UlSICs->
- Thousand 10^3->Million 10^6->billion 10^9->trillion 10^{12} instructions per second
- Byte->Kilo->Mega->Giga->Tera
- 8bit-> 16 bit->32bit->64->128 bit processors
- Machine Code-> Assembly Language-> HLLs
- Procedural-> structured -> object oriented languages
- Numbers+ Chars+ Text+ Graphics+ Audio+ Video+ Multimedia

Definition

Data structure is a way to store and organize data in order to facilitate access and modifications.
Example of storing an array in randomly stored Memory matrix, which would have been possible if objects were stored in an organized way
If the strengths and limitations of a structure will be known, then we can utilize the best one available for a given structure.
Static: Values are fixed
Dynamic Data Structure: updations can take place, insertions, deletions can happen
Data Structures fro geometric Problems like KD-Trees
Data Structures for external Memory, Cache-oblivious data structures
Self-adjusting data structures, Persistent Data Structures, Retroactive Data Structures

❶

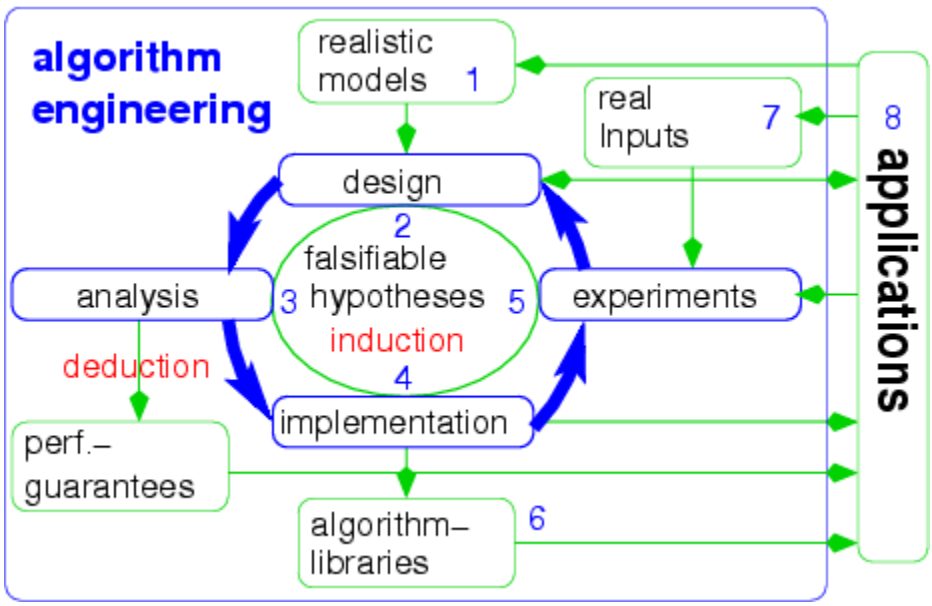
L2: Algorithms and their characteristics

- Algorithm is a sequence of Finite and unambiguous computational steps that transform the input into the correct output effectively.
  Finite theoretically means having finite number of steps, but for an algorithm to be effective this finite amount has to be within reasonable limits of computing resources available.
  Discuss correct with the point that in some cases incorrect solutions are acceptable if they are within the range of allowed error, which is also good in many cases, where the correct algorithms are not available with the limitation of current resources.
  Unambiguous is to be explained with deterministic theory.
- Algorithm is also called a pseudo code
- Assume we have infinite speed processors with infinite memory, do we still need to study data structure and algorithms. Yes! It is an art… It is fun… Why people run in a race. Even if there are cars and airplanes
- Examples of Online Banking, Shopping, Reservations and Biotechnology

Desirable characteristics of an algorithm

- Modularity
- Scalability
- Graceful Degradation for size and number of inputs
- Maintainability
- Simple
- User friendly(To be correlated with performance
- Extensible
- Programmer Time
- Concurrency
- Distributed
- Upload performance
- Security
- Power Efficiency
- Hardware/OS compliant

Performance draws line between feasible and impossible. Algorithms give language for talking about program behavior. Performance can be used to "pay" for other things, such as security, features and user-friendliness



Binary Number	Octal Number*	Decimal Number	Hexadecimal
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	10	8	8
1001	11	9	9
1010	12	10	A
1011	13	11	B
1100	14	12	C
1101	15	13	D
1110	16	14	E
1111	17	15	F

❷

L3: Some Mathematics

Given a sequence $a1$, $a2$… of numbers, the infinite sum $a1 + a2 + \cdots$ can be written

$$\sum_{k=1}^{\infty} a_k \quad or \quad \lim_{n\to\infty} \sum_{k=1}^{\infty} a_k$$

For any real number c and any finite sequences $a1$, $a2$... an and $b1$, $b2$... bn,

$$\sum_{k=1}^{n}(ca_k + b_k) = c\sum_{k=1}^{n} a_k + \sum_{k=1}^{n} b_k \,.$$

Arithmetic Series (difference of two successive numbers is a constant)

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

Sum of squares

$$\sum_{k=0}^{n} k^2 = \frac{n(n+1)(2n+1)}{6},$$

Sum of cubes

$$\sum_{k=0}^{n} k^3 = \frac{n^2(n+1)^2}{4}.$$

Geometric Series (every two consecutive terms have a common ratio)

$$\sum_{k=0}^{n} x^k = 1 + x + x^2 + \cdots + x^n$$

$$\sum_{k=0}^{n} x^k = \frac{x^{n+1} - 1}{x - 1}.$$

When the summation is infinite and $|x| < 1$, we have the infinite decreasing geometric series

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}.$$

Harmonic Series (by taking the reciprocals of an Arithmetic Progression)

$$\begin{aligned} H_n &= 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n} \\ &= \sum_{k=1}^{n} \frac{1}{k} \\ &= \ln n + O(1)\,. \end{aligned}$$

Telescoping series

$$\sum_{k=1}^{n}(a_k - a_{k-1}) = a_n - a_0\,,$$

The finite product $a1$ $a2 \cdot \cdot$ an can be written

$$\prod_{k=1}^{n} a_k\,.$$

Permutations

A ***permutation*** of a finite set S is an ordered sequence of all the elements of S, with each element appearing exactly once. For example, if $S = \{a, b, c\}$, there are 6 permutations of S:

abc, *acb*, *bac*, *bca*, *cab*, *cba*.

There are $n!$ permutations of a set of n elements, since the first element of the sequence can be chosen in n ways, the second in n - 1 ways, the third in n - 2 ways, and so on The number of k-permutations of an n-set is

$$n(n-1)(n-2)\cdots(n-k+1) = \frac{n!}{(n-k)!}\,,$$

A ***k-combination*** of an n-set S is simply a k-subset of S. For example, there are six 2-combinations of the 4-set $\{a, b, c, d\}$: *ab*, *ac*, *ad*, *bc*, *bd*, *cd*

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}\,.$$

Binomial Expansion and Binomial Coefiicients

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k}.$$

$$2^n = \sum_{k=0}^{n} \binom{n}{k}.$$

Induction

Let P(n) is a statement about integer n.

1. Prove P(1) is true
2. If all p(1),P(2),…….,P(n) are true then P(n+1) is also true

We know

$1=1^2$

$1+3=2^2$

$1+3+5=3^2$

$1+3+5+7=4^2$

$1+3+5+\ldots+(2n-1)=n^2$

1. P(1) is true because $1=1^2$
2. Assume it is true for P(n)

$1+3+5+\ldots+(2n-1)=n^2$ and we have to prove for P(n+1)

Adding (2n+1) on both sides

$1+3+5+\ldots+(2n-1)+(2n+1)=n^2+2n+1=(n+1)^2$ Hence proved by induction

The **logarithm** of a number to a given base is the power or exponent to which the base must be raised in order to produce that number. For example, the logarithm of 1000 to base 10 is 3, because 3 is the power to which ten must be raised to produce 1000

$$\text{if } x = b^y, \text{ then } y = \log_b(x)$$

$$\log(xy) = \log x + \log y$$

$$\log\left(\frac{x}{y}\right) = \log x - \log y$$

$$\log_b x = \frac{\log_k x}{\log_k b}.$$

Log(x) for base 2 is denoted as lg x and log(x) for base e is written as ln x

$$\frac{\log_2 27}{\log_2 9} = \frac{\log_2 c \cdot \log_c 27}{\log_2 c \cdot \log_c 9} = \frac{\log_c 27}{\log_c 9}$$

$$\frac{\log_3 27}{\log_3 9} = \frac{\log_3 3^3}{\log_3 3^2} = \frac{3}{2}$$

Exponential series $e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots.\infty$

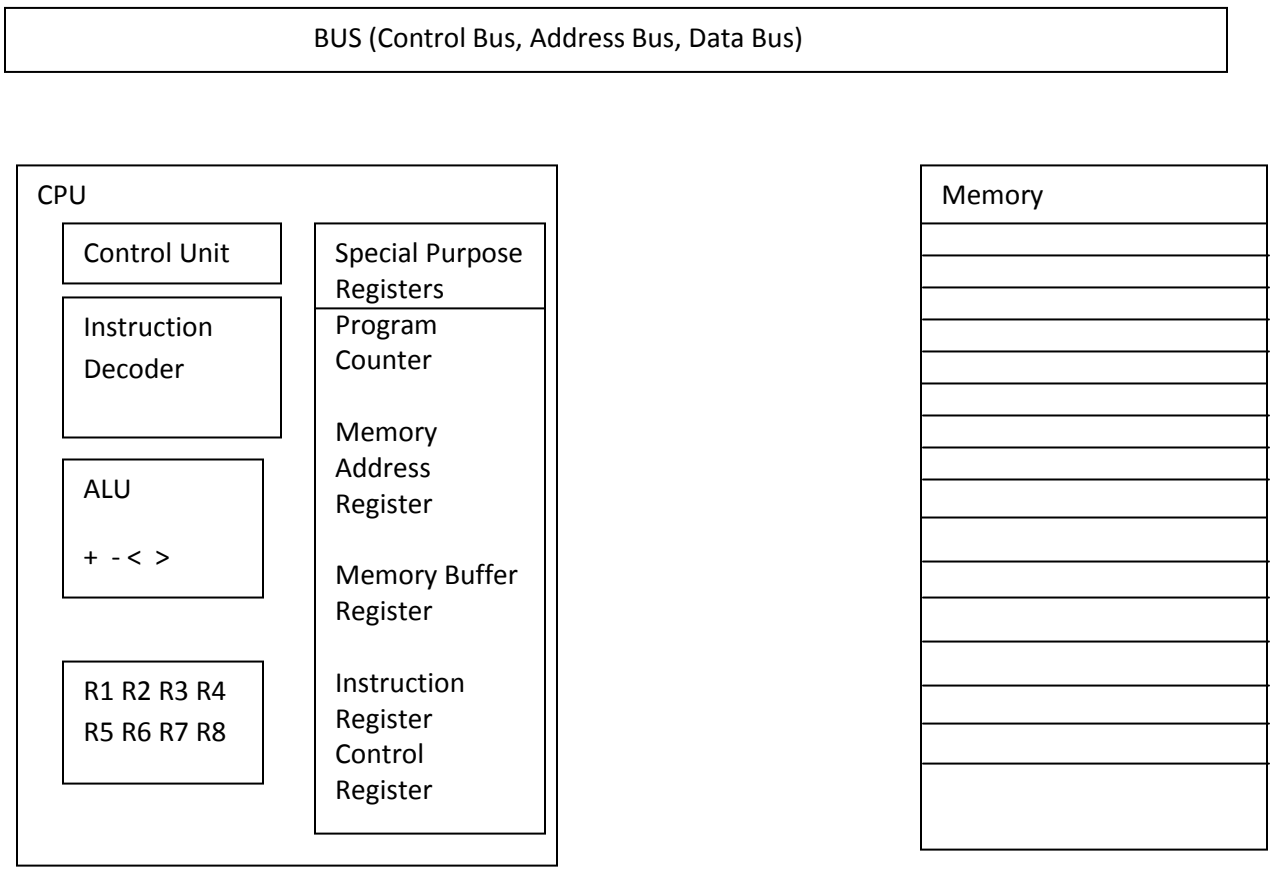
How many times we should divide n to reach 1 : lg n

❸

L4: Fetch-Decode-Execute Cycle

1. The PC value is copied into MAR
2. The CU then refers to the Memory pointed to by the MAR
3. The Instruction is fetched into the MBR
4. The entries in MBR are transferred to the IR
   These are also called micro instructions
   Decode Phase
   The Instruction decoder unit in the Control unit is responsible for the execution of this phase.
5. The opcode part of the instruction is transferred to the Instruction Decoder
6. The Instruction Decoder then decodes the opcode part (ADD), interprets and understands what to do (In this case the adder in ALU is invoked)
7. The PC is incremented to point to the next consecutive instruction

- If an instruction consists of multiple units of readable memory then multiple fetch may be required one after the other. Different instructions have different instruction length. One instruction cycle may consist of few machine cycles which in turn may consist of few clock cycles. Study of processor is also study of state machine.
- Starting point of the design of CPU is instruction set.
- Instructions commonly found in real computers:
  - Arithmetic: add, subtract, multiply, divide, remainder, floor, ceiling). Also,
  - Shift left/shift right (good for multiplying/dividing by 2^k).
  - Data movement: load, store, copy.
  - Control: conditional/unconditional branch, subroutine call and return.
- Each of these instructions takes a constant amount of time



L5: Code Tuning

- ✓ Used to write better code
- ✓ Needs better understanding of the programming language and its compiler
- ✓ Is equivalent of code optimization at Higher Language Level

Loop Jamming, Loop Fusion and Fission

Combining loops that operate over same range of values

```
int i, a[100], b[100];
for (i = 0; i < 100; i++) {
        a[i] = 1;
        b[i] = 2;
 }
```

After loop fission

```
int i, a[100], b[100];
for (i = 0; i < 100; i++)
{
        a[i] = 1;
}
for (i = 0; i < 100; i++)
{
        b[i] = 2;
}
```

Unswitching of loops

```
int i, w, x[1000], y[1000];
for (i = 0; i < 1000; i++)
{
        x[i] = x[i] + y[i];
if (w)=1
        y[i] = 0;
}
```

```
int i, w, x[1000], y[1000];
if (w)
{
for (i = 0; i < 1000; i++)
{
        x[i] = x[i] + y[i];
        y[i] = 0;
}
}
else
{
for (i = 0; i < 1000; i++)
{
        x[i] = x[i] + y[i];
}
}
```

Loop Unrolling

```
for (int x = 0; x < 100; x++)
{
delete(x);
}
for (int x = 0; x < 100; x += 2)
{
        delete(x);
        delete(x+1);
}
```

Loop-invariant code

```
for (i = 0; i < n; ++i) {
        x = y + z;
        a[i] = 6 * i + x * x;
}
```

```
x = y + z;
t1 = x * x;
for (i = 0; i < n; ++i) {
        a[i] = 6 * i + t1;
}
```

Minimize work inside loops

```
For (i = 1 to n/2)
{…
}
```

Need to compute n/2 times in all iterations

```
m = n/2;
For (i = 1 to m)
{…
}
```

Use of sentinel values

```
While(i<n) and (x<>a[i])
{
i=i+1;
}
```

```
A[n+1]=x;
While(x<>a[i])
{
i=i+1;
}
if i==(n+1)
print number not present
else
print position of number i
```

Order condition testing in Switch Case & If-Else by frequency

Read(empNo)
Case Grade(student)
1: {……}
2: {……}
3: {……}
4: {……}
Endcase

Read(empNo)
Case Grade(student)
2: {……}
3: {……}
1: {……}
4: {……}
endcase
❺

L6: Code Tuning II

Common Sub expression Elimination

```
a = b * c + g;
d = b * c * d;

tmp = b * c;
a = tmp + g;
d = tmp * d;
```

Minimizing array references

If the same array element is repeatedly referred inside a loop, then move it outside the loop

```
For( a=0;a<5;a++)
{
for (b=0;b<10;b++)
{
        Total[b]=total[b]*sum[a]
}
}
for( a=0;a<5;a++)
{
        Sum_now=sum[a]
for (b=0;b<10;b++)
{
        Total[b]=total[b]*sum_now;
}
}
```

Use Constants of Correct Type

```
Float x;
X=5;      convert 5 to 5.0 and then stores into x
int i;
i=3.14; convert 3.14 to 3 and store into i
```

Precompute Results

```
For(i=0;i<100;i++)
{
Y=log(x)/log(2)
B=log(a)/log(2)
}

Twolog=log(2)
For(i=0;i<100;i++)
{
        Y=log(x)/ Twolog
        B=log(a)/ Twolog
}
```

Dead Code elimination

```
int xyz()
{
int a = 24;
int b = 25; /* Assignment to dead variable */
int c;
c = a- 2;
return c;
b = 24; /* Unreachable code */
return 0;
 }
```

Exploit Algebraic Identities

Algebraic identities can be used to replace costlier operations by cheaper ones Whenever we need to find whether sqrt(x) <sqrt(y), we can use the algebraic identity which says x < y only when x < y. So it is enough to check if x < y in this case.

not (A or B) is cheaper than not A and not B

Lazy computations

```
n=x*x+2*y+z
if q>10 then
{
……
}
else q > n then
{
…..
}
else
{
…
}

Should be
if q>10 then
{
……
}
else
{
n=x*x+2*y+z
if q > n then
{
…..
}
else
{
…
}}
```

Short circuiting and reordering

```
if (a>b) && (c>d) && (e>f)
{
…..
}
```

Using Locality of reference by avoiding cache misses

```
for i in 0..n
        for j in 0..m
                for k in 0..p
                        C[i][j] = C[i][j] + A[i][k] * B[k][j];

for i in 0..n
        for k in 0..m
                for j in 0..p
                        C[i][j] = C[i][j] + A[i][k] * B[k][j];
```

It would be advantageous to refer to several memory addresses that share the same row (spatial locality). By keeping the row number fixed, the second element changes more rapidly. In C and C++, this means the memory addresses are used more consecutively. One can see that since j affects the column reference of both matrices C and B, it should be iterated in the innermost loop (this will fix the row iterators, i and k, while j moves across each column in the row). This will not change the mathematical result, but it improves efficiency. By switching the looping order for j and k, the speedup in large matrix multiplications becomes dramatic.

❻

L7: Complexity Analysis of an Algorithm

Complexity of an Algorithm: Resources required for running that algorithm
To estimate how long a program will run.
To estimate the largest input that can reasonably be given to the program.
To compare the efficiency of different algorithms.
To help focus on the parts of code that are executed the largest number of times.
To choose an algorithm for an application
Memory now mainly concerned with bandwidth, small gadgets, and smart cards. Vista requires 1 GB of RAM.
Running in RAM or Cache is still a big issue
Small programs more efficient (Concept of time space trade off does not hold good always)

It follows Von-Neumann Architecture

Empirical Study

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Use a function, like the built-in clock(), system.currentTimeMiilis() function, to get an accurate measure of the actual running time
- Plot the results
- Time in NSec/Size

Problems in the Empirical study

- ∑ It is necessary to implement the algorithm, which may be difficult
- ∑ Results may not be indicative of the running time on other inputs not included in the experiment.
- ∑ In order to compare two algorithms, the same hardware and software environments must be used
- ∑ Even in same hardware and software environments the results may vary depending upon the processor load, sharing of resources, No. of background processes, Actual status of Primary and Secondary Memory at the time of running the program, Compiler, Network Architecture, programming language

Apriori Vs Posterior Analysis

Apriori – Designing then making
Posterior - Making then waking up after the problem crops up
Posterior Analysis refers to the technique of coding a given solution and then measuring its efficiency. it provides the actual time taken by the program. This is useful in practice but Apriori is always better.
There is corresponding guarantee that any algorithm that is better in performance in its apriori analysis will be better in performance in its posterior analysis

Instruction count as a measure of complexity

Is not related to type of input (input size in terms of number of bytes like in factoring, or the number of inputs or the type of input like in sorting)
Intelligent compilers can make your count useless
Language/ Compiler/interpreter issues will also play a role
Parallel/ pipelining/superscalar executions
Different instructions may be very differently loaded in terms of resource requirements
Can be a first preliminary indication to compare the size of two algorithms but should not cheat you
Only 10% of the instructions may be actually responsible for the 90% of resource usage

Micro and macro analysis

Micro Analysis

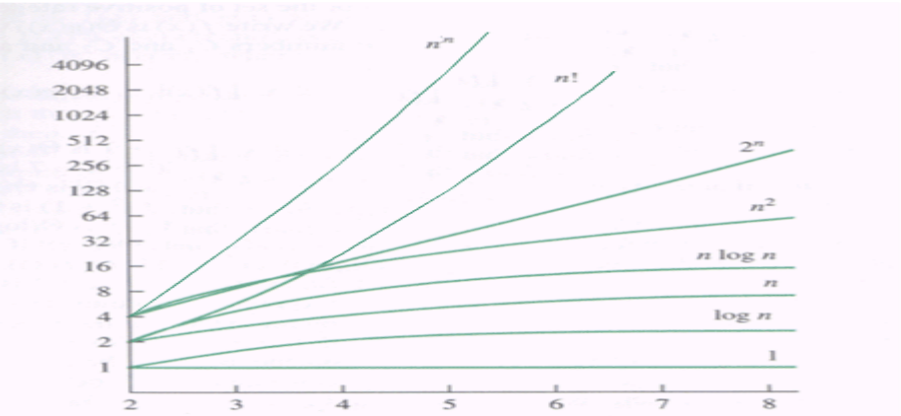
- To count each and every operation of the program.
- Detailed, Takes more time and is complex and tedious
- (Average lines of codes are in the range of 3 to 5 million lines)
- Those operations which are not dependent upon the size or number of the input will take constant time and will not participate in the growth of the time or space function, So they need not be part of our analysis

Macro analysis

- Deals with selective instructions which are dominant & costliest. Selection of right instructions is very important
- Comparisons and Swapping are basic operations in sorting algorithms
- Arithmetic operations are basic operations in math algorithms
- Comparisons are basic operations in searching algorithms
- Multiplication and Addition are basic operations in matrix multiplication

❼

L8: Relative Goodness of functions



Function/value	2	4	8	16	32	64	128	256	512	1024	2048=2^{11}	2^{20}	2^{30}	2^{40}
Lg n	1	2	3	4	5	6	7	8	9	10	11	12	30	40
N	2	4	8	16	32	64	128	256	512	1024	2048	4096	2^{30}	2^{40}
N logn	2	8	24	64	160	384	896	2048	4608	10240	22528	49152	2^{35}	2^{46}
N^2	4	16	64	256	1024	4096	16384	65536	262144	1048576	4194304	2^{40}	2^{60}	2^{80}
N^3	8	64	512	4096	32768	262144	2097152	16777216	2^{27}	2^{30}	2^{33}	2^{60}	2^{90}	2^{120}
N^{10}	2^{10}	2^{20}	2^{30}	2^{40}	2^{50}	2^{60}	2^{70}	2^{80}	2^{90}	2^{100}	2^{110}	2^{200}	2^{300}	2^{400}
2^N	4	16	256	2^{16}	2^{32}	2^{64}	2^{132}	2^{256}	2^{512}	2^{1024}	2^{2048}	∞	∞	∞
N!	2	24	40320	20922789888800	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞

Show some functions take less values for small numbers but grow very rapidly afterwards
Some values go beyond the available resource limits of computers
Assuming one processor executes one trillion instructions per second

Function/size of the problem	1	10	50	60	100	1000	10^4	10^5	10^6	10^{12}
Log n	10^{-12}	$3*10^{-11}$	$5*10^{-11}$	$5*10^{-11}$	$6*10^{-11}$	10^{-11}	$3.3*10^{-10}$	$6.5*10^{-10}$	10^{-10}	10^{-7}
N	10^{-12}	10^{-11}	$5*10^{-10}$	$6*10^{-10}$	10^{-10}	10^{-9}	10^{-08}	10^{-07}	10^{-06}	1 sec
N lgn	10^{-12}	$3*10^{-10}$	$5*10^{-9}$	$6*10^{-08}$	$6.64*10^{-9}$	$9.965*10^{-8}$	$1.3*10^{-7}$	$1.5*10^{-04}$	$1.8*10^{-03}$	40 sec
N^2	10^{-12}	10^{-10}	$2.5*10^{-9}$	$3.3*10^{-9}$	10^{-08}	10^{-06}	10^{-04}	10^{-02}	1	317 centuries
N^3	10^{-12}	10^{-09}	$8*10^{-7}$	10^{-7}	10^{-06}	10^{-03}	1	16.6 minutes	12 days	∞
N^{10}	10^{-12}	10^{-02}	1.13 days	7 days	3.17 years	∞	∞	∞	∞	∞
2^N	$0.5*10^{-11}$	10^{-09}	16 min	12 days	∞	∞	∞	∞	∞	∞
N!	$1*10^{-12}$	$3.6*10^{-6}$ n=16 5 minus	∞	∞	∞	∞	∞	∞	∞	∞

Show- The utility of log n and relative difficulty of the functions. What are the practical limits of every function?

	1 second	1minute	1 hour	1 month	1 year
Logn	$2^{1000000000000}$	$2^{60000000000000}$	∞	∞	∞
N	10^{12}	$6*10^{13}$	$3.6*10^{15}$	$2.5*10^{18}$	$3.1*10^{19}$
Nlogn	10^{12}	$6*10^{13}$	$3.6*10^{15}$	$2.5*10^{18}$	$3.1*10^{19}$
N^2	10^6	$7.74*10^6$	$6.0*10^7$	$1.6*10^9$	$5.6*10^9$
N^3	10^4	$3.9*10^4$	$1.53*10^5$	$1.37*10^6$	$3.15*10^6$
N^{10}	16	24	36	70	90
2^N	40	46	52	61	65
N!	14	15	16	18	20

Why lower order terms are not required

As seen in the previous tables if there is a function of higher growth term present then lower growth function becomes negligible for the large values of n, so all lower growth functions in the expression can be discarded. The constant in the highest growth term can be discarded because that does not participate in the growth of the function. This value can also change with change in implementation specifics. But anyhow the importance of it cannot be overlooked.

❽

L9: Getting Bounds on Algorithm Complexity

Worst-average-best case analysis

Worst Case

- Goodness of an algorithm is most often expressed in terms of its worst-case running time.
- Need for a bound on one's pessimism, Every Body needs a guarantee. This is the maximum time an algorithm will take on a given input size
- Ease of calculation of worst-case times
- In case of critical systems we cannot rely on average or best case times
- Worst Case for all sorting problems is when the inputs are in the reverse order

Average Case

- Very difficult to compute
- Average-case running times are calculated by first arriving at an understanding of the average nature of the input, and then performing a running-time analysis of the algorithm for this configuration
- Needs assumption of statistical and probabilistic distribution of input e.g. uniform probability distribution
- It is supposed that all inputs are equally likely to occur
- If we have the same worst case time for two algorithms then we can go for average case analysis
- If the average case is also same then we can go for micro analysis or empirical analysis

Best Case

- Not used in general
- Best case may never occur
- Can be a bogus or cheat algorithm that is otherwise very slow but works well on a particular input
- A particular input is likely to occur more than 90% of the time then we can go for a customized algorithm for that input
- Best Case for all sorting problems is that sequence is already in the sorted sequence

Asymptotic analysis

Asymptotic analysis means studying the behavior of the function when n→ infinity or very large
Problems size will keep on increasing so asymptotic analysis is very important
Limiting behavior
Nested loops should be analyzed inside out. The total running time for a statement inside innermost loop is given by its running time multiplied by the product of the sizes of all for loops
The running time of an if/else statement is not more than the running time of the test, plus the larger of the running times of statements contained inside if and else conditions
We are more concerned with large input size because small size inputs will not very much in running time

Big-Oh, Big-Omega and Big-Theta Analysis

Independent of hardware and software and valid for any input

$n^2 + 3n + 4 \leq 2n^2$ for all $n_0 > 10$ is $O(n^2)$,

$3 \log n + \log \log n \leq 4 \log n$ for all $n_0 > 2$ $O(\log n)$

$3n^3 + 20n^2 + 5 \leq 4n^3$ for all $n_0 > 21$ $O(n^3)$

$7n - 2 \leq 7n$ for all $n_0 > 1$ $O(n)$

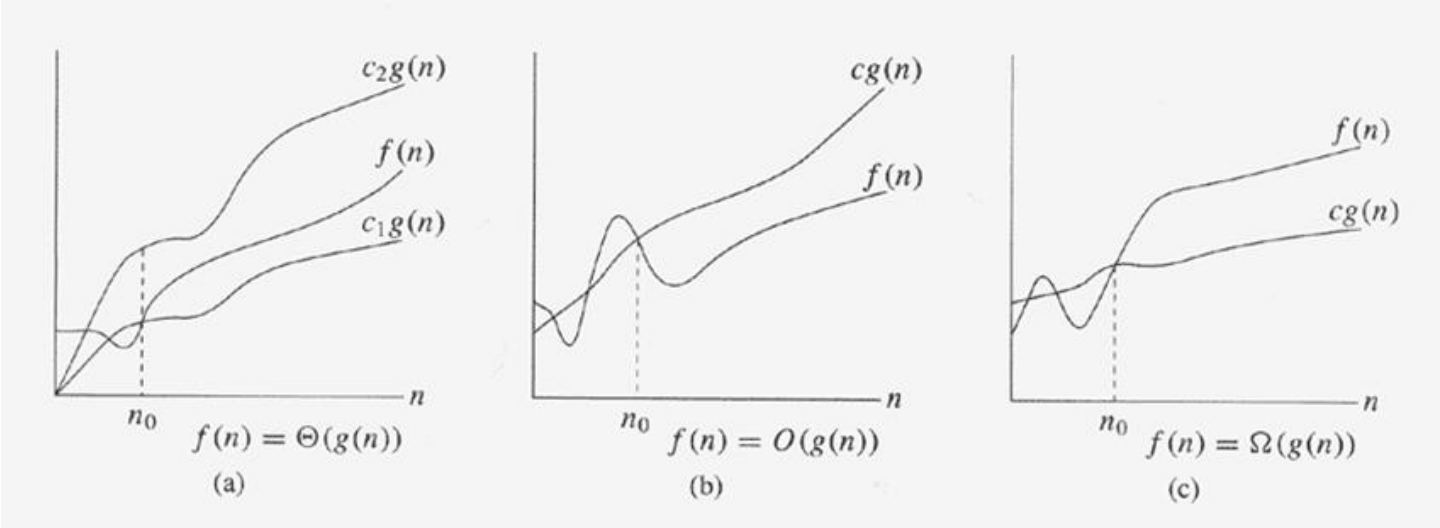
$a_0 n^0 + a_1 n^1 + a_2 n^2 + \ldots\ldots a_k n^k$ $O(n^k)$

n^2 is not $O(n)$

$n^2 \leq cn$ *and* $n \leq c$ cannot be satisfied since *c* must be a constant

f(n) = O(g(n)) if there are positive constants c and n_0 such that $f(n) \leq cg(n)$ for all $n \geq n_0$ and $c>0$ This notation is known as Big-Oh notation

$n^2 + O(n) = O(n^2)$

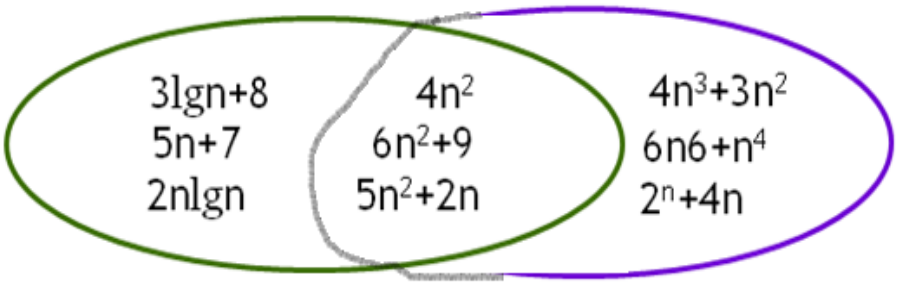


how to find the constant for the sake of argument

$n^2 + 3n + 4 \geq n^2$ for all $n_0 > 1$	$O(n^2)$,
$3 \log n + \log \log n \geq \log n$ for all $n_0 > 2$	$O(\log n)$
$3n^3 + 20n^2 + 5 \geq n^3$ for all $n_0 > 1$	$O(n^3)$
$7n-2 \geq n$ for all $n_0 > 1$	$O(n)$
$a_0n^0+a_1n^1+a_2n^2+\ldots\ldots a_kn^k$	$O(n^k)$

f(n) = Ω(g(n)) if there are positive constants c and n_0 such that f(n) >= cg(n) for all n >= n_0. This notation is known as Big-Omega notation

$O(n^2)$ $\theta(n^2)$ $\Omega(n^2)$



f(n) = Θ(g(n)) if there are positive constants c_1, c_2 and n_0 such that $c_1g(n) <= f(n) <= c_2g(n)$, for all n >= n_0. This notation is known as Big-Theta notation

$20n^2+17n+9$ belongs to $\Theta(n^2)$

$8n+2$ does not belong to $\Theta(n^2)$

n^3 does not belong to the $\Theta(n^2)$

Shrinking lower and upper bounds is a area of research

Instead of an instance we are giving T(n) for a function of n as a time to run an algorithm or goodness of an algorithm

$\Theta(g(n))=O(g(n)) \cap \Omega(g(n))$

Notions of upper bound and lower bound and the tightness of these bounds

Lack of the actual T(n) leads us to expressing these terms in terms of Oh or Omega notation

❾

L10: Finding Greatest Common Divisor

Naïve method

Input: Two integer's m an n
Output: Greatest factor that divides both
1. Factorize m: find primes
   m1, m2, m3…… such that m= m1*m2*m3…
2. Factorize n : find primes
   n1, n2, n3…… such that n= n1*n2*n3…
3. Find common factors in both multiply and print result

Additional Benefits: we had found prime numbers also on the sides

Euclid Algorithm

It is using a clever technique which was explored after studying the deeper properties of the numbers involved.
1 divide m by n and let r be the remainder
2 if r=0, algorithm terminates; n is the answer
3 set m=n; n=r; and go to step 1

Algorithm without swapping

1 divide m by n and let r be the remainder
2 if r=0, algorithm terminates; n is the answer
3 divide n by r and let m be the remainder
4 if m=0, algorithm terminates; r is the answer
5 divide r by m and let n be the remainder
6 if n=0, algorithm terminates; m is the answer
7. go to step 1

Algorithm without additional variable

1. Divide m by n and let m be the remainder
2. If m=0, the algorithm terminates with answer n
3. Divide n by m and let n be the remainder
4. if n=0 the algorithm terminates with answer m; otherwise go to step 1
If we go deeper them let m=ag and n=bg also m=x*n+r
$T(a, b) = 1 + T(b, r_0) = 2 + T(r_0, r_1) = \ldots = N + T(r_{N-2}, r_{N-1}) = N + 1$
If the Euclidean algorithm requires N steps for a pair of natural numbers $a > b > 0$, the smallest values of a and b for which this is true are the Fibonacci numbers F_{N+2} and F_{N+1}, respectively.
Worst-case number of steps
By induction. If $N = 1$, b divides a with no remainder; the smallest natural numbers for which this is true is $b = 1$ and $a = 2$, which are F_2 and F_3, respectively.
Now assume that the result holds for all values of N up to $M - 1$. The first step of the M-step algorithm is $a = q_0 b + r_0$, and the second step is $b = q_1 r_0 + r_1$. Since the algorithm is recursive, it required $M - 1$ steps to find GCD(b, r_0) and their smallest values are F_{M+1} and F_M. The smallest value of a is therefore when $q_0 = 1$, which gives $a = b + r_0 = F_{M+1} + F_M = F_{M+2}$.
For N steps, then b is greater than or equal to F_{N+1} which in turn is greater than or equal to φ^{N-1}, where φ is the golden ratio. Since $b \geq \varphi^{N-1}$, then $N - 1 \leq \log_\varphi b$. Since $\log_{10}\varphi > 1/5$, $(N - 1)/5 < \log_{10}\varphi \log_\varphi b = \log_{10} b$. Thus, $N \leq 5 \log_{10} b$. Thus, the Euclidean algorithm always needs less than $O(h)$ divisions, where h is the number of digits in the smaller number b.

Euclid algorithm

```
#include<stdio.h>
int main()
                        {
                                int i,m,n,gcd=1;
                                scanf("%d%d",&m,&n);
                                for(i=2; i<=n;)
                                {
                                        if ((m%i) == 0 && (n%i) == 0)
                                {
                                        gcd=gcd*i;
                                        m=m/i;
                                        n=n/i;
                                }
                                else
                                        i++;
                                }
                                printf("GCD of the number is %d\n",gcd );
                                return(0);
                        }
```

Recursive algorithm

GCD(m,n) = GCD(n,m mod n)

```
#include<stdio.h>
int main()
                        {
                                int m,n,gc=1;
                                scanf("%d%d",&m,&n);
                                gc=GCD(m,n);
                                printf("GCD of the number is %d\n",gc );
                        return(0);
                        }
int GCD(m,n)
{
        int k;
        if (m == 0) return n;
                else if (n == 0) return m;
                        k = m%n;
                        printf("%d",k);
                GCD(n,k);
```

Iterative algorithm without swapping

```
#include<stdio.h>
int main()
{
int m,n,r;
scanf("%d%d",&m,&n);
for(;;)
{
r=m%n;
if(r==0)
{
printf("GCD is %d",n);
break;
}
```

```
m=n%r;
if(m==0)
{
printf("GCD is %d",r);
break;
}
n=r%m;
if(n==0)
{
printf("GCD is %d",m);
break;
}
}
return(0);
}
```

Algortihm without swapping and additional varibales

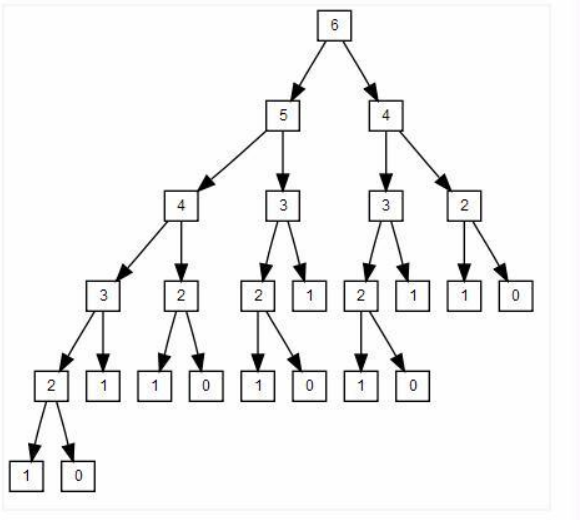
```
#include<stdio.h>
int main()
{
int m,n;
scanf("%d%d",&m,&n);
for(;;)
{
m=m%n;
if(m==0)
{
printf("GCD is %d",n);
break;
}
n=n%m;
if(n==0)
{
printf("GCD is %d",m);
break;
}
}
return(0);
}
#include<stdio.h>
int main()
{int m,n;
scanf("%d%d",&m,&n);
while(m!=n)
{
if(m>n)
m=m-n;
else
n=n-m;
}
printf("GCD is %d",m);
return(0);
}
```

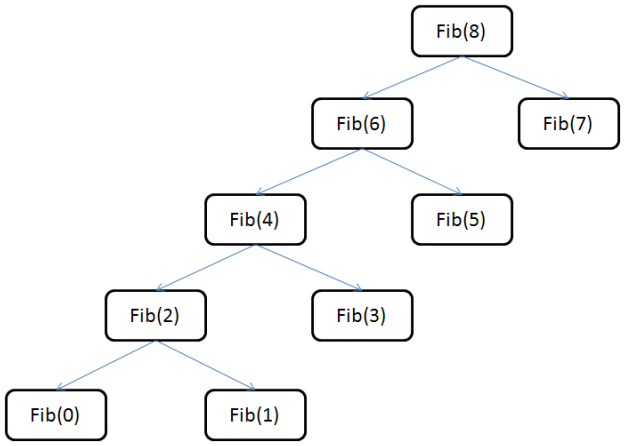
Discussion on different phases of any recursion algorithm mainly Base condition, Decomposition and recomposition. Also how recursion may be useful in making the code shorter and simpler, but in some cases it may prove very costly.

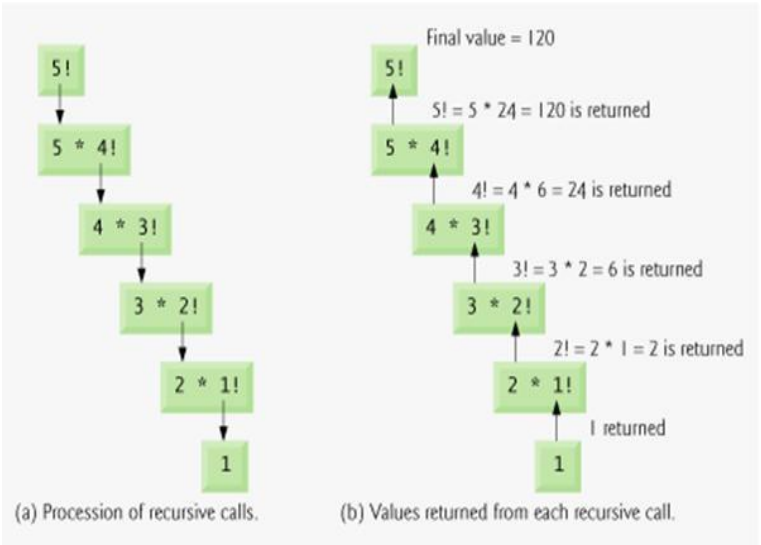
Example of fibonacci sequence like fib(6) breaking into fib(5),fib(4) and so on and making the number of function calls of the exponential order . While the iterative fibonacci sequence will be of the order of O(n). Fibonacci sequence can also be solved in logn time by recursive squaring method

Fn+1 Fn = Fn Fn-1 1 1 = x x^n is to find the nth fibonacci number It is θ(logn)

Fn Fn-1 Fn-1 Fn-2 1 0







(a) Procession of recursive calls. (b) Values returned from each recursive call.