Matrix Chain Multiplication-Module 4

Elizabeth Isaac
Department of Computer Science and Engineering
MACE

Dynamic programming

- Dynamic programming is an algorithm design technique that can be used when the solution to a problem can be viewed as the result of a sequence of decisions.
- Used to solve problems with overlapping sub problems
- Solve each sub problem only once.
- Store the results of sub problems, so that we do not have to re-compute them when needed later.

Dynamic programming

- Basic principle behind dynamic programming is "**Principle of Optimality** "
- An optimal sequence of decision has the property that whatever the initial state and conditions are, the remaining decisions must constitute an optimal decision sequence with regard to the state resulting from the first decision

Matrix chain multiplication

- **Matrix chain multiplication** is an optimization problem concerning the most efficient way to multiply a given sequence of matrices.
- There are many options because matrix multiplication is associative.
- No matter how the product is parenthesized, the result obtained will remain the same.
- The problem may be solved using dynamic programming.

$$A_1 \times A_2 \times A_3 \times A_4$$

Matrix chain multiplication

```
MATRIX-MULTIPLY(A, B)
1  if A.columns ≠ B.rows
2      error "incompatible dimensions"
3  else let C be a new A.rows × B.columns matrix
4      for i = 1 to A.rows
5          for j = 1 to B.columns
6              c_ij = 0
7              for k = 1 to A.columns
8                  c_ij = c_ij + a_ik · b_kj
9      return C
```

- We can multiply two matrices A and B only if they are ***compatible***
- the number of columns of A must equal the number of rows of B. If A is a p x q and B is a q x r matrix,
- resulting matrix C is a p x r matrix
- number of scalar multiplications- pqr

Matrix chain multiplication

- four matrices A, B, C, and D, there are five possible options:
  - $((AB)C)D = (A(BC))D = (AB)(CD) = A((BC)D) = A(B(CD))$.

For example A is a 10×30 matrix,

B is a 30×5 matrix

C is a 5×60 matrix

$(AB)C$ *needs* $(10 \times 30 \times 5) + (10 \times 5 \times 60)$

$= 1500 + 3000$

$= 4500$ operations

$A(BC)$ needs $(30 \times 5 \times 60) + (10 \times 30 \times 60)$

$= 9000 + 18000$

$= 27000$ operations.

Matrix chain multiplication

- Dynamic programming method to determine how to optimally parenthesize a matrix chain.

follow the four-step sequence

- 1. Characterize the structure of an optimal **parenthesization**.

A matrix series

$$A_{i.....j}$$

$$(A_{i.....k})(A_{k+1.....j})$$

- 2. Recursively define the value of an optimal solution.
  - recursive definition for the minimum cost of parenthesizing the product of $A_i \ A_{i+1} \ \ldots\ldots\ldots\ A_j$

$$m[i,j] = \begin{cases} 0 & \text{if } i = j, \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\} & \text{if } i < j. \end{cases}$$

Matrix chain multiplication

- 3. Computing **the optimal costs**

```
MATRIX-CHAIN-ORDER(p)
 1  n = p.length − 1
 2  let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
 3  for i = 1 to n
 4      m[i, i] = 0
 5  for l = 2 to n                    // l is the chain length
 6      for i = 1 to n − l + 1
 7          j = i + l − 1
 8          m[i, j] = ∞
 9          for k = i to j − 1
10              q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
11              if q < m[i, j]
12                  m[i, j] = q
13                  s[i, j] = k
14  return m and s
```

The algorithm first computes $m[i, i] = 0$ for $i = 1, 2, \ldots, n$ (the minimum costs for chains of length 1) in lines 3–4. It then uses recurrence (15.7) to compute $m[i, i + 1]$ for $i = 1, 2, \ldots, n - 1$ (the minimum costs for chains of length $l = 2$) during the first execution of the **for** loop in lines 5–13. The second time through the loop, it computes $m[i, i+2]$ for $i = 1, 2, \ldots, n-2$ (the minimum costs for chains of length $l = 3$), and so forth. At each step, the $m[i, j]$ cost computed in lines 10–13 depends only on table entries $m[i, k]$ and $m[k + 1, j]$ already computed.

Matrix chain multiplication

- 4. Compute the value of an optimal solution.

```
PRINT-OPTIMAL-PARENS(s, i, j)
1  if i == j
2       print "A"_i
3  else print "("
4       PRINT-OPTIMAL-PARENS(s, i, s[i, j])
5       PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
6       print ")"
```

Time Complexity of matrix chain multiplication is : **O(n^3)**

Matrix chain multiplication

$$m[i,j] = \begin{cases} 0 & \text{if } i = j\ , \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\} & \text{if } i < j\ . \end{cases}$$

We want to start with i=j , then i<j starting with a spread of 1, working our way up

I \ j	1	2	3	4	5
1	0				
2	x	0			
3	x	x	0		
4	x	x	x	0	
5	x	x	x	x	0

Matrix chain multiplication

- $M[i,j] = \begin{cases} 0 \text{ if } i = j \\ \min_{i<=k<j} \{M[i,j] = M[i,k] + M[k+1,j] + p_{i-1} p_k p_j\} \end{cases}$

Step 1: Fill the table for i = j

Matrix chain multiplication

- $M[i,j] = \begin{cases} 0 \text{ if } i = j \\ \min_{i<=k<j} \{M[i,j] = M[i,k] + M[k+1,j] + p_{i-1} p_k p_j\} \end{cases}$

Step 1: Fill the table for:

i = 1, j = 2

i = 2, j = 3

i = 3, j = 4

i = 4, j = 5

Matrix chain multiplication

- $M[i,j] = \begin{cases} 0 \text{ if } i = j \\ \min_{i<=k<j} \{M[i,k] + M[k+1,j] + p_{i-1} p_k p_j\} \end{cases}$

$$A_1 * A_2 * A_3 * A_4 * A_5$$

4*10 10*3 3*12 12*20 20*7

$p_0 p_1$ $p_1 p_2$ $p_2 p_3$ $p_3 p_4$ $p_4 p_5$

$M[1,2] = \min_{1<=k<2} \{M[1,1] + M[1+1,2] + p0p1p2\}$

$M[1,2] = \min_{1<=k<2} \{0+0+4*10*3\}$

$M[1,2] = 120$

Matrix chain multiplication

I \ j	1	2	3	4	5
1	0	120			
2	x	0			
3	x	x	0		
4	x	x	x	0	
5	x	x	x	x	0

Matrix chain multiplication

$$m[i,j] = \begin{cases} 0 & \text{if } i = j\ , \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\} & \text{if } i < j\ . \end{cases}$$

- $M[2,3] = \min_{2<=k<3}\{M[2,2] + M[2+1,3] + p_1p_2p_3\}$

  $M[2,3] = \min_{2<=k<3}\{0+0+10*3*12\}$

  $M[2,3] = 360$

- $M[3,4] = \min_{3<=k<4}\{M[3,3] + M[3+1,4] + p_2p_3p_4\}$

  $M[3,4] = \min_{3<=k<4}\{0+0+3*12*20\}$

  $M[2,3] = 720$

- $M[4,5] = \min_{4<=k<5}\{M[4,4] + M[4+1,5] + p_3p_4p_5\}$

  $M[4,5] = \min_{4<=k<5}\{0+0+12*20*7\}$

  $M[1,2] = 1680$

Matrix chain multiplication

i \ j	1	2	3	4	5
1	0	120			
2	x	0	360		
3	x	x	0	720	
4	x	x	x	0	1680
5	x	x	x	x	0

Matrix chain multiplication

s

i \ j	1	2	3	4	5
1	0	1			
2	x	0	2		
3	x	x	0	3	
4	x	x	x	0	4
5	x	x	x	x	0

Matrix chain multiplication

$$m[i,j] = \begin{cases} 0 & \text{if } i = j, \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\} & \text{if } i < j. \end{cases}$$

- M[1,3] = min $_{1<=k<3}$

k=1

= M[1,1] + M[1+1,3] + $p_0p_1p_3$

= 0 + 360 + 4 *10 * 12

= 840

k= 2

= M[1,2] + M[2+1,3] + $p_0p_2p_3$

= 120 + 0 + 4 * 3 * 12

= 264

i	1	2	3	4	5
1	0	120			
2	x	0	360		
3	x	x	0	720	
4	x	x	x	0	1680
5	x	x	x	x	0

Matrix chain multiplication

i \ j	1	2	3	4	5
1	0	120	264		
2	x	0	360		
3	x	x	0	720	
4	x	x	x	0	1680
5	x	x	x	x	0

$$m[i,j] = \begin{cases} 0 & \text{if } i = j\,, \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\} & \text{if } i < j\,. \end{cases}$$

- M[2,4] = min $_{2<=k<4}$

k = 2

= M[2,2] + M[2+1,4] + $p_1p_2p_4$

= 0 + 720 + 10 * 3 *20

= 1320

k = 3

= M[2,3] + M[3+1,4] +$p_1p_3p_4$

= 360 + 0 + 10 * 12 *20

= 2760

Matrix chain multiplication

i \ j	1	2	3	4	5
1	0	120	264		
2	x	0	360	1320	
3	x	x	0	720	
4	x	x	x	0	1680
5	x	x	x	x	0

$$m[i,j] = \begin{cases} 0 & \text{if } i = j, \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\} & \text{if } i < j. \end{cases}$$

- M[3,5] = min $_{3<=k<5}$

k = 3

= M[3,3] + M[3+1,5] + $p_2p_3p_5$

= 0 + 1680 + 3 * 12 * 7

= 1932

k = 4

= M[3,4] + M[4+1,5] +$p_2p_4p_5$

= 720 + 0 + 3 * 20 * 7

= 1140

Matrix chain multiplication

i \ j	1	2	3	4	5
1	0	120	264		
2	x	0	360	1320	
3	x	x	0	720	1140
4	x	x	x	0	1680
5	x	x	x	x	0

Matrix chain multiplication

i \ j	1	2	3	4	5
1	0	1	2		
2	x	0	2	2	
3	x	x	0	3	4
4	x	x	x	0	4
5	x	x	x	x	0

$$m[i,j] = \begin{cases} 0 & \text{if } i = j, \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\} & \text{if } i < j. \end{cases}$$

- M[1,4] = min $_{1<=k<4}$

k = 1

= M[1,1] + M[1+1,4] + $p_0p_1p_4$

= 0 + 1320 + 4 * 10 * 20

= 2120

k = 2

= M[1,2] + M[2+1,4] +$p_0p_2p_4$

= 120 + 720 + 4 * 3 * 20

= 1080

k = 3

= M[1,3] + M[3+1,4] +$p_0p_3p_4$

= 264 + 0 + 4 * 12 * 20

= 1224

Matrix chain multiplication

i \ j	1	2	3	4	5
1	0	120	264	1080	
2	x	0	360	1320	
3	x	x	0	720	1140
4	x	x	x	0	1680
5	x	x	x	x	0

$$m[i,j] = \begin{cases} 0 & \text{if } i = j, \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\} & \text{if } i < j. \end{cases}$$

- M[2,5] = min $_{2<=k<5}$

k = 2

= M[2,2] + M[2+1,5] + $p_1p_2p_5$

= 0 + 1140 + 10 * 3 * 7

= 1350

k = 3

= M[2,3] + M[3+1,5] +$p_1p_3p_5$

= 360 + 1680 + 10 * 12 * 7

= 2880

k = 4

= M[2,4] + M[4+1,5] +$p_1p_4p_5$

= 1320 + 0 + 10 * 20 * 7

= 2720

j / i	1	2	3	4	5
1	0	120	264	1080	
2	x	0	360	1320	1350
3	x	x	0	720	1140
4	x	x	x	0	1680
5	x	x	x	x	0

s

i \ j	1	2	3	4	5
1	0	1	2	2	
2	x	0	2	2	2
3	x	x	0	3	4
4	x	x	x	0	4
5	x	x	x	x	0

$$m[i,j] = \begin{cases} 0 & \text{if } i = j, \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\} & \text{if } i < j. \end{cases}$$

- M[1,5] = min $_{1<=k<5}$

k = 1

= M[1,1] + M[1+1,5] + $p_0p_1p_5$

= 0 + 1350 + 4 * 10 * 7

= 1630

k = 2

= M[1,2] + M[2+1,5] +$p_0p_2p_5$

= 120 + 1140 + 4 * 3 * 7

= 1344

k = 3

= M[1,3] + M[3+1,5] +$p_0p_3p_5$

= 264 + 1680 + 4 * 12 * 7

= 2280

k = 4

= M[1,4] + M[4+1,5] + $p_0p_4p_5$

= 1080 + 0 + 4 * 20 * 7

= 1640

j / i	1	2	3	4	5
1	0	120	264	1080	1344
2	x	0	360	1320	1350
3	x	x	0	720	1140
4	x	x	x	0	1680
5	x	x	x	x	0

s

i \ j	1	2	3	4	5
1	0	1	2	2	2
2	x	0	2	2	2
3	x	x	0	3	4
4	x	x	x	0	4
5	x	x	x	x	0

We now know that we can multiply A_1 to A_5 in a few as 1344 multiplication operations!

But where do we put our brackets?

We must focus on the selected k values

- $M[i,j] = 0$ if $i = j$

  $\min_{i<=k<j} \{M[i,j] = M[i,k] + M[k+1,j] + p_{i-1} p_k p_j\}$

$$A_1 * A_2 * A_3 * A_4 * A_5$$

4*10 10*3 3*12 12*20 20*7

K = 2

$M[1,5] = M[1,2] + M[3,5] + p_0 p_2 p_5$

$(A_1 * A_2) * (A_3 * A_4 * A_5)$

K = 4

$M[3,5] = M[3,4] + M[5,5] + p_2 p_4 p_5$

$(A_1 * A_2) * ((A_3 * A_4) * A_5)$

- $M[i,j] = 0$ if $i = j$

$$\min_{i<=k<j} \{M[i,j] = M[i,k] + M[k+1,j] + p_{i-1} p_k \quad p_j\}$$

(A1 * A2) ((A3 * A4) A5)

4*10 10*3 3*12 12*20 20*7

(A1*A2)

4*10*3 =120 // 4*3

(A3*A4)

3*12* 20 =720// 3*20

(A3*A4)*A5

3*20*7=420//3*7

(A1*A2) * (A3*A4)*A5

4*3*7=84

120+720+420+84=1344

THANK YOU

Module-1

Elizabeth Isaac
Assistant Professor, M A College of Engg
DCS

Course Outcomes: After the completion of the course the student will be able to

CO#	CO
CO1	Analyze any given algorithm and express its time and space complexities in asymptotic notations. **(Cognitive Level: Apply)**
CO2	Derive recurrence equations and solve it using Iteration, Recurrence Tree, Substitution and Master's Method to compute time complexity of algorithms. **(Cognitive Level: Apply)**
CO3	Illustrate Graph traversal algorithms & applications and Advanced Data structures like AVL trees and Disjoint set operations. **(Cognitive Level: Apply)**
CO4	Demonstrate Divide-and-conquer, Greedy Strategy, Dynamic programming, Branch-and Bound and Backtracking algorithm design techniques **(Cognitive Level: Apply)**
CO5	Classify a problem as computationally tractable or intractable, and discuss strategies to address intractability **(Cognitive Level: Understand)**
CO6	Identify the suitable design strategy to solve a given problem. **(Cognitive Level: Analyze)**

Syllabus

Module-1 (Introduction to Algorithm Analysis)

Characteristics of Algorithms, Criteria for Analysing Algorithms, Time and Space Complexity - Best, Worst and Average Case Complexities, Asymptotic Notations - Big-Oh (O), Big- Omega (Ω), Big-Theta (Θ), Little-oh (o) and Little- Omega (ω) and their properties. Classifying functions by their asymptotic growth rate, Time and Space Complexity Calculation of simple algorithms.

Analysis of Recursive Algorithms: Recurrence Equations, Solving Recurrence Equations – Iteration Method, Recursion Tree Method, Substitution method and Master's Theorem (Proof not required).

Module–2 (Advanced Data Structures and Graph Algorithms)

Self Balancing Tree - AVL Trees (Insertion and deletion operations with all rotations in detail, algorithms not expected); Disjoint Sets- Disjoint set operations, Union and find algorithms.

DFS and BFS traversals - Analysis, Strongly Connected Components of a Directed graph, Topological Sorting.

Module–3 (Divide & Conquer and Greedy Strategy)

The Control Abstraction of Divide and Conquer- 2-way Merge sort, Strassen's Algorithm for Matrix Multiplication-Analysis. The Control Abstraction of Greedy Strategy- Fractional Knapsack Problem, Minimum Cost Spanning Tree Computation- Kruskal's Algorithms - Analysis, Single Source Shortest Path Algorithm - Dijkstra's Algorithm-Analysis.

Module-4 (Dynamic Programming, Back Tracking and Branch & Bound))

The Control Abstraction- The Optimality Principle- Matrix Chain Multiplication-Analysis, All Pairs Shortest Path Algorithm - Floyd-Warshall Algorithm-Analysis. The Control Abstraction of Back Tracking – The N Queen's Problem. Branch and Bound Algorithm for Travelling Salesman Problem.

Module-5 (Introduction to Complexity Theory)

Tractable and Intractable Problems, Complexity Classes – P, NP, NP- Hard and NP-Complete Classes- NP Completeness proof of Clique Problem and Vertex Cover Problem- Approximation algorithms- Bin Packing, Graph Coloring. Randomized Algorithms (Definitions of Monte Carlo and Las Vegas algorithms), Randomized version of Quick Sort algorithm with analysis.

Algorithm

- Desirable properties of an algorithm:
  - Input - zero or more quantities are externally supplied
  - Output - at least one output is produced
  - Definiteness - each instruction is clear and unambiguous
  - Finiteness - the algorithm terminates after a finite number of steps
  - Effectiveness - every instruction must be very basic so that it can be carried out, in principle, by a person using only pencil and paper

Basics

- Four research areas in algorithms
- 1 How to devise algorithms?
- 2 How to validate algorithms?
- 3 How to analyse algorithms?
- 4 How to test algorithms?
- We will spend more time on algorithm analysis and design

Basics

- Development of Algorithm
  - Statement of the problem
  - Development of Model
  - Design of Algorithm
  - Correctness of the Algorithm
  - Implementation
  - Analysis of Complexity
  - Program Testing(profiling &debugging)
  - Comparison with the existing Algorithm
  - Documentation

Performance Analysis

- Performance evaluation
  - A priori estimates (*Performance analysis*)
  - A posteriori testing (*Performance testing*)

Performance Analysis

- Two criteria are used to judge algorithms:

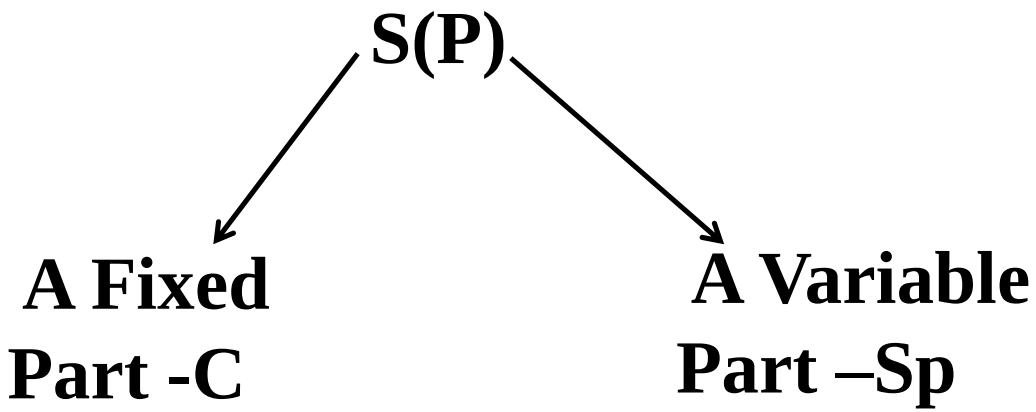
(i) time complexity

(ii) space complexity.

- Time Complexity of an algorithm is the amount of CPU time it needs to run to completion.
- Space Complexity of an algorithm is the amount of memory it needs to run to completion.

Space Complexity

- Memory space S(P) needed by a program P, consists of two components



S(P)=C+Sp

Space Complexity

- A fixed part that is independent of the characteristics (e.g., number, size) of the inputs and outputs - typically includes the instruction space (i.e., space for the code), space for

  simple variables and fixed-size component variables, space for constants etc
- A variable part that consists of the space needed by components whose size is dependent on the particular problem instance being solved - i.e., on the instance characteristics

Space Complexity: Example 1

```
1.  Algorithm abc (a, b, c)
2.  {
3.     return a+b+b*c+(a+b-c)/(a+b)+4.0;
4.  }
```

Space Complexity: Example 2

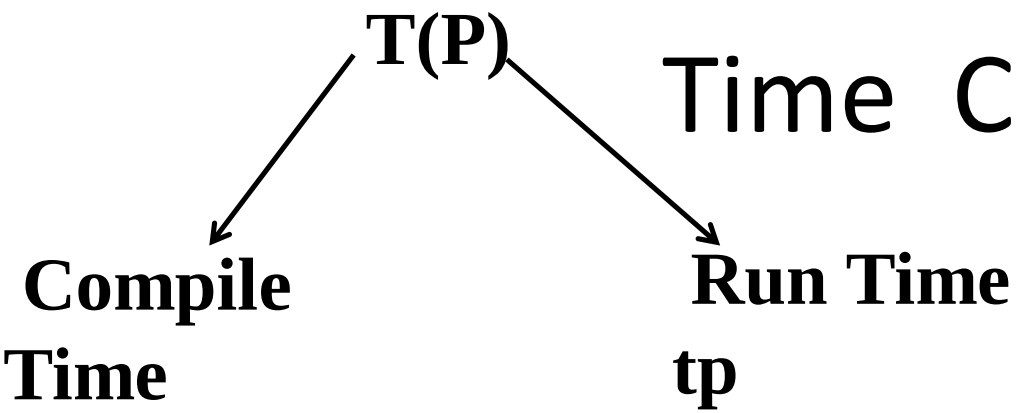
```
1.  Algorithm Sum(a[], n)
2.  {
3.      s:= 0.0;
4.      for i = 1 to n do
5.          s := s + a[i];
6.      return s;
7.  }
```

$S(Sum) \geq (n+3)$

Space Complexity: Example 3

```
1.  Algorithm RSum(a, n)
2.  {
3.      if (n<=0) then return 0.0;
4.      else return Rsum(a,n-1)+a[n];
5.  }
```

Time Complexity



Time T(P) taken by a program P is the sum of the compile time and the run (or execution) time

Compile time - independent of instance characteristics; also a compiled program can be run several times without recompilation

Consequently, we concern ourselves with just the run time of a program

Run time is denoted by tp (instance characteristic)

$$t_p(n)=c_aADD(n)+ c_sSUB(n)+ c_mMUL(n)+ c_dDIV(n)+\ldots\ldots\ldots\ldots$$

Program step count

A program step is loosely defined as a syntactically or semantically meaningful segment of a program that has an execution time that is independent of the instance characteristics

- Given the difficulty of determining the exact number of additions, subtractions, and so on, that are needed to solve a problem instance with characteristics given by n, we can lump all the operations together and obtain a count for the total number of operations
- We can go one step further and count only the number of

program steps

- A program step is loosely defined as a syntactically or semantically meaningful segment of a program that has an execution time that is independent of the instance characteristics

Program step count

- a+b+b*c+(a+b)/(a-b) → one step;
- comments → zero steps;
- `while (<expr>) do` → step count equal to the number of times <expr> is executed.
- `for i=<expr> to <expr1> do` → step count equal to number of times <expr1> is checked.

Time Complexity: Example 1

```
1.  Algorithm abc (a, b, c)
2.  {
3.    return a+b+b*c+(a+b-c)/(a+b)+4.0;
4.  }
```

Time Complexity: Example 1

	Statements
1	Algorithm Sum(a[],n)
2	{
3	S = 0.0;
4	for i=0 to n do
5	s = s+a[i];
6	return s;
7	}

Time Complexity: Example 2

	Statements
1	Algorithm Sum(a[],n,m)
2	{
3	for i=0 to n do
4	for j=0 to m do
5	s = s+a[i][j];
6	return s;
7	}

```
Algorithm log
 {
 for(i=1;i<n;i=i*2 )
   Printf(“*”)
}
```

i value varies from 1,2,4,8,16,32……..upto n

$2^0, 2^1, 2^2, 2^3, 2^4 \ldots \ldots \ldots \ldots 2^k$

n=2^k

k=log n

So For loop log n times

Using the count variable

```
1   Algorithm Sum(a, n)
2   {
3       s := 0.0;
4       count := count + 1; // count is global; it is initially zero.
5       for i := 1 to n do
6       {
7           count := count + 1; // For for
8           s := s + a[i]; count := count + 1; // For assignment
9       }
10      count := count + 1; // For last time of for
11      count := count + 1; // For the return
12      return s;
13  }
```

Using the count variable

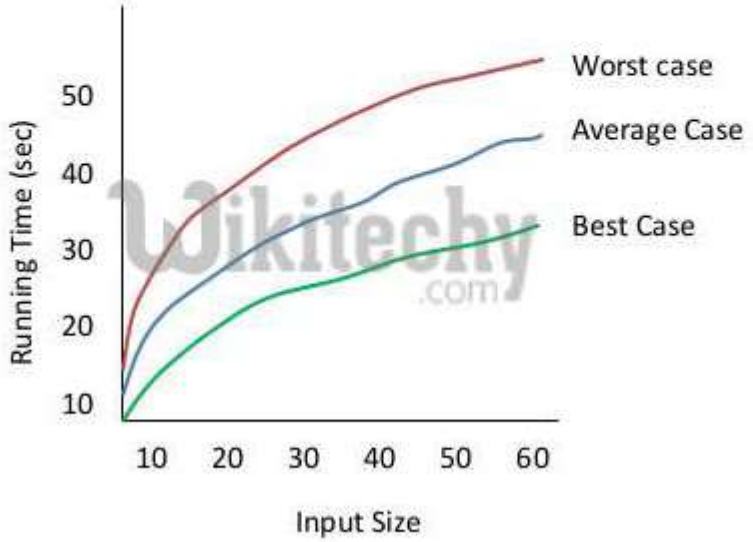
```
1   Algorithm RSum(a, n)
2   {
3       count := count + 1; // For the if conditional
4       if (n ≤ 0) then
5       {
6           count := count + 1; // For the return
7           return 0.0;
8       }
9       else
10      {
11          count := count + 1;  // For the addition, function
12                               // invocation and return
13          return RSum(a, n − 1) + a[n];
14      }
15  }
```

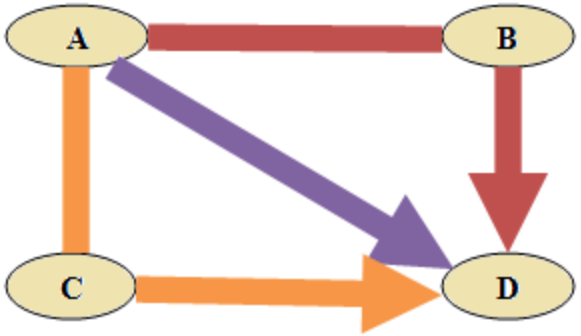
$$t_{\mathsf{RSum}}(n) = \begin{cases} 2 & \text{if } n = 0 \\ 2 + t_{\mathsf{RSum}}(n-1) & \text{if } n > 0 \end{cases}$$

Space and Time Complexity (Contd.)

- The time complexity of an algorithm is given by the number of steps taken by the algorithm to compute the function
- The number of specific steps is itself a function of the instance characteristics
- The best-case step count is the minimum number of steps that can be executed for the given parameters
- The worst-case step count is the maximum number of steps that can be executed for the given parameters
- The average step count is the average number of steps executed on instances with the given parameter

Best / Average/ Worst Case





Direction = A-B-D

Average Case

Direction = A-C-D

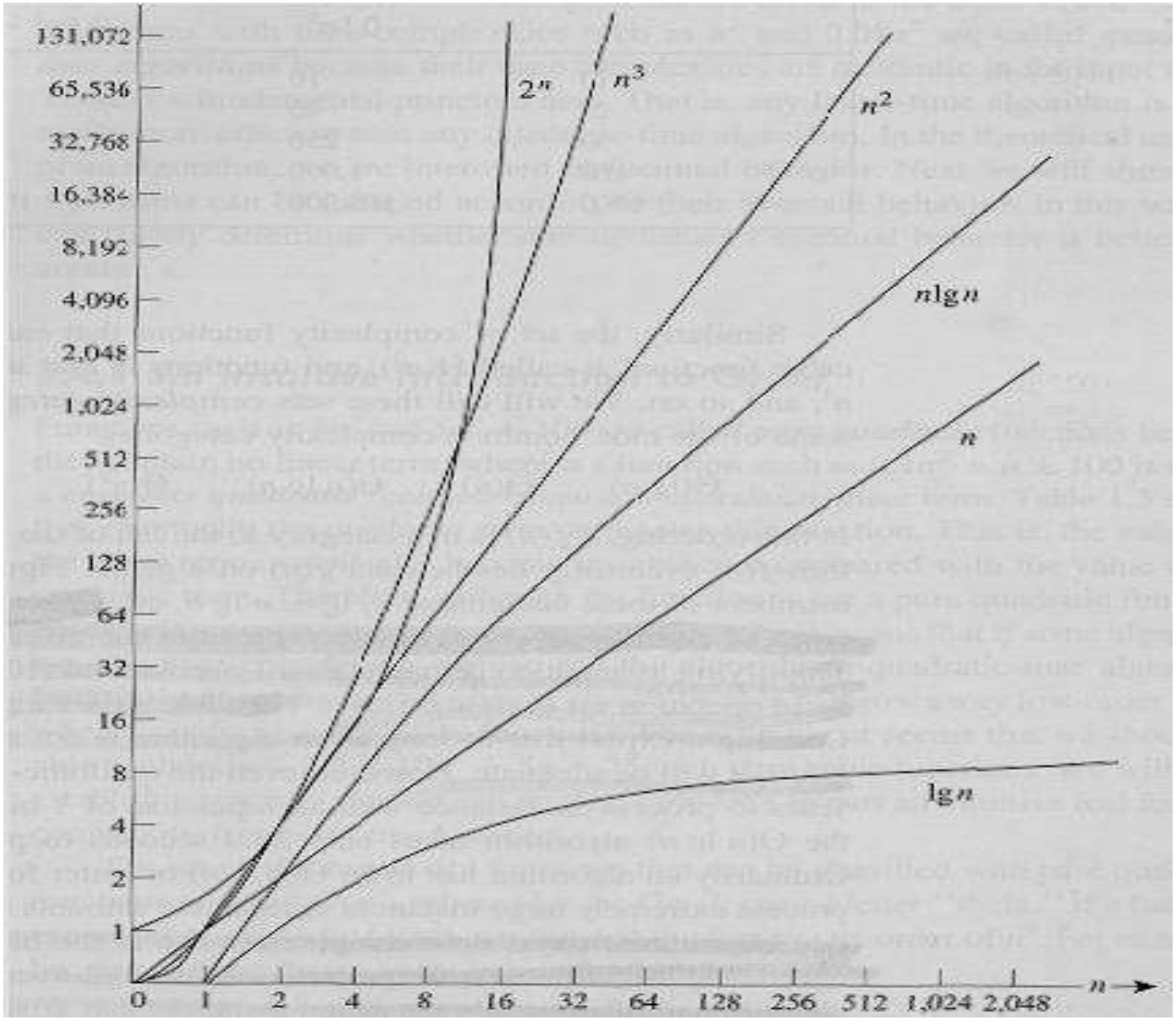
Worst Case

Direction = A-D

Best Case

Algorithm	Best Time Complexity	Average Time Complexity	Worst Time Complexity	Worst Space Complexity
Linear Search	O(1)	O(n)	O(n)	O(1)
Binary Search	O(1)	O(log n)	O(log n)	O(1)
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Merge Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
Quick Sort	O(nlogn)	O(nlogn)	O(n^2)	O(log n)
Heap Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
Bucket Sort	O(n+k)	O(n+k)	O(n^2)	O(n)
Radix Sort	O(nk)	O(nk)	O(nk)	O(n+k)
Tim Sort	O(n)	O(nlogn)	O(nlogn)	O(n)
Shell Sort	O(n)	O((nlog(n))^2)	O((nlog(n))^2)	O(1)

Common orders of magnitude



Common orders of magnitude

- We denote O(1) to mean a computing time that is a constant, O(n) is called linear, O(n2) is called quadratic, 0(n3) is called cubic, and O(2n) is called exponential If an algorithm takes time O(log n), it is faster, for sufficiently large n, than if it had taken O(n)

Tables showing how various functions grow with n

	constant	*logarithmic*	*linear*	*N-log-N*	*quadratic*	*cubic*	*exponential*
n	**O(1)**	**O(log *n*)**	**O(*n*)**	**O(*n* log *n*)**	**O(n^2)**	**O(n^3)**	**O(2^n)**
1	1	1	1	1	1	1	2
2	1	1	2	2	4	8	4
4	1	2	4	8	16	64	16
8	1	3	8	24	64	512	256
16	1	4	16	64	256	4,096	65536
32	1	5	32	160	1,024	32,768	4,294,967,296
64	1	6	64	384	4,069	262,144	1.84×10^{19}

$\log_b(x \cdot y) = \log_b(x) + \log_b(y)$

$\log_b(x / y) = \log_b(x) - \log_b(y)$

$\log_b(x^y) = y \cdot \log_b(x)$

$\log_b(c) = 1 / \log_c(b)$

$\log_b(x) = \log_c(x) / \log_c(b)$

$\log_b(x)$ is undefined when $x \leq 0$

$\log_b(0)$ is undefined

$y = \log_b(x)$, $x = b^y$

$\log_b(1) = 0$

$\log_b(b) = 1$

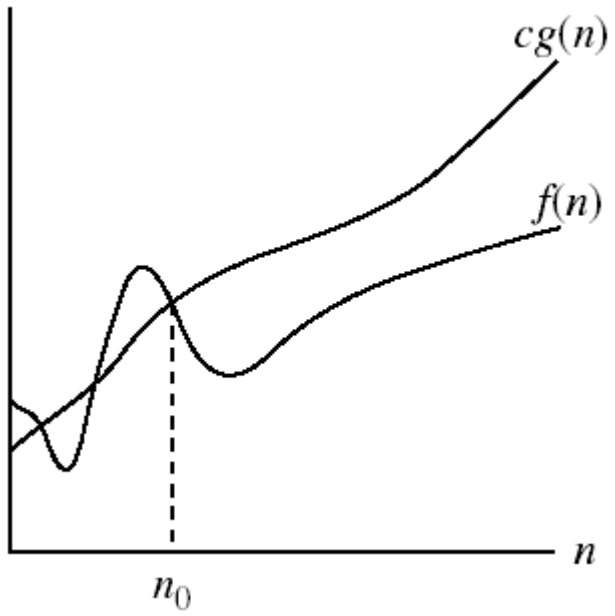
Asymptotic Analysis

- **Asymptotic notations** are the mathematical **notations** used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value

Asymptotic notations

- *O-notation*

$$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\} .$$



$g(n)$ is an ***asymptotic upper bound*** for $f(n)$.

- f(n) =100+n
- Let F(n)=3n+2, g(n)=n

$3n+2 \leq 4n$

	3n+2	4n
n=1	3*1+2=5	4*1=4
n=2	3*2+2=8	4*2=8
n=3	3*3+2=11	4*3=12
N=4	3*4+2=14	4*4=16

$F(n) \leq c*g(n)$ where c=4 and n0=2

$F(n)=O(4n)$

- F(n)=100n+6
- Let F(n)=100n, g(n)=6

$100n+6 \leq 101n$

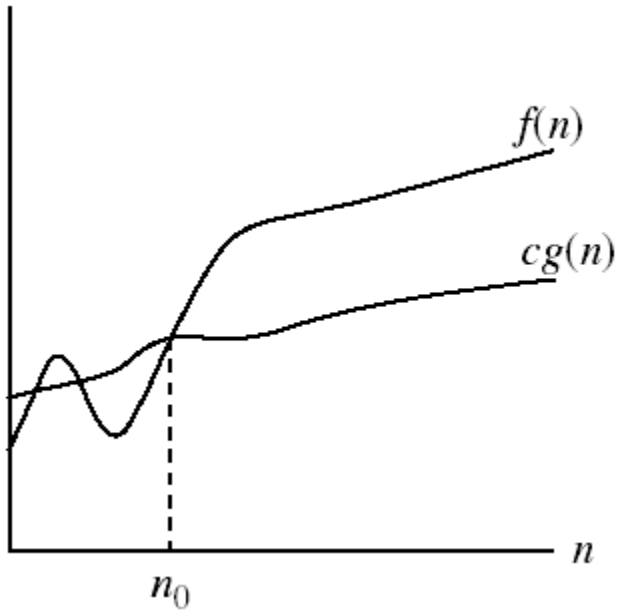
$F(n) \leq c*g(n)$ where $c=101$ and $n0=6$

$F(n)=O(n)$

Asymptotic notations

Ω - *notation*

$$\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0\} .$$



$g(n)$ is an ***asymptotic lower bound*** for $f(n)$.

- F(n)=3n+2
- Let F(n)=3n+2, g(n)=n

$3n+2 \geq 3n$

	3n+2	3n
n=1	3*1+2=5	3*1=3
n=2	3*2+2=8	3*2=6
n=3	3*3+2=11	3*3=9
N=4	3*4+2=14	3*4=12

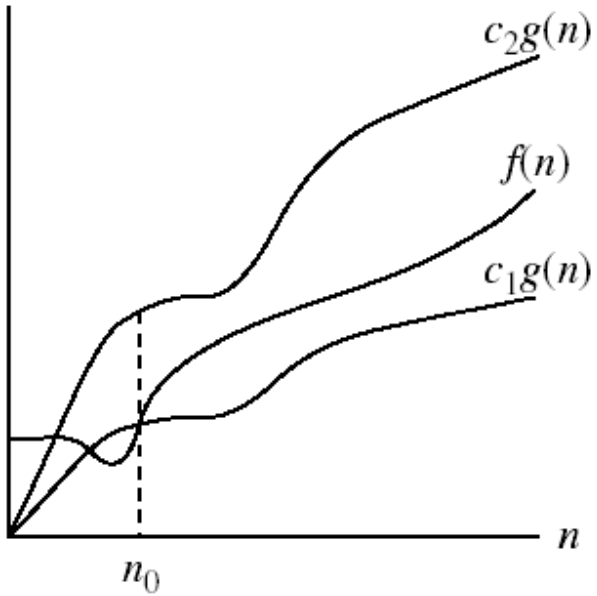
$F(n) \geq c*g(n)$ where c=3 and n0>=0

F(n)=

Asymptotic notations

Θ-*notation*

$$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\} .$$



$g(n)$ is an ***asymptotically tight bound*** for $f(n)$.

Asymptotic notations

- **little-Oh Defn**: $f(n) = o(g(n))$

if for all positive constants c there exists an n_0 such that $f(n) < c \cdot g(n)$ for all $n \geq n_0$

Asymptotic notations

- **little-Omega Defn**: $f(n) = \omega(g(n))$

if for all positive constants $c>0$ there exists an $n_{0>} 0$ such that $0 <= c \cdot g(n) < f(n)$ for all $n \geq n_0$

References

- Ellis Horowitz, Sartaj Sahni, Sanguthevar Rajasekaran, Fundamentals of Computer Algorithms, 2nd edition, Universities Press (India) Pvt. Ltd., 2008
- 2 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Cliord Stein, Introduction to Algorithms, 3rd edition, The MIT Press,

Cambridge, 2009