Advanced Analysis of Algorithms

Dr. Qaiser Abbas
Department of Computer Science & IT,
University of Sargodha, Sargodha, 40100, Pakistan
qaiser.abbas@uos.edu.pk

Background

- Algorithm:
  - A sequence of computational steps that transforms the input into the output.
- Analysis:
  - Performance characteristics of algorithms via time and space
  - Existing ways of problem solving (initial state, transition function, final state like in AI)
  - Evaluate its suitability for a particular problem (Using accuracy, precision, recall, etc. measures)

Properties of Algorithms

- Independent of Hardware, Operating Systems, Compilers and Languages.
- Bad Design e.g. badly chosen starting point, repeat algorithm several times, etc.

Complexity

- Goal is to classify algorithms according to their performance characteristics e.g. Time and Space
- This can be done in two ways:
  - Method 1: Absolute Value
    - Space required: Bytes?
    - Time required: Seconds?
  - Second Method:
    - Size of the problem represented by n
    - Independent of machine, operating system, compiler and language used

Space Complexity

- How much Space required to execute an algorithm?
  - A simple space example - adding two arrays of integers. Assume our array size is N and A,B are input arrays.
  - The following code requires N+N+N=3N space:

```
int C[N];
for (int i=0; i < N; i++)
C[i] = A[i] + B[i];
```

Space Complexity

- Suppose we no longer need B - we can reuse it. The following code requires only N+N=2N space:

```
for (int i=0; i < N; i++)
B[i] = A[i] + B[i];
```

- Space is no longer a problem
- Many algorithms compromise space requirements since memory is cheap

Time Complexity

- How much time does each function require?
- Actual time (i.e. seconds) hard to measure - varies from machine to machine and system to system
- Consider time as number of basic operations:
  - one arithmetic op, e.g. + - *
  - one assignment
  - one read
  - one write
  - etc.

Time Complexity

```
Begin                          //labels do not cost anything
    Initialize n               //1
    For i=0 to n               //n+2
        Print i                //n+1
    End For                    //label
End                            //label
--------------------------------------------------------------------------
                               2n+4
```

- Ignore machine dependent constants instead of the actual running time
- Look at the growth of the running time.
- To express time requirements we use "Big-0" notation.
- So, final T(n) = O(n)

Big “O” Notation

- **Definition**: function f(n) is O(g(n)) if there exist constants k and N such that for all n>=N: f(n) <= k * g(n).
  - The notation is often confusing: f = O(g) is read "f is big-oh of g.“
- Generally, when we see a statement of the form f(n)=O(g(n)):
  - f(n) is the formula that tells us exactly how many operations the function/algorithm in question will perform when the problem size is n.

Big “O” Notation

- g(n) is like an *upper bound* for f(n). Within a constant factor, the number of operations required by your function is *no worse* than g(n).

- Why is this useful?
  - We want out algorithms to *scalable*. Often, we write program and test them on relatively small inputs. Yet, we expect a user to run our program with larger inputs. Running-time analysis helps us *predict* how efficient our program will be in the `real world'.

Example 1

- If T(n) = 7n+100
- What is T(n) for different values of n???

n	T(n)	Comment
1	107	Contributing factor is 100
5	135	Contributing factor is 7n and 100
10	170	Contributing factor is 7n and 100
100	800	Contribution of 100 is small
1000	7100	Contributing factor is 7n
10000	70100	Contributing factor is 7n
10^6	7000100	What is the contributing factor????

- When approximating T(n) we can IGNORE the 100 term for very large value of n and say that T(n) can be approximated by 7(n)

Example 2

- $T(n) = n^2 + 100n + \log_{10}n + 1000$

n	T(n)	n^2		100n		$\log_{10}n$		1000	
		Val	%	Val	%	Val	%	Val	%
1	1101	1	0.1%	100	9.1%	0	0%	1000	90.8%
10	2101	100	5.8%	1000	47.6%	1	0.05%	1000	47.6%
100	21002	10000	47.6%	10000	47.6%	2	0.99%	1000	4.76%
10^5	10,010,001,005	10^{10}	99.9%	10^7	.099%	5	0.0%	1000	0.00%

- When approximating T(n) we can IGNORE the last 3 terms and say that T(n) can be approximated by n^2

Growth Rates

```
sum++;
```

This is O(1)

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

This is O(n)

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

This is $O(n^2)$

Comparison of Growth Rates

N	$\log_2 N$	$N \log_2 N$	N^2	N^3	2^N
1	0	0	1	1	2
2	1	2	4	8	4
8	3	24	64	512	256
64	6	384	4096	262,144	About 5 years
128	7	896	16,384	2,097,152	Approx 6 billion years, 600,000 times more than age of univ.

(If one operation takes 10^{-11} seconds)

Facts

- Gives us means for comparing algorithms.
- It tells us about the growth rate of an algorithm as input size becomes large.
- Its also called the asymptotic growth rate or asymptotic order or simply order of the function.

Break

Dynamic Programming

- An algorithmic paradigm that solves a given complex problem by breaking it into subproblems and stores the results of subproblems to avoid computing the same results again.
- Following are the two main properties of a problem that suggest that the given problem can be solved using Dynamic programming (DP).
  - Overlapping Subproblems
  - Optimal Substructure

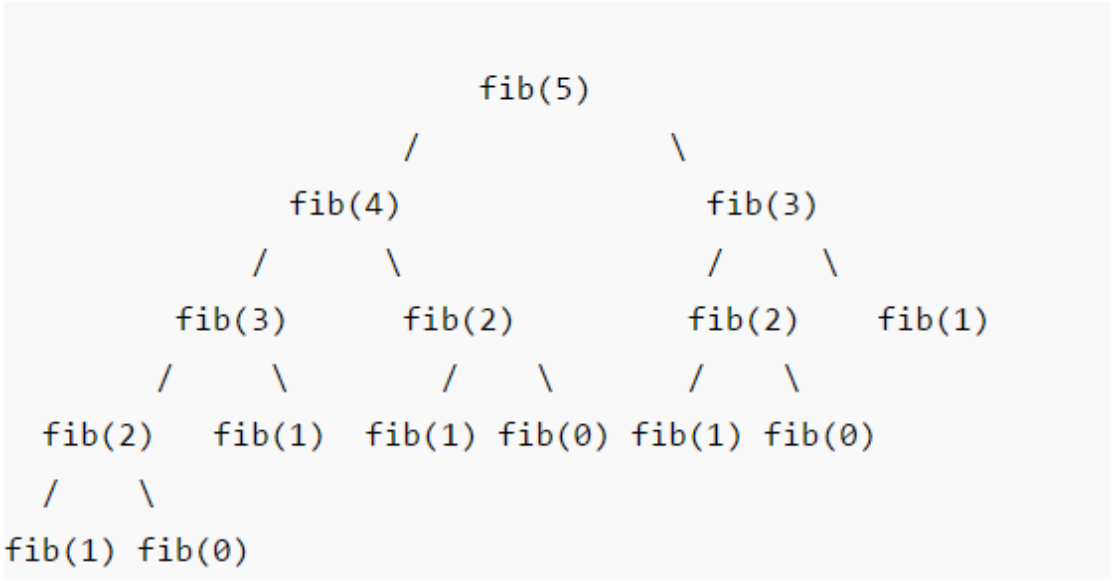
Overlapping Subproblems

- DP is mainly used when solutions of same subproblems are needed again and again.
- In DP, computed solutions to subproblems are stored in a table so that these don't have to recomputed.
- So DP is not useful when there are no common (overlapping) subproblems because there is no point of storing the solutions if they are not needed again.
- For example, If we take example of following recursive program for Fibonacci Numbers, there are many subproblems which are solved again and again.

Overlapping Subproblems

```c
/* simple recursive program for Fibonacci numbers */
int fib(int n)
{
   if ( n <= 1 )
      return n;
   return fib(n-1) + fib(n-2);
}
```

Recursion tree for execution of *fib(5)*



Overlapping Subproblems

- Function *f(3)* is being called 2 times. If we would have stored the value of *f(3)*, then instead of computing it again, we would have reused the old stored value.
- There are following two different ways to store the values so that these values can be reused.
  - *a) Memoization (Top Down):*
  - *b) Tabulation (Bottom Up):*

Overlapping Subproblems

- *a)* ***Memoization*** *(Top Down):* A recursive program that it looks into a lookup table before computing solutions.
- Initialize a lookup array with NIL values.
- Whenever we need solution to a subproblem, we first look into the lookup table. If the precomputed value is there then we return that value, otherwise we calculate the value and put the result in lookup table so that it can be reused later.

Following is the memoized version for nth Fibonacci Number.

```c
/* Memoized version for nth Fibonacci number */
#include<stdio.h>
#define NIL -1
#define MAX 100

int lookup[MAX];

/* Function to initialize NIL values in lookup table */
void _initialize()
{
  int i;
  for (i = 0; i < MAX; i++)
    lookup[i] = NIL;
}

/* function for nth Fibonacci number */
int fib(int n)
{
   if(lookup[n] == NIL)
   {
    if ( n <= 1 )
      lookup[n] = n;
    else
      lookup[n] = fib(n-1) + fib(n-2);
   }

   return lookup[n];
}

int main ()
{
  int n = 40;
  _initialize();
  printf("Fibonacci number is %d ", fib(n));
  getchar();
  return 0;
}
```

Overlapping Subproblems

- *b)* ***Tabulation*** *(Bottom Up):* The tabulated program for a given problem builds a table in bottom-up fashion and returns the last entry from table.

```
/* tabulated version */
#include<stdio.h>
int fib(int n)
{
  int f[n+1];
  int i;
  f[0] = 0;   f[1] = 1;
  for (i = 2; i <= n; i++)
      f[i] = f[i-1] + f[i-2];

  return f[n];
}

int main ()
{
  int n = 9;
  printf("Fibonacci number is %d ", fib(n));
  getchar();
  return 0;
}
```

Overlapping Subproblems

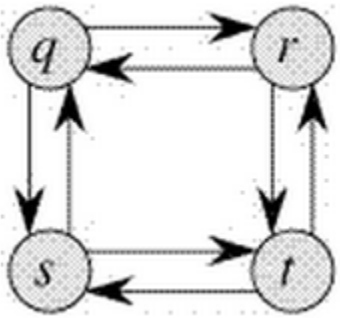
- Both (tabulated and Memoized) store the solutions of subproblems.
- In Memoized version, table is filled on demand while in tabulated version, starting from the first entry, all entries are filled one by one.
- Unlike the tabulated version, all entries of the lookup table are not necessarily filled in memoized version. For example, memoized solution of LCS problem doesn't necessarily fill all entries.

Optimal Substructure

- When optimal solution of the given problem can be obtained using optimal solutions of its subproblems then this is called Optimal Substructure Property.
- For example the shortest path problem has following optimal substructure property: If a node x lies in the shortest path from a source node u to destination node v then the shortest path from u to v is combination of shortest path from u to x and shortest path from x to v.
- The standard All Pair Shortest Path algorithms like Floyd–Warshall and Bellman–Ford are typical examples of Dynamic Programming.

Optimal Substructure

- On the other hand the Longest Path Problem doesn't have the Optimal Substructure property.
- Here by Longest Path we mean longest simple path (path without cycle) between two nodes.
- Consider the following unweighted graph. There are two longest paths from q to t: q -> r ->t and q ->s->t.
- Unlike shortest paths, these longest paths do not have the optimal substructure property.
- For example, the longest path q->r->t is not a combination of longest path from q to r and longest path from r to t, because the longest path from q to r is q->s->t->r.



Matrix Chain Multiplication

- **Problem**: Given a sequence of matrices, find the most efficient way to multiply these matrices together.
- The problem is not actually to perform the multiplications, but merely to decide in which order to perform the multiplications.

Matrix Chain Multiplication

- As matrix multiplication is associative, so no matter how we parenthesize the product, the result will be the same. For example, if we had four matrices A, B, C, and D, we would have:

  (ABC)D = (AB)(CD) = A(BCD) = ...

- However, the order in which we parenthesize the product affects the number of simple arithmetic operations needed to compute the product, or the efficiency.
- For example, suppose A is a 10 × 30 matrix, B is a 30 × 5 matrix, and C is a 5 × 60 matrix. Then,

  (AB)C = (10×30×5) + (10×5×60) = 1500 + 3000 = 4500 operations

  A(BC) = (30×5×60) + (10×30×60) = 9000 + 18000 = 27000 operations.

- Clearly the first parenthesization requires a smaller number of operations.

Matrix Chain Multiplication

- Given an array p[] which represents the chain of matrices such that the i^{th} matrix A_i is of dimension p[i-1] x p[i].
- We need to write a function MatrixChainOrder() that should return the minimum number of multiplications needed to multiply the chain.

  Input: p[] = {40, 20, 30, 10, 30}

  Output: 26000
- There are 4 matrices of dimensions 40x20, 20x30, 30x10 and 10x30.
- Let the input 4 matrices be A, B, C and D.
- The minimum number of multiplications are obtained by putting parenthesis in following way

  (A(BC))D --> 20*30*10 + 40*20*10 + 40*10*30

Matrix Chain Multiplication

- Input: p[] = {10, 20, 30, 40, 30} Output: 30000
  - There are 4 matrices of dimensions 10x20, 20x30, 30x40 and 40x30.
  - Let the input 4 matrices be A, B, C and D.
  - The minimum number of multiplications are obtained by putting parenthesis in following way

    ((AB)C)D --> 10*20*30 + 10*30*40 + 10*40*30
- Input: p[] = {10, 20, 30} Output: 6000
  - There are only two matrices of dimensions 10x20 and 20x30. So there is only one way to multiply the matrices, cost of which is 10*20*30

Matrix Chain Multiplication

- **Optimal Substructure:**
  - In a chain of matrices of size n, we can place the first set of parenthesis in n-1 ways.
  - For chain ABCD, then there are 3 ways to place first set of parenthesis e.g. A(BCD), (AB)CD and (ABC)D.
  - Placing a set of parenthesis is division of problem into subproblems of smaller size known as optimal substructure property and can be easily solved using recursion.
  - Minimum number of multiplication for a chain of size n = Minimum among all n-1 placements

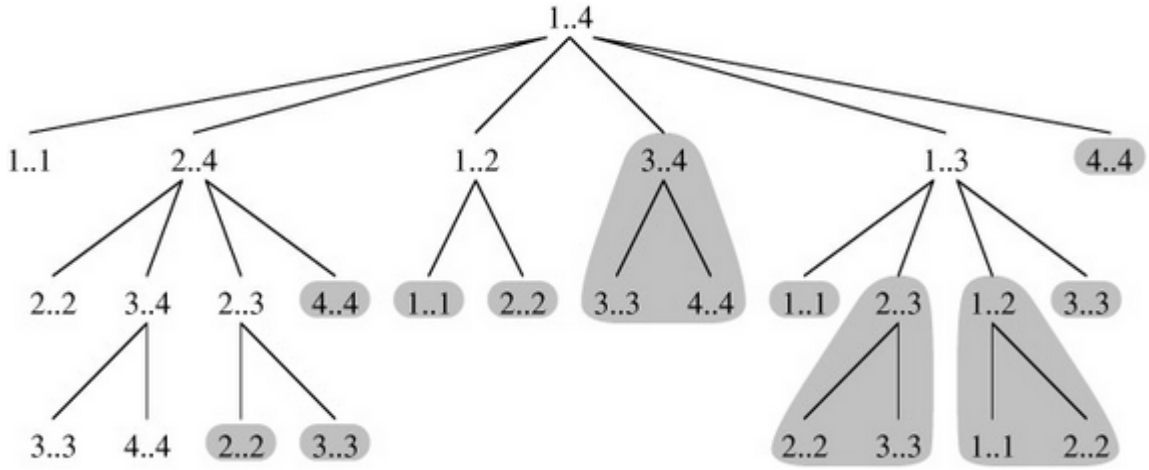
- **Overlapping Subproblems**: Following is a recursive implementation that simply follows the above optimal substructure property.

```
// Matrix Ai has dimension p[i-1] x p[i] for i = 1..n
int MatrixChainOrder(int p[], int i, int j)
{
    if(i == j)
        return 0;
    int k;
    int min = INT_MAX;
    int count;

    // place parenthesis at different places between first and last matrix,
    // recursively calculate count of multiplcations for each parenthesis
    // placement and return the minimum count
    for (k = i; k <j; k++)
    {
        count = MatrixChainOrder(p, i, k) +
                MatrixChainOrder(p, k+1, j) +
                p[i-1]*p[k]*p[j];

        if (count < min)
            min = count;
    }

    // Return minimum count
    return min;
}

// Driver program to test above function
int main()
{
    int arr[] = {1, 2, 3, 4, 3};
    int n = sizeof(arr)/sizeof(arr[0]);

    printf("Minimum number of multiplications is %d ",
                           MatrixChainOrder(arr, 1, n-1));

    getchar();
    return 0;
}
```

Matrix Chain Multiplication

- Time complexity of the naive recursive approach is exponential.
- Following is the recursion tree for a matrix chain of size 4.
- It should be noted that the function computes the same subproblems again and again.

Matrix Chain Multiplication



- Calling of same subproblems again is **Overlapping Subproblems property**.
- Matrix Chain Multiplication (MCM) problem has both properties of a dynamic programming (DP).
- Like other typical DP problems, re-computations of same subproblems can be avoided by constructing a temporary array **m[][]** in bottom-up manner.

```
// Matrix Ai has dimension p[i-1] x p[i] for i = 1..n
int MatrixChainOrder(int p[], int n)
{

    /* For simplicity of the program, one extra row and one extra column are
       allocated in m[][].  0th row and 0th column of m[][] are not used */
    int m[n][n];

    int i, j, k, L, q;

    /* m[i,j] = Minimum number of scalar multiplications needed to compute
       the matrix A[i]A[i+1]...A[j] = A[i..j] where dimention of A[i] is
       p[i-1] x p[i] */

    // cost is zero when multiplying one matrix.
    for (i = 1; i < n; i++)
        m[i][i] = 0;

    // L is chain length.
    for (L=2; L<n; L++)
    {
        for (i=1; i<=n-L+1; i++)
        {
            j = i+L-1;
            m[i][j] = INT_MAX;
            for (k=i; k<=j-1; k++)
            {
                // q = cost/scalar multiplications
                q = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j];
                if (q < m[i][j])
                    m[i][j] = q;
            }
        }
    }

    return m[1][n-1];
}

int main()
{
    int arr[] = {1, 2, 3, 4};
    int size = sizeof(arr)/sizeof(arr[0]);

    printf("Minimum number of multiplications is %d ",
                        MatrixChainOrder(arr, size));

    getchar();
    return 0;
}
```

- **Dynamic Programming Solution:** Following is C/C++ implementation for MCM problem using DP.
- Time Complexity: O(n^3)

Cormen's Slides

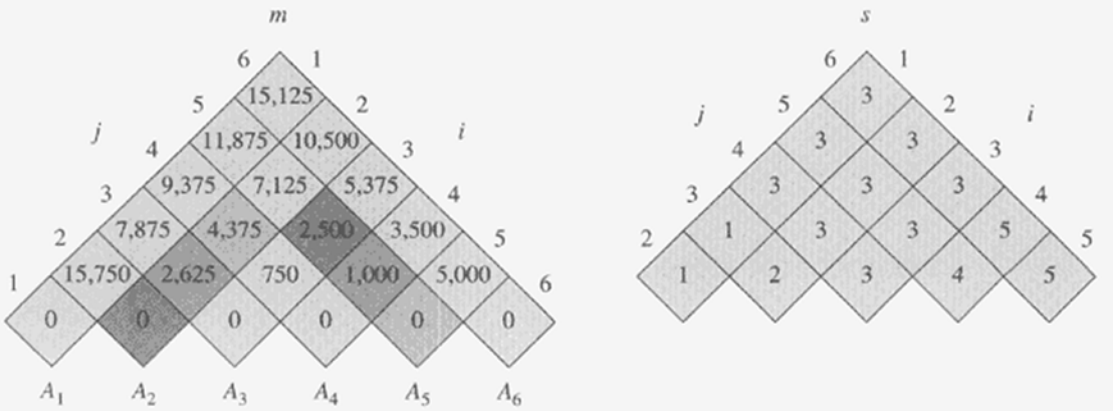


Figure 15.3 The m and s tables computed by MATRIX-CHAIN-ORDER for $n = 6$ and the following matrix dimensions:

matrix	dimension
A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25

The tables are rotated so that the main diagonal runs horizontally. Only the main diagonal and upper triangle are used in the m table, and only the upper triangle is used in the s table. The minimum number of scalar multiplications to multiply the 6 matrices is $m[1, 6] = 15{,}125$. Of the darker entries, the pairs that have the same shading are taken together in line 9 when computing

$$m[2, 5] = \min \begin{cases} m[2, 2] + m[3, 5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13000\,, \\ m[2, 3] + m[4, 5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125\,, \\ m[2, 4] + m[5, 5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11375 \end{cases}$$
$$= 7125\,.$$

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

Read it Yourself

```
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 ")"
```

In the example of Figure 15.5, the call PRINT-OPTIMAL-PARENS$(s, 1, 6)$ prints the parenthesization $((A_1(A_2A_3))((A_4A_5)A_6))$.

Assignment # 1

15.2-1

Find an optimal parenthesization of a matrix-chain product whose sequence of dimensions is $\langle 5, 10, 3, 12, 5, 50, 6 \rangle$.

15.2-2

Give a recursive algorithm MATRIX-CHAIN-MULTIPLY(A, s, i, j) that actually performs the optimal matrix-chain multiplication, given the sequence of matrices $\langle A_1, A_2, \ldots, A_n \rangle$, the s table computed by MATRIX-CHAIN-ORDER, and the indices i and j. (The initial call would be MATRIX-CHAIN-MULTIPLY$(A, s, 1, n)$.)

15.2-3

Use the substitution method to show that the solution to the recurrence (15.6) is $\Omega(2^n)$.

Reading

- Abo-Sinna, Mahmoud. (2004). Multiple objective (fuzzy) dynamic programming problems: A survey and some applications. Applied Mathematics and Computation. 157. 861-888.
- Has Dynamic Programming Improved Decision Making?, John Rust, Annual Review of Economics 2019 11:1, 833-858