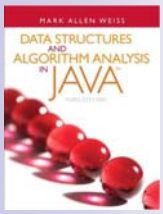


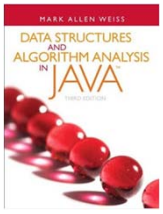
FIU FLORIDA INTERNATIONAL UNIVERSITY

Algorithm Analysis (III)

Dr. Antonio L. Bajuelos

FIU School of Computing & Information Sciences

Note: The most of the information of these slides was extracted and adapted from Weiss's book, "*Data Structures and Algorithm Analysis in Java*". They are provided for COP3530 students only. Not to be published or publicly distributed without permission by the publisher.



COP-3530 - Data Structures

Module #1: Algorithm Analysis (part III)

Outline:

- **Calculating the Running Time for a Program.**
  - **Counting method**
  - **General programming rules**
  - **Algorithm Analysis for recursive methods**
- **The Maximum Subsequence Sum Problem**

Calculating the Running Time for a Program

- Java code fragment to calculate $\sum_{i=1}^{N} i^3$

```
                public static int sum( int n )
                {
                  int partialSum;
1                 partialSum = 0;
2                 for ( int i = 1; i <= n; i++ )
3                   partialSum += i * i * i;
4                 return partialSum;
                }
```

- We assume that the declarations count for no time.
- Lines 1 and 4 count for one unit each.
- Line 3 counts for four units per time executed (two *, one +, and one assignment) and is executed N times, for a total of $4N$ units.
- Line 2 has the costs of initializing i, testing $i \leq N$, and incrementing i. The total cost of all these is 1 to initialize, $N+1$ for all the tests, and N for all the increments, which is $2N + 2$.
- We ignore the costs of calling the method and returning, for a total of $6N + 4$.
- Thus, we say that **this method is $O(N)$**

3

Calculating the Running Time for a Program. General Rules

- **Rule 1— For loops.**
  - The running time of a for loop is at most the running time of the statements inside the for loop (including tests) times the number of iterations.
- **Rule 2—Nested loops.**
  - The total running time of a statement inside a group of nested loops is the running time of the statement multiplied by the product of the sizes of all the loops.
  - Example: The following program fragment is $O(N^2)$:

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

4

Calculating the Running Time for a Program. General Rules

- **Rule 3—Consecutive Statements.**
  - These just add. This means that the maximum is the one that counts by the rule:

    $T_1(N)+T_2(N) = O(f(N)+g(N))$ (or O(max($f(N)$, $g(N)$))),
  - Example: The following program fragment, which has $O(N)$ work followed by $O(N^2)$ work, is also $O(N^2)$:

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

5

Calculating the Running Time for a Program. General Rules

- **Rule 4 - if/else.**
  - For the fragment:

```
if( condition )
    S1
else
    S2
```

  - the running time never more than the running time of the test/condition plus the larger of the running times of S1 and S2.

6

Algorithm Analysis: Example of Nested Loops

- Some strategies to calculate Sum = 1+2+…+N

Algorithm A

```
public static int Sum1( int N )
{
  int Sum = 0;
  for( int i = 0;  i < N; i++ )
     for( int j = i; j < N; j++ )
        Sum += 1;
  return Sum;
}
```

Algorithm B

```
public static int Sum2( int N )
{
   int Sum = 0;
   for( int i = 0;  i < N; i++ )
        Sum += i;
   return Sum;
}
```

Algorithm C

```
public static int Sum3( int N )
{
   return (N*(N+1))/2;
}
```

	Algorithm A	Algorithm B	Algorithm C
Complexity:			

7

Algorithm Analysis: Other Rules

- The basic strategy **is to analyze from the inside** (or deepest part) out works.
- If there are method calls, these must be analyzed first.
- If there are **recursive methods**, there are several options:
  - If the recursion is really immediate **for loop**, the analysis is trivial.
  - **Example:** The following method is really just a simple loop and is $O(N)$:

```
public static long factorial( int n )
{
     if( n <= 0 )
       return 1;
     else
       return n * factorial( n - 1 );
}
```

  - **Exercise:** What's T(N) for the **recursive** factorial version?

8

Algorithm Analysis: Recursive Methods

- **When recursion is properly used, it is difficult to convert the recursion into a simple loop structure.**
  - The algorithm analysis will involve a recurrence relation that needs to be solved.
  - Example (**Fibonacci Sequence**):

```
    public static long fib( int n )
    {
1       if( n <= 2 )
2           return 1;
        else
3           return fib( n - 1 ) + fib( n - 2 );
    }
```

- The total time required to calculate **fib(n)** (for N ≥ 2) is:

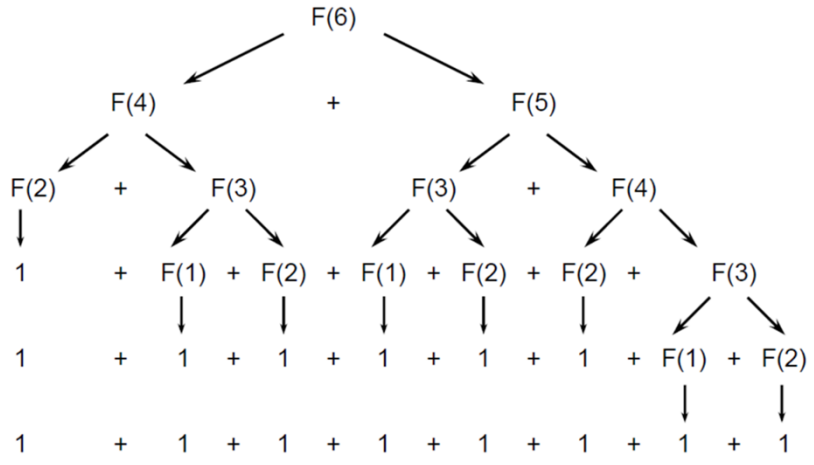
  T(N) = T(N − 1) + T(N − 2) + *C*,

  where the *C* (constant) accounts for the work at line 1 plus the addition at line 3.

9

Algorithm Analysis: Fibonacci Sequence

Simulation: F(6) = 8



10

Algorithm Analysis: Recursive Fibonacci

- The total time, required to calculate **fib(n)** (for N ≥ 2) is:

  T(N) = T(N − 1) + T(N − 2) + 2,

  where the 2 accounts for the work at line 1 plus the addition at line 3.
- We determine a **lower bound** on T(N):

$$
\begin{aligned}
T(N) &= T(N-1) + T(N-2) + 2 \\
&\geq T(N-2) + T(N-2) + 2 \\
&= 2T(N-2) + 2 = 2(T(N-3) + T(N-4) + 2) + 2 \\
&\geq 2(T(N-4) + T(N-4) + 2) + 2 \\
&= 2^2T(N-4) + 2^2 + 2 \\
&= 2^2(T(N-5) + T(N-6) + 2) + 2^2 + 2 \\
&\geq 2^3T(N-6) + 2(2^2 + 2^1 + 2^0) \\
&\ldots \\
&= 2^kT(N-2k) + 2(2^k - 1)
\end{aligned}
$$

The base case is reached when N - 2k = 2

Hence $\mathbf{T(N) \geq 2^{(N-2)/2}T(2) + 2(2^{(N-2)/2} - 1)}$

and then **Recursive Fibonacci is exponential** (**$O(2^N)$**)

11

Fibonacci Sequence. Iterative version

```
public static long fib( int n)
{
        int prev1 = 1;
        int prev2 = 1;
        for(int i = 2; i < n; i++)
        {
              int savePrev1 = prev1;
              prev1 = prev2;
              prev2 = savePrev1 + prev2;
        }
        return prev1;
}
```

- It's not elegant solution but more **efficient solution**:
- **$T(n) = c_1n + c_2$ is $O(n)$.**

12

Algorithm Analysis: An example

- **Maximum Subsequence Sum Problem (MSSP).**

  Given (possibly negative) integers $A_1, A_2, \ldots, A_N$, find the maximum value of $\sum_{k=i}^{j} A_k$

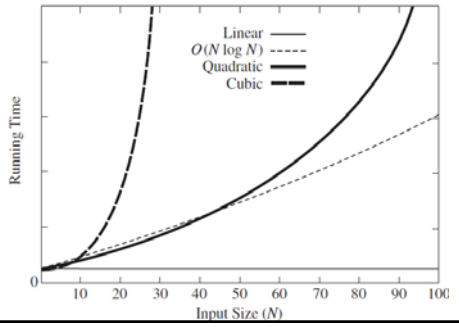
  Note: the maximum subsequence sum is 0 if all the integers are negative.)

- Example:
  - For input −2, 11, −4, 13, −5, −2, the answer is 20 ($A_2 + A_3 + A_4$).

Algorithm Analysis: An example

- We will discuss **three algorithms** to solve this problem.
- Example of the running time on "some" computer:

Input Size	Algorithm Time 1 $O(N^3)$	2 $O(N^2)$	3 $O(N \log N)$	4 $O(N)$
$N = 100$	0.000159	0.000006	0.000005	0.000002
$N = 1{,}000$	0.095857	0.000371	0.000060	0.000022
$N = 10{,}000$	86.67	0.033322	0.000619	0.000222
$N = 100{,}000$	NA	3.33	0.006700	0.002205
$N = 1{,}000{,}000$	NA	NA	0.074870	0.022711



Maximum Subsequence Sum Problem in O(N^3)

```
1   public static int maxSubSum1( int [ ] a )
2   {
3     int maxSum = 0;
4     for( int i = 0; i < a.length; i++ )
5       for( int j = i; j < a.length; j++ )
6         {
7           int thisSum = 0;
8           for( int k = i; k <= j; k++ )
9                 thisSum += a[ k ];
10          if( thisSum > maxSum )
11                maxSum = thisSum;
12        }
13    return maxSum;
14  }
```

- **Analysis:**
  - The loop at line 4 is of size N.
  - The loop at line 5 has size $N - i$ (in the worst-case may be of size N).
  - The loop a line 8 has size $j-i+1$, which, again, we must assume is of size N.
  - Loops are $O(C \cdot N \cdot N \cdot N) = O(N^3)$.
  - Line 3 takes only $O(1)$, and lines 10 and 11 are $O(N^2)$ total, since they are easy expressions inside only two loops.
  - Total: $\boldsymbol{O(N^3)}$

15

Maximum Subsequence Sum Problem in O(N^2)

```
1   public static int maxSubSum2( int [ ] a )
2   {
3     int maxSum = 0;
4     for( int i = 0; i < a.length; i++ )
5     {
6       int thisSum = 0;
7       for( int j = i; j < a.length; j++ )
8       {
9         thisSum += a[ j ];
10        if( thisSum > maxSum )
11          maxSum = thisSum;
12      }
13    }
14    return maxSum;
15  }
```

- **Comments & Analysis:**
  - Avoid the O(N^3) running time of **maxSubSum1** by removing a **for loop**.
  - Note that $\sum_{k=i}^{j} A_k = A_j + \sum_{k=i}^{j-1} A_k$, so the computation at lines 8 and 9 in algorithm **maxSubSum1** is improperly expensive.
  - **maxSubSum2** is $\boldsymbol{O(N^2)}$

16

Maximum Subsequence Sum Problem in O(*N*)

```
1     public static int maxSubSum3( int [ ] a )
2     {
3         int maxSum = 0, thisSum = 0;
4         for( int j = 0; j < a.length; j++ )
5         {
6             thisSum += a[ j ];
7             if( thisSum > maxSum )
8                 maxSum = thisSum;
9             else if( thisSum < 0 )
10                thisSum = 0;
11        }
12        return maxSum;
13    }
```

- **Comments:**
  - Note that if a[i] is negative, then it cannot possibly represent the start of the optimal sequence, since any subsequence that begins by including a[i] would be improved by beginning.
  - Similarly, any negative subsequence cannot possibly be a prefix of the optimal subsequence.
  - If we detect that the subsequence from a[i] to a[j] is negative, then we can advance i, so we can advance it all the way to j+1.
  - **maxSubSum3** is **O(*N*)**
- Examples: **(3,-8,2,-1,9,-11,4,4)** and **(3,-8,-2,-1,9,-11,4,4)**

17

Maximum Subsequence Sum Problem in O(N)

- The **maxSubSum3** is typical of many smart algorithms:
  - The running time is obvious, but the correctness is not.
  - For these algorithms, **formal correctness proofs** are always required.
  - Many of these algorithms require complex programming, but run quickly.
  - Extra advantage - it makes only one pass through the data, and once a[i] is read and processed, it does not need to be remembered – **online algorithm**.

18

Complementary recommended material

Self study:

- **Module 1 - Algorithm Analysis - Example Notes I.PDF**
- **Module 1 - Algorithm Analysis - Example Notes II.PDF**

19

20