DATA STRUCTURES AND ALGORITHMS

Lecture Notes 2

Prepared by İnanç TAHRALI

Recapture

- Asymptotic Notations
  - O Notation
  - Ω Notation
  - Θ Notation
  - o Notation

Big Oh Notation (O)

Provides an “upper bound” for the function *f*

Definition :

- T(N) = O (f(N)) if there are positive constants *c* and n_0 such that

  $T(N) \le c\, f(N)$ when $N \ge n_0$
  - T(N) grows no faster than f(N)
  - growth rate of T(N) is less than or equal to growth rate of f(N) for large N
  - f(N) is an upper bound on T(N)
    - not fully correct !

Omega Notation (Ω)

- **Definition :**

  $T(N) = \Omega(f(N))$ if there are positive constants c and n_0 such that $T(N) \geq c\,f(N)$ when $N \geq n_0$

  - T(N) grows no slower than f(N)
  - growth rate of T(N) is greater than or equal to growth rate of f(N) for large N
  - f(N) is a lower bound on T(N)
    - not fully correct !

Theta Notation (θ)

- **Definition :**

  $T(N) = \theta\ (h(N))$ if and only if

  $T(N) = O(h(N))$ and $T(N) = \Omega(h(N))$

  - $T(N)$ grows as fast as $h(N)$
  - growth rate of $T(N)$ and $h(N)$ are equal for large N
  - $h(N)$ is a tight bound on $T(N)$
    - not fully correct !

Little o Notation (o)

- **Definition :**

  T(N) = o(p(N)) if

  T(N) = O(p(N)) and T(N)≠θ(p(N))

  - p(N) grows strictly faster than T(N)
  - growth rate of T(N) is less than the growth rate of p(N) for large N
  - p(N) is an upperbound on T(N) (but not tight)
    - not fully correct !

ROAD MAP

- **Model**
- **What to Analyze ?**
- **Running Time Analysis**
- General Rules
- Recursive Calls
- Maximum Subsequence Sum Problem
- Binary Search
- Experimentally checking analysis

MODEL

- A formal framework for analysis (simplify the real computers)
- There are many models
  - Automata
  - Turing Machine
  - RAM

MODEL

- We use RAM model (normal computer)
  - addition
  - multiplication
  - comparison
  - assignment

    take a unit time
  - fixed size word (32 bit)
  - no complicated operation supported
    - requires an algorithm (algorithm may not take unit time)

What to Analyze ?

- Running Time (most important !)
- Required memory space

Run-time complexity is effected by

  - compiler
  - computer

usually effects the constants & lower order terms

  - algorithm

use asymptotic notation

Running Time Analysis

- Emprical → after implementation
- Theoritical → before implementation

- If there are many algorithms ideas, we need to evaluate them without implementation

- We will use 0-notation
  - drop constants
  - ignore lower order terms

Running Time Analysis

- **Example:**

```
int sum (int N)
{
    int i, partialsum;              → does not count
    partialsum = 0;                 → 1
    for (i=1; i<=N; i++)            → 1+(N+1)+N
          partialsum+=i*i*i;       →  3N
    return partialsum;              → 1
}
                                   =  5N + 4
                                   =  θ(N)
```

ROAD MAP

- Model
- What to Analyze ?
- Running Time Analysis
- **General Rules**
- Recursive Calls
- Maximum Subsequence Sum Problem
- Binary Search
- Experimentally checking analysis

GENERAL RULES

- **RULE 1 : For Loops**

  The running time of a for loop is at most the running time of the statements in the for loop times the number of iterations

```
int i, a = 0;
for (i=0; i<n; i++)
{
  print i;
  a=a+i;
}
return i;
```

$T(n) = \theta(n)$

GENERAL RULES

- **RULE 2 : Nested Loops**

  Analyze nested loops inside out

Example :

```
for (int i=1;i<=q;i++)
{
  for (int j=1;j<=r;j++)
      k++;
}
```

$\theta(r)$

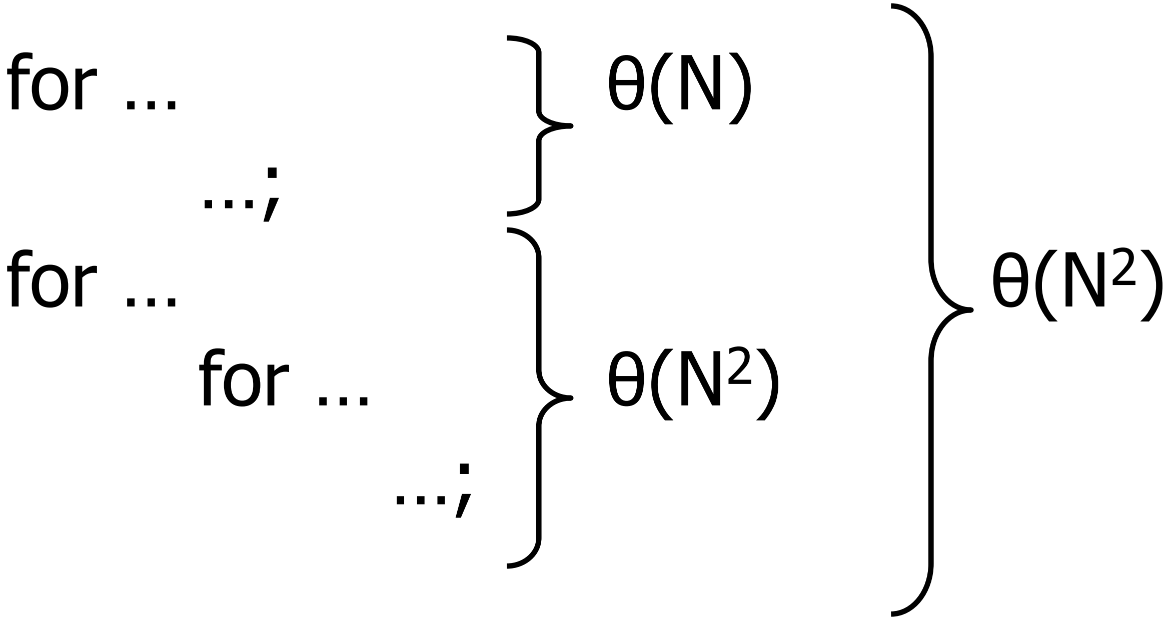
$\theta(q)$

$T(n) = \theta(r*q)$

GENERAL RULES

- **RULE 3 : Consequtive Statements**

  Add the running times



```
for …
          …;
for …
          for …
                    …;
```

GENERAL RULES

- **RULE 4 : If / Else**

$$\left.\begin{array}{lll} \text{if (condition)} & \} & T_3(n) \\ \quad\text{S1;} & \} & T_1(n) \\ \text{else} & & \\ \quad\text{S2;} & \} & T_2(n) \end{array}\right\} T(n)$$

Running time is never more than the running time of the test plus larger of the running times of S1 and S2

(may overestimate but never underestimates)

$T(n) \leq T_3(n) + \max\,(T_1(n), T_2(n))$

Types of complexition

$$T_{worst}(N) = \max_{|I|=N}\{T(I)\} \quad \rightarrow usually \quad used$$

$$T_{av}(N) = \sum_{|I|=N} T(I).\Pr(I) \quad \rightarrow difficult \;\; to \;\; compute$$

$$T_{best}(N) = \min_{|I|=N}\{T(I)\}$$

$$T_{worst}(N) \geq T_{av}(N) \geq T_{best}(N)$$

$$T(n) = O(T_{worst}(n)) = \Omega(T_{best}(n))$$

GENERAL RULES

- **RULE 4 : If / Else**

$$\left.\begin{array}{ll}
\text{if (condition)} & \} \; T_3(n) \\
\quad \text{S1;} & \} \; T_1(n) \\
\text{else} & \\
\quad \text{S2;} & \} \; T_2(n)
\end{array}\right\} T(n)$$

$$T_w\ (n) = T_3(n) + \max\ (T_1(n), T_2(n))$$

$$T_b\ (n) = T_3(n) + \min\ (T_1(n), T_2(n))$$

$$T_{av}\ (n) = p(T)T_1(n) + p(F)T_2(n) + T_3(n)$$

p(T) → p (condition = True)
p(F) → p (condition = False)

GENERAL RULES

- **Example :**

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

$$\left.\begin{array}{l} \left.\begin{array}{l} \end{array}\right\} T_3(n) = \theta(n) \\ \left.\begin{array}{l} \end{array}\right\} T_1(n) = \theta(n^2) \\ \\ \left.\begin{array}{l} \end{array}\right\} T_2(n) = \theta(n) \end{array}\right\} T(n)$$

$$T_w(n) = T_3(n) + \max(T_1(n), T_2(n)) = \theta(n^2)$$
$$T_b(n) = T_3(n) + \min(T_1(n), T_2(n)) = \theta(n)$$

if $p(T) = p(F) = ½$

$$T_{av}(n) = p(T)T_1(n) + p(F)T_2(n) + T_3(n) = \theta(n^2)$$

$$T(n) = O(n^2) = \Omega(n)$$

ROAD MAP

- Model
- What to Analyze ?
- Running Time Analysis
- General Rules
- **Recursive Calls**
- Maximum Subsequence Sum Problem
- Binary Search
- Experimentally checking analysis

RECURSIVE CALLS

Example 1:

Algorithm for computing factorial

```
int factorial (int n)
{
  if (n<=1)
      return 1;
  else
      return n*factorial(n  1);
}
```

1 for multiplication
+ 1 for substraction
+ cost of evaluation of
factorial(n-1)

T(n) = cost of evaluation of factorial of n

T(n) = 4 + T(n-1)

T(1) = 2

RECURSIVE CALLS

$T(n) = 4 + T(n-1)$

$T(n) = 4 + 4 + T(n-2)$

$T(n) = 4 + 4 + 4 + T(n-3)$

$\vdots$

$T(n) = k*4 + T(n-k)$ k= n-1 =>

$T(n) = (n-1)*4 + T(n-(n-1))$

$T(n) = (n-1)*4 + T(1)$

$T(n) = (n-1)*4 + 2$

$T(n) = \theta(n)$

RECURSIVE CALLS

Example 2:

Algorithm for fibonacci series

```
Fib (int N)
{
  if (N<=1)
      return 1;
  else
      return Fib(N-1)+Fib(N-2);
}
```

$T(N) = T(N-1) + T(N-2) + 4$, $T(1)=T(0)=2$

$\Rightarrow T(N) = O(2^n)$ (by induction)

ROAD MAP

- Model
- What to Analyze ?
- Running Time Analysis
- General Rules
- Recursive Calls
- **Maximum Subsequence Sum Problem**
- Binary Search
- Experimentally checking analysis

MAXIMUM SUBSEQUENCE SUM PROBLEM

- Given (possibly negative) integers $A_1, A_2, \ldots, A_N$, find

$$\max_{1 \le i \le j \le N} \sum_{k=i}^{j} A_k$$

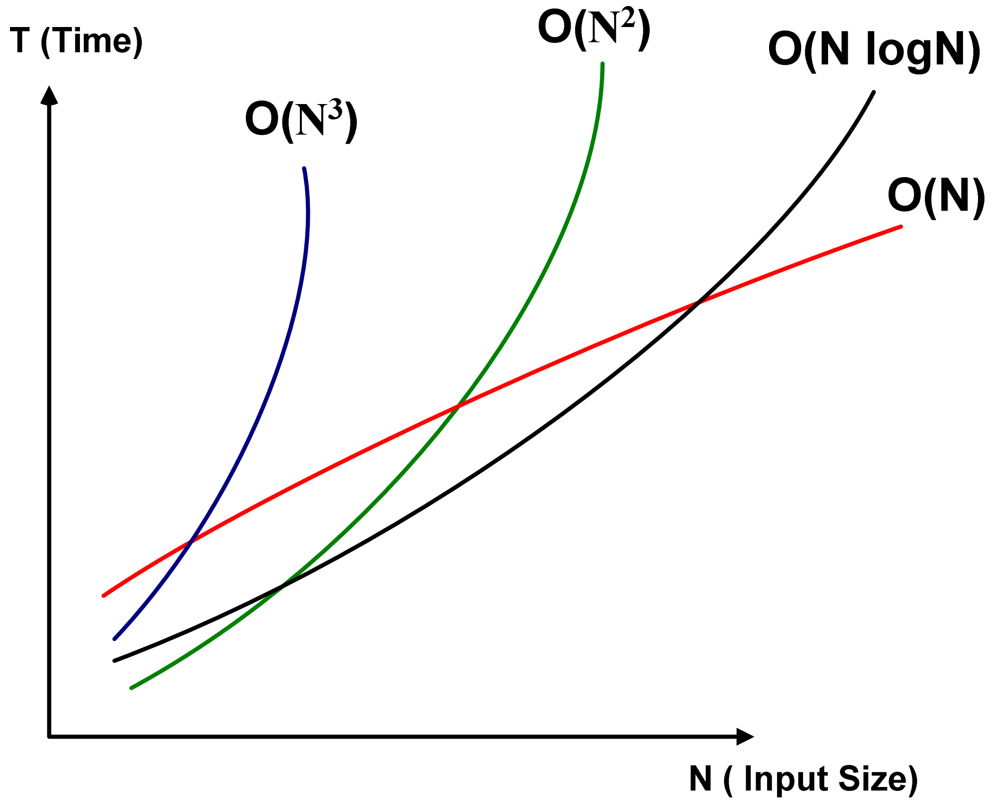
-2 11 -4 13 -5 -2

A[2,4] → sum = 20

A[2,5] → sum = 15

- There are many algorithms to solve this problem !

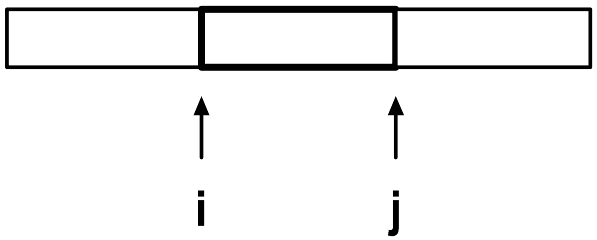
MAXIMUM SUBSEQUENCE SUM PROBLEM



- for small N all algorithms are fast
- for large N O(N) is the best

MAXIMUM SUBSEQUENCE SUM PROBLEM

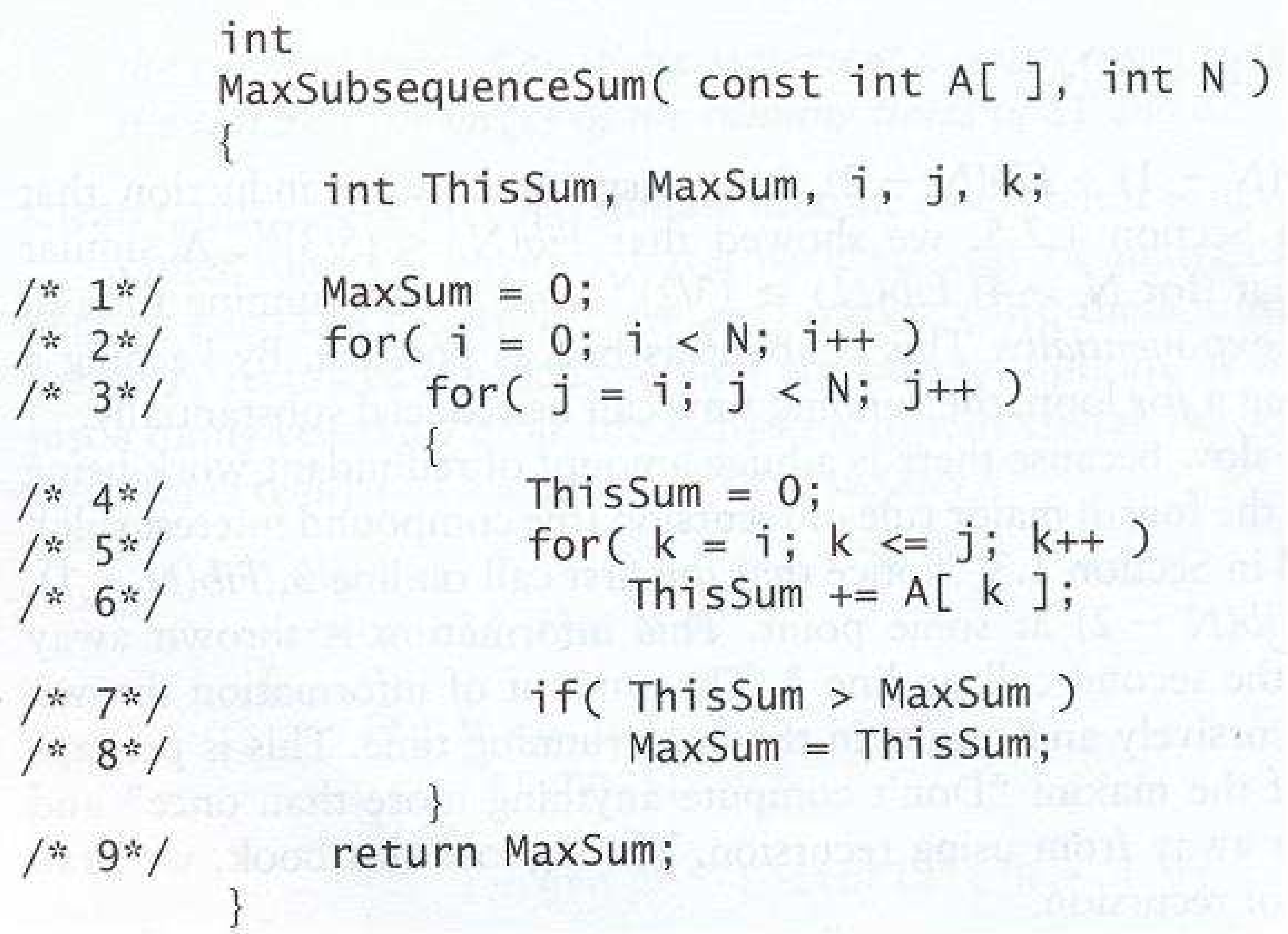
1) Try all possibilities exhaustively



```
for each i (0 to N-1)
   for each j (i to N-1)
        compute the sum of subsequence for (i to j)
        check if it is maximum
```

$T(N) = O(N^3)$

Algorithm 1:



```
        int
        MaxSubsequenceSum( const int A[ ], int N )
        {
            int ThisSum, MaxSum, i, j, k;

/* 1*/      MaxSum = 0;
/* 2*/      for( i = 0; i < N; i++ )
/* 3*/          for( j = i; j < N; j++ )
                {
/* 4*/              ThisSum = 0;
/* 5*/              for( k = i; k <= j; k++ )
/* 6*/                  ThisSum += A[ k ];

/* 7*/              if( ThisSum > MaxSum )
/* 8*/                  MaxSum = ThisSum;
                }
/* 9*/      return MaxSum;
        }
```

MAXIMUM SUBSEQUENCE SUM PROBLEM

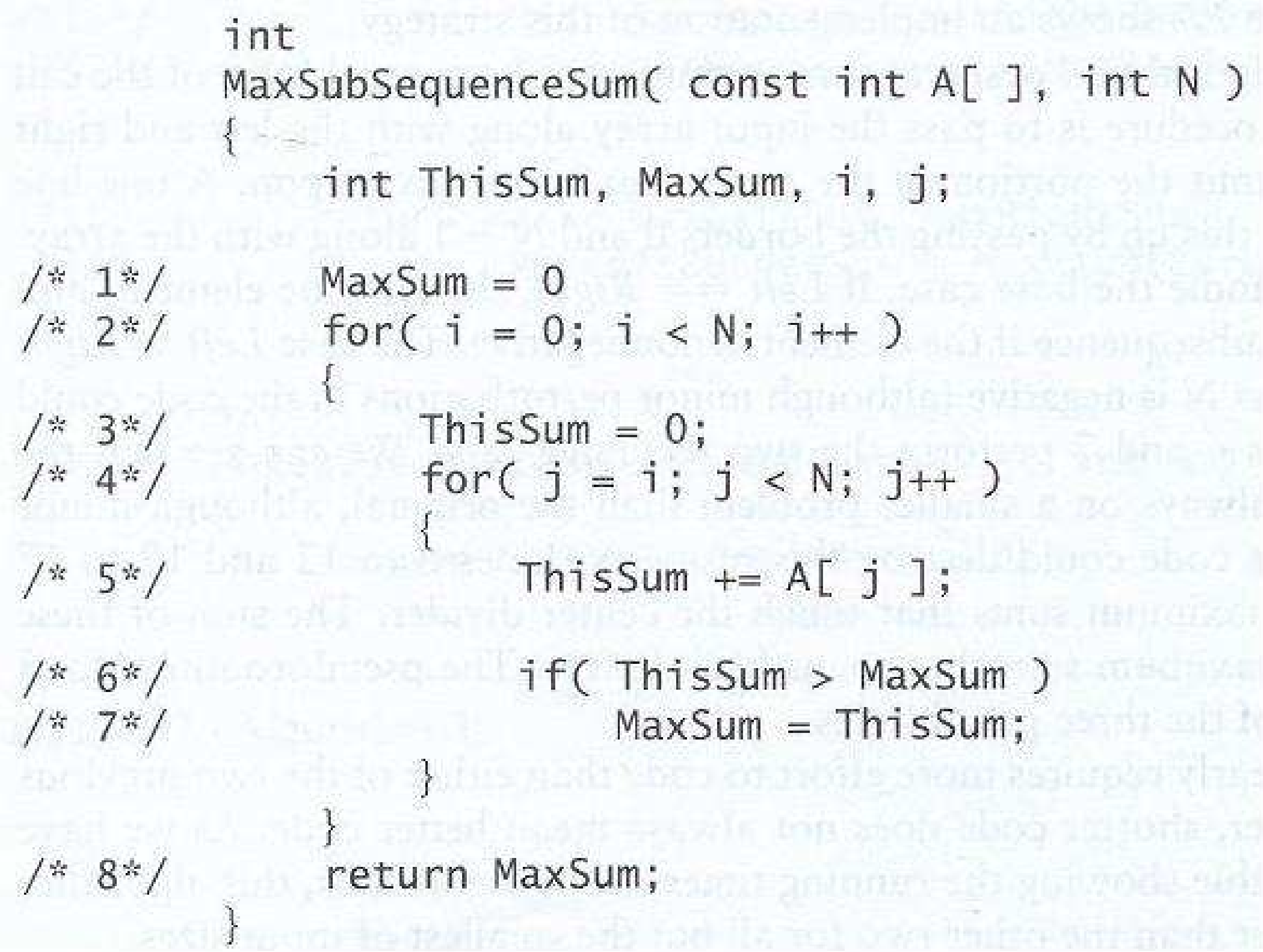
2) How to compute sum of a subsequence

$$\sum_{k=i}^{j} A_k = A_j + \sum_{k=i}^{j-1} A_k$$

- We can use previous subsequence sum to calculate current one in O(1) time

T(N) = O(N^2)

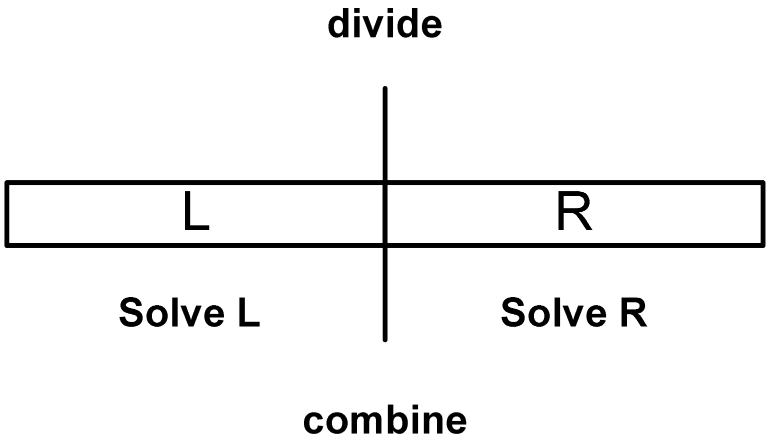
Algorithm 2:



```
        int
        MaxSubSequenceSum( const int A[ ], int N )
        {
                int ThisSum, MaxSum, i, j;

/* 1*/          MaxSum = 0
/* 2*/          for( i = 0; i < N; i++ )
                {
/* 3*/              ThisSum = 0;
/* 4*/              for( j = i; j < N; j++ )
                    {
/* 5*/                  ThisSum += A[ j ];

/* 6*/                  if( ThisSum > MaxSum )
/* 7*/                      MaxSum = ThisSum;
                    }
                }
/* 8*/          return MaxSum;
        }
```

MAXIMUM SUBSEQUENCE SUM PROBLEM

3) Compeletely different algorithm ?
Divide and Conquer Strategy



Maximum subsequence can be

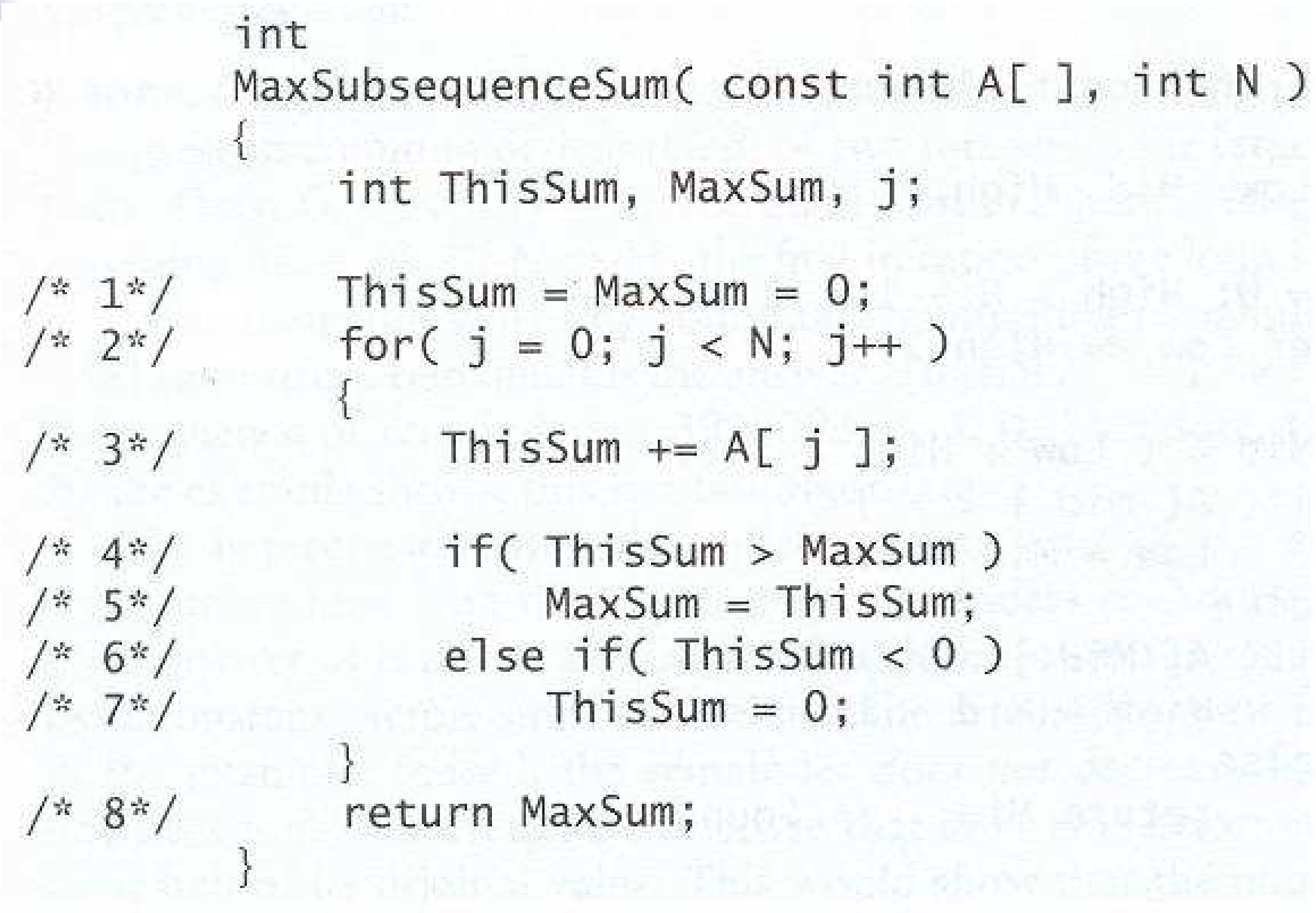
- in L
- in R

solved recursively

- in the middle, in both sides
  largest sum in L ending with middle element
  +
  largest sum in R begining with middle element

```
static int
MaxSubSum( const int A[ ], int Left, int Right )
{
        int MaxLeftSum, MaxRightSum;
        int MaxLeftBorderSum, MaxRightBorderSum;
        int LeftBorderSum, RightBorderSum;
        int Center, i;

        if( Left == Right )   /* Base Case */
            if( A[ Left ] > 0 )
                return A[ Left ];
            else
                return 0;

        Center = ( Left + Right ) / 2;
        MaxLeftSum = MaxSubSum( A, Left, Center );
        MaxRightSum = MaxSubSum( A, Center + 1, Right );

        MaxLeftBorderSum = 0; LeftBorderSum = 0
        for( i = Center; i >= Left; i-- )
        {
            LeftBorderSum += A[ i ];
            if( LeftBorderSum > MaxLeftBorderSum )
                MaxLeftBorderSum = LeftBorderSum;
        }

        MaxRightBorderSum = 0; RightBorderSum = 0;
        for( i = Center + 1; i <= Right; i++ )
        {
            RightBorderSum += A[ i ];
            if( RightBorderSum > MaxRightBorderSum )
                MaxRightBorderSum = RightBorderSum;
        }

        return Max3( MaxLeftSum, MaxRightSum,
                MaxLeftBorderSum + MaxRightBorderSum );
}
```

Algorithm 4:

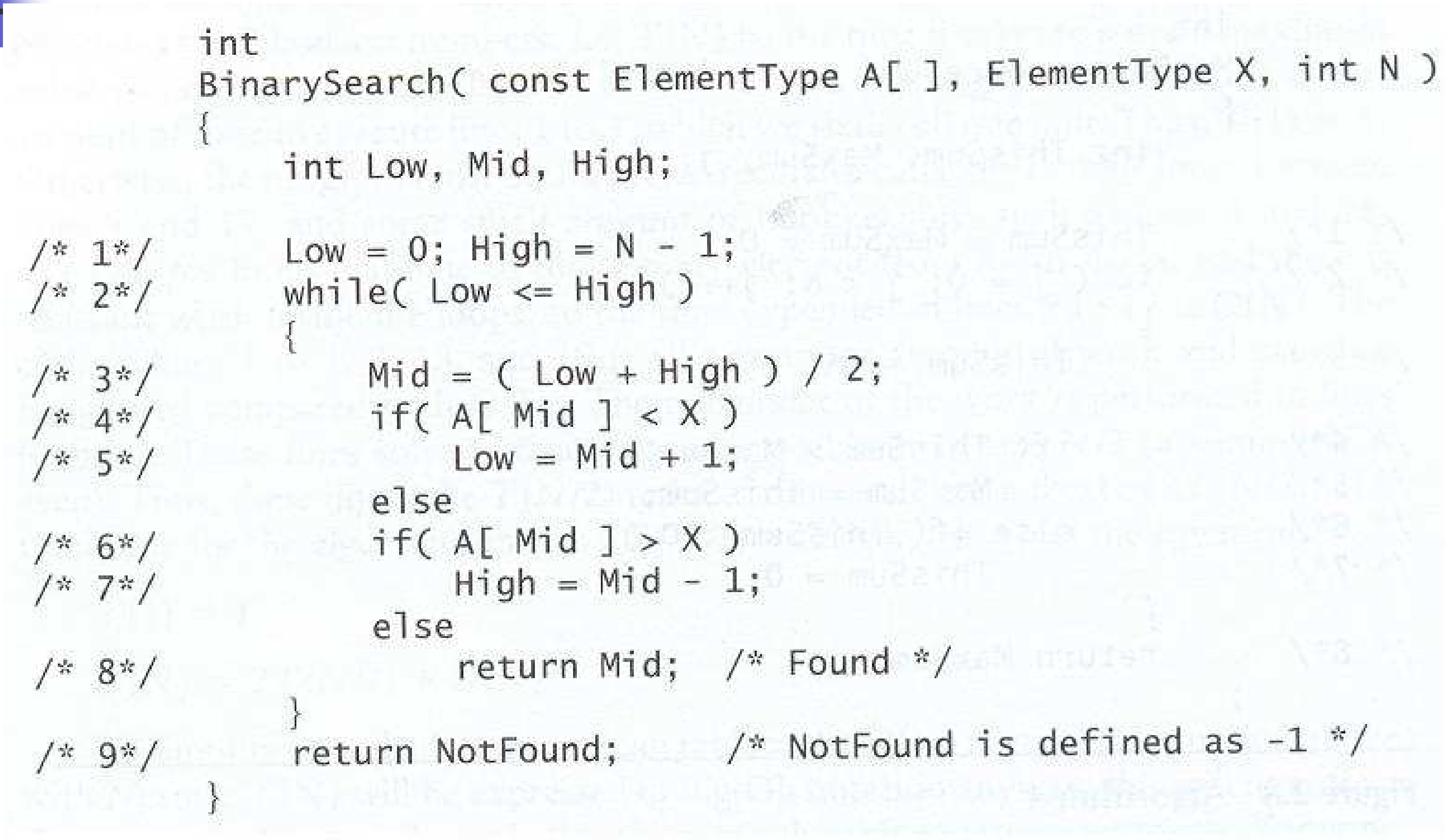


```
        int
        MaxSubsequenceSum( const int A[ ], int N )
        {
            int ThisSum, MaxSum, j;

/* 1*/      ThisSum = MaxSum = 0;
/* 2*/      for( j = 0; j < N; j++ )
            {
/* 3*/          ThisSum += A[ j ];

/* 4*/          if( ThisSum > MaxSum )
/* 5*/              MaxSum = ThisSum;
/* 6*/          else if( ThisSum < 0 )
/* 7*/              ThisSum = 0;
            }
/* 8*/      return MaxSum;
        }
```

ROAD MAP

- Model
- What to Analyze ?
- Running Time Analysis
- General Rules
- Recursive Calls
- Maximum Subsequence Sum Problem
- **Binary Search**
- Experimentally checking analysis

Binary Search



```
int
BinarySearch( const ElementType A[ ], ElementType X, int N )
{
    int Low, Mid, High;

    Low = 0; High = N - 1;
    while( Low <= High )
    {
        Mid = ( Low + High ) / 2;
        if( A[ Mid ] < X )
            Low = Mid + 1;
        else
        if( A[ Mid ] > X )
            High = Mid - 1;
        else
            return Mid;   /* Found */
    }
    return NotFound;      /* NotFound is defined as -1 */
}
```

ROAD MAP

- Model
- What to Analyze ?
- Running Time Analysis
- General Rules
- Recursive Calls
- Maximum Subsequence Sum Problem
- Binary Search
- **Checking your analysis experimentally**

Checking Your Analysis Experimentally

- Implement your algorithm, measure the running time of your implementation
  - check with theoretical results

- When N doubles (2x)
  - linear => 2x
  - quadratic => 4x
  - cubic => 8x

what about logarithm ?

O(N) O(N logN)

not easy to see the difference !

Checking Your Analysis Experimentally

$T_t(N)$ = theoretical running time
$T_e(N)$ = experimental running time

Let $T_t(N) = O(f(N))$

compute $$\frac{T_e(N)}{T_t(N)} = \frac{T_e(N)}{f(N)}$$

- if converges to a constant
  f(N) is a tight bound
- if converges to zero
  not tight bound (overestimate)
- if diverges
  underestimate

Checking Your Analysis Experimentally

N	**T_t**	**T_e**	**T_e / T_t**

Checking Your Analysis Experimentally