DATA STRUCTURES AND ALGORITHMS

Lecture Notes 2

Prepared by İnanç TAHRALI
Recapture

- Asymptotic Notations
  - $\mathcal{O}$ Notation
  - $\Omega$ Notation
  - $\Theta$ Notation
  - $o$ Notation
Big Oh Notation (O)

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

**Definition :**
- $T(N) = O\left(f(N)\right)$ if there are positive constants $c$ and $n_0$ such that
  $T(N) \leq c f(N)$ when $N \geq n_0$
  - $T(N)$ 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 ($\Omega$)

**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 ($\theta$)

**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:**

\[ T(N) = o(p(N)) \text{ if } T(N) = O(p(N)) \text{ and } T(N) \neq \theta(p(N)) \]

- \( p(N) \) 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
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
  - 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 affected by
- compiler
- computer
- algorithm

The compiler usually affects the constants & lower order terms.

Use asymptotic notation.
Running Time Analysis

- Empirical → after implementation
- Theoretical → 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:

```c
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
= \( \Theta(N) \)
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.

```c
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:

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

\[ T(n) = \theta(r \times q) \]
GENERAL RULES

- **RULE 3 : Consecutive Statements**
  Add the running times

\[
\text{for } \ldots
\]
\[
\text{for } \ldots
\]
\[
\text{for } \ldots
\]
\[
\theta(N^2) \quad \theta(N) \quad \theta(N^2)
\]
GENERAL RULES

- RULE 4: If / Else

if (condition) 
  S1;

else
  S2;

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

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)
Types of complexity

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

\[ T_{\text{av}}(N) = \sum_{|I|=N} T(I) \cdot \Pr(I) \quad \rightarrow \text{difficult to compute} \]

\[ T_{\text{best}}(N) = \min_{|I|=N} \{ T(I) \} \]

\[ T_{\text{worst}}(N) \geq T_{\text{av}}(N) \geq T_{\text{best}}(N) \]

\[ T(n) = O(T_{\text{worst}}(n)) = \Omega(T_{\text{best}}(n)) \]
GENERAL RULES

- RULE 4: If / Else

\[
\begin{align*}
\text{if (condition)} & \quad \{ \quad T_3(n) \quad \} \\
\quad & \quad \{ \quad T_1(n) \quad \} \\
\text{else} & \\
\quad S2; & \quad \{ \quad T_2(n) \quad \}
\end{align*}
\]

\[
\begin{align*}
T_w(n) &= T_3(n) + \max(T_1(n), T_2(n)) \\
T_b(n) &= T_3(n) + \min(T_1(n), T_2(n)) \\
T_{av}(n) &= p(T)T_1(n) + p(F)T_2(n) + T_3(n)
\end{align*}
\]

\[
\begin{align*}
p(T) &\rightarrow p \text{ (condition = True)} \\
p(F) &\rightarrow p \text{ (condition = False)}
\end{align*}
\]
GENERAL RULES

Example:

\[
\text{if (condition)} \quad \begin{cases} 
  T_3(n) = \theta(n) \\
  T_1(n) = \theta(n^2) \end{cases} \\
\text{else} \quad \begin{cases} 
  T_2(n) = \theta(n) \\
  T(n) \end{cases}
\]

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

if \( p(T) = p(F) = \frac{1}{2} \)

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

\[
T(n) = O(n^2) = \Omega(n)
\]
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

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

\[ T(n) = \text{cost of evaluation of factorial of n} \]
\[ T(n) = 4 + T(n-1) \]
\[ T(1) = 2 \]
RECURSIVE CALLS

\[ T(n) = 4 + T(n-1) \]
\[ T(n) = 4 + 4 + T(n-2) \]
\[ T(n) = 4 + 4 + 4 + T(n-3) \]
\[ \ldots \]
\[ T(n) = k*4 + T(n-k) \quad \text{for} \quad k= n-1 \Rightarrow \]
\[ T(n) = (n-1)*4 + T(n-(n-1)) \]
\[ T(n) = (n-1)*4 + T(1) \]
\[ T(n) = (n-1)*4 + 2 \]
\[ T(n) = \theta (n) \]
**Example 2:**
Algorithm for fibonacci series

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

\[ T(N) = T(N-1) + T(N-2) + 4, \quad T(1)=T(0)=2 \]

\[ \Rightarrow T(N) = O(2^n) \quad \text{(by induction)} \]
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

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

- Example:
  
  
<table>
<thead>
<tr>
<th></th>
<th>-2</th>
<th>11</th>
<th>-4</th>
<th>13</th>
<th>-5</th>
<th>-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A[2,5]</td>
<td>→ sum = 15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 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

\[
\begin{array}{c}
  i \\
\end{array}
\begin{array}{c}
  j \\
\end{array}
\]

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:

```c
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:

```c
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) Completely different algorithm?
Divide and Conquer Strategy

Maximum subsequence can be
- in L

solved recursively
- in R

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

    /* 1*/
    if( Left == Right ) /* Base Case */
    /* 2*/
    if( A[ Left ] > 0 )
        return A[ Left ];
    else
        return 0;

    /* 5*/
    Center = ( Left + Right ) / 2;
    /* 6*/
    MaxLeftSum = MaxSubSum( A, Left, Center );
    /* 7*/
    MaxRightSum = MaxSubSum( A, Center + 1, Right );

    /* 8*/
    MaxLeftBorderSum = 0; LeftBorderSum = 0
    /* 9*/
    for( i = Center; i >= Left; i-- )
    {
        /*10*/
        LeftBorderSum += A[ i ];
        /*11*/
        if( LeftBorderSum > MaxLeftBorderSum )
            MaxLeftBorderSum = LeftBorderSum;
    }

    /*13*/
    MaxRightBorderSum = 0; RightBorderSum = 0;
    /*14*/
    for( i = Center + 1; i <= Right; i++ )
    {
        /*15*/
        RightBorderSum += A[ i ];
        /*16*/
        if( RightBorderSum > MaxRightBorderSum )
            MaxRightBorderSum = RightBorderSum;
    }

    /*18*/
    return Max3( MaxLeftSum, MaxRightSum,
                 MaxLeftBorderSum + MaxRightBorderSum );
}
Algorithm 4:

```c
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 )
           MaxSum = ThisSum;

        /* 5*/
        else if( ThisSum < 0 )
           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

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

    /* 1*/
    Low = 0; High = N - 1;
    /* 2*/
    while( Low <= High )
    {
        /* 3*/
        Mid = ( Low + High ) / 2;
        /* 4*/
        if( A[ Mid ] < X )
        /* 5*/
            Low = Mid + 1;
        else
            /* 6*/
            if( A[ Mid ] > X )
            /* 7*/
                High = Mid - 1;
            else
            /* 8*/
                return Mid; /* Found */
    }
    /* 9*/
    return Not Found; /* Not Found 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 $\Rightarrow 2x$
  - quadratic $\Rightarrow 4x$
  - cubic $\Rightarrow 8x$

what about logarithm?
- $O(N)$  $O(N \log N)$
- not easy to see the difference!
Checking Your Analysis Experimentally

\[ T_t (N) = \text{theoretical running time} \]
\[ T_e (N) = \text{experimental running time} \]

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

\[
\frac{T_e(N)}{T_t(N)} = \frac{T_e(N)}{f(N)}
\]

- 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

<table>
<thead>
<tr>
<th>N</th>
<th>$T_t$</th>
<th>$T_e$</th>
<th>$T_e/T_t$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Checking Your Analysis Experimentally