1. Introduction to Performance Analysis

• Goal: To analyze the efficiency of algorithms

• Definition:
  
  • The space complexity of a program is the amount of memory that it needs to run to completion.
  
  • The time complexity of a program is the amount of computer time that it needs to run to completion.

• Space Complexity

  • \( S(p) = C_p + V_p(I) \)
    
    ▪ \( S(p) \) is the total space requirement for program \( p \)
    ▪ \( C_p \) is the fixed space requirement of program \( p \).
    ▪ \( V_p(I) \) is the variable space requirement of instance \( I \) of program \( p \).

  • fixed space requirement: the space requirement that do not depend on the number and size of the program input and output. It includes the instruction space + simple variables space.

  • variable space requirement: the space requirement that depends on the particular instance \( I \) of the problem being solved.
• Example : /* only fixed space requirement */

```c
main()
{
    int a=7,b=15;
    printf("%d\n",a+b);
}
```

• Example :

```c
int fact(int n)
{
    if (n==0) return (1);
    else return(n * (n-1));
}
```

assume we need x bytes to store information of each recursive call, we need approx. total variable space of (n+1) * x bytes

• Time Complexity

• The time taken by a program

\[ T(p) = \text{compile time (fixed) + run (or execution) time } E_p(I) \]

- \( T_p \) is the total time requirement for program \( p \)
- \( E_p(I) \) is the total run time for program \( p \) with particular instance \( I \).

• For \( E_p(I) \), need to know a detailed knowledge of executable code and the time needed to perform each operation on specific hardware.

For example : \( c = a+b; \) \( \rightarrow \) load \( a \); load \( b \); add; store \( c \)

** very difficult.
• Use other methods to estimate $T(p)$

• use system command such as "time" in Unix to approximate the run
time. difficult to analyze!

• set a global counter in your program to count the number of steps that
a program needs to solve an instance $I$.

very difficult for a complex problem, we may need to find out the best,
average and worst case scenarios

Example : Add two arrays $a$ and $b$

```
for (i=0; i < rows; i++) /* count++ */
for (j=0; j < cols; j++) /* count++ */
c[i][j] = a[i][j]+b[i][j]; /* count++ */
```

Assume count = 0 (initially) and each step takes constant time, we
have

- i for-loop statement, executed $rows + 1$ times,
- j for-loop statement, executed $rows*(cols + 1)$ times,
- the statement in j-loop, executed $rows*cols$ times
- total counts : $2*rows*cols + 2*rows + 1$

If $rows >> cols$, should interchange the matrices to minimize the
total counts.

• Asymptotic notation : The approximation of step counts (Only cover
Big O here. You also need to know definitions of Big Omega $\Omega$ and
Big Theta $\Theta$)
• Def [Big O] : A function f(n) is said to be O(g(n)) iff there exist positive constants c and n₀ such that f(n) \leq c \cdot g(n) for all n, n \geq n₀. Read : f of n is big o of g of n

• O(g(n)) is an upper bound of f(n), should try to find as small g(n) as possible

• Example :
  - f(n) = 3n+2, O(n) because 3n+2 \leq 4n for all n \geq 2
  - f(n) = 10, O(1) because 10 \leq 10 \cdot 1 for all n > 0
  - f(n) = 10n²+4n+2, O(n²) because f(n) \leq 11 \cdot (n²) for all n \geq 5
  - f(n) = 6\cdot(2^n)+n², O(2^n) because f(n) \leq 7\cdot(2^n) for all n \geq 4

• You may think of f(n) is the running time of program, n is the input size of the program and g(n) is the approximate upper bound of f(n), i.e. worst case

• Example 1: For matrix addition, we have

  \begin{align*}
  \text{O(rows)} & \quad \text{for i for-loop statement} \\
  \text{O(rows*cols)} & \quad \text{for j for loop statement} \\
  \text{O(rows*cols)} & \quad \text{for the statement in j for loop}
  \end{align*}

  \text{Total : O(rows*cols)}
• Example 2: Binary Search

each iteration: constant c time to find middle element; compare elements & discard half of the list

\# elements after each iteration: \( n \rightarrow \frac{n}{2} \rightarrow \frac{n}{4} \rightarrow \frac{n}{2^i} \rightarrow \ldots \rightarrow 1 \)

we have \( \log n \) iterations; total time = \( c \times \log n = O(\log n) \)

• Example 3: factorial

each iteration, constant c time,
each time we reduce the number \( n \) by 1
we have \( n \) iterations
total time = \( c \times n = O(n) \)

• Main Problem: constant numbers and lower terms are eliminated. They may be a very large number.

• Here are the list of common time complexities:

\[ O(1), O(\log n), O(n), O(n \log n), O(n^2), O(2^n) \text{ etc} \]

<table>
<thead>
<tr>
<th>Time</th>
<th>Name</th>
<th>( n \rightarrow )</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Constant</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>log</td>
<td>Logarithmic</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>( n )</td>
<td>Linear</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>( n \log n )</td>
<td>Log linear</td>
<td>0</td>
<td>2</td>
<td>8</td>
<td>24</td>
<td>64</td>
<td>160</td>
<td></td>
</tr>
<tr>
<td>( n^2 )</td>
<td>Quadratic</td>
<td>1</td>
<td>4</td>
<td>16</td>
<td>64</td>
<td>256</td>
<td>1024</td>
<td></td>
</tr>
<tr>
<td>( n^3 )</td>
<td>Cubic</td>
<td>1</td>
<td>8</td>
<td>64</td>
<td>512</td>
<td>4096</td>
<td>32768</td>
<td></td>
</tr>
<tr>
<td>( 2^n )</td>
<td>Exponential</td>
<td>2</td>
<td>4</td>
<td>16</td>
<td>256</td>
<td>65536</td>
<td>4294967296</td>
<td></td>
</tr>
<tr>
<td>( n! )</td>
<td>Factorial</td>
<td>1</td>
<td>2</td>
<td>24</td>
<td>40326</td>
<td>2E13</td>
<td>4E47</td>
<td></td>
</tr>
</tbody>
</table>
2. Mathematical Induction and Recursion

• Recursion:
  
  base case: solve directly
  arbitrary size: break into smaller problems

• Mathematical Induction:
  
  base case: prove it is correct directly
  arbitrary size n: prove it is correct by assuming OK m < n

• Use Mathematical induction to prove correctness of recursive algorithms and to compute the run time of recursive algorithms

• The correctness of the recursive factorial function

fact(n) {
    if (n == 0) return 1;
    else return (n * fact(n-1));
}

Proof by induction on n

Basis: To show fact(0) returns 1, i.e. 0! = 1

    fact() correctly return 1 when n = 0. Obvious case.

Inductive Hypothesis: assume fact(k) correctly compute k!
Inductive step: show that fact(k+1) correctly compute (k+1)!

\[
\text{fact}(k+1) = (k+1) \times \text{fact}(k); \quad \text{// from the program}
\]

by the hypothesis, \( \text{fact}(k) = k! \)

\[
\Rightarrow \text{fact}(k+1) = (k+1) \times k! = (k+1)! \quad < \text{DONE} >
\]

• The running time of Towers of Hanoi

Let \( T(n) \) be the total number of moves for \( n \) disks

- If \( n = 1 \) disk \( \Rightarrow T(1) = 1 \)
- If \( n > 1 \) disks \( \Rightarrow T(n) = T(n-1) + T(1) + T(n-1) = 2T(n-1) + 1 \)

Guess: \( T(n) = O(2^n - 1) \)

Use mathematical induction to show that \( T(n) = O(2^n - 1) \)

Basis: \( n = 1, T(n) = 2^1 - 1 = 1 \)

Inductive Hypothesis: assume \( T(k) = 2^k - 1 \)

Inductive step: show that \( T(k+1) = 2^{(k+1)} - 1 \)

\[
\begin{align*}
T(k+1) &= 2T(k) + 1 \\
&= 2(2^k - 1) + 1 \quad \text{(by hypothesis)} \\
&= 2^{(k+1)} - 1
\end{align*}
\]

Conclusion: the running time of Towers of Hanoi algorithm is \textit{exponential}!
3. Key property of prefix expression

Let $T[1..n]$ be a string, if $E = T[1..k]$ is a prefix expression and let $Y = T[k+1..m]$ and $k < m \leq n$, then $EY$ cannot be a prefix expression, i.e. only one end point is possible for prefix $E$.

Show by induction on number of chars in $E$, i.e. $|E|$ :

Recall : 
- $<\text{prefix}> = <\text{identifier}> | <\text{operator}><\text{prefix}><\text{prefix}>
- $<\text{operator}> = + | - | * | /
- $<\text{identifier}> = a | b | \ldots | z$

- Basis : If $|E| = 1$, then $E$ is single letter $\Rightarrow$ $EY$ cannot be prefix

- Inductive Hypothesis : Assume $|E| < n$ and $E$ is prefix, then $EY$ cannot be prefix

- Show : $|E| = n$ and $E$ is prefix $\Rightarrow$ $EY$ cannot be prefix

Let $E = <\text{operator}> E_1 E_2$, where $E_1$ and $E_2$ are prefix and $|E_1| < n$ and $|E_2| < n$

Assume $EY$ is prefix for some non-empty string $Y$
i.e. $EY = <\text{operator}> W_1 W_2$, $W_1$ and $W_2$ are prefix

If $|W_1| > |E_1| \Rightarrow W_1 = E_1 W^*$ cannot be prefix.
By hypothesis, $E_1$ is prefix $\Rightarrow E_1 W^*$ cannot be prefix.

If $|W_1| < |E_1| \Rightarrow E_1 = W_1 E^*$ cannot be prefix.
By hypothesis, $W_1$ is prefix $\Rightarrow W_1 E^*$ cannot be prefix.
This cannot be true since $E_1$ is prefix.

Therefore, $W_1 = E_1 \Rightarrow EY = <\text{operator}> E_1 W_2$

By the same argument, $W_2 = E_2$ and $Y$ is empty $\Rightarrow$ contradiction.