Data structures and Algorithms
Basic definitions and notations

Pham Quang Dung

Hanoi, 2012

Outline

First example

Find the longest subsequence of a given sequence of numbers

- Given a sequence $s = \langle a_1, \ldots, a_n \rangle$
- a subsequence is $s(i,j) = \langle a_i, \ldots, a_j \rangle$, $1 \le i \le j \le n$
- weight $w(s(i,j)) =$

$$\sum_{k=i}^{j} a_k$$

- Problem : find the subsequence having largest weight

Example

- sequence : -2, 11, -4, 13, -5, 2
- The largest weight subsequence is 11, -4, 13 having weight 20

Direct algorithm

- Scan all possible subsequences $\binom{n}{2} = \frac{n^2+n}{2}$
- Compute and keep the largest weight subsequence
- C++ code :

```
int maxSum = 0;
for(int i = 0; i < n; i++){
     for(int j = i; j < n; j++){
          int sum = 0;
          for(int k = i; k <= j; k++)
               sum += a[k];
          if(maxSum < sum)
               maxSum = sum;
     }
}
```

- Analyzing the complexity by counting the number of basic operations
- $\frac{n^3}{6} + \frac{n^2}{2} + \frac{n}{3}$

- Observation : $\sum_{k=i}^{j} a[k] = a[j] + \sum_{k=i}^{j-1} a[k]$
- C++ code :

```
int maxSum = 0;
for(int i = 0; i < n; i++){
    int sum = 0;
    for(int j = i; j < n; j++){
        sum += a[j];
        if(maxSum < sum)
            maxSum = sum;
    }
}
```

- Complexity : $\frac{n^2}{2} + \frac{n}{2}$

Recursive algorithm

- Divide the sequence into 2 subsequences at the middle $s = s_1 :: s_2$
- The largest subsequence might
  - be in s_1 or
  - be in s_2 or
  - start at some position of s_1 and end at some position of s_2
- C++ code :

```
int maxLeft(int i, int j){
    int maxSum = -1000000;
    int sum = 0;
    for(int k = j; k >= i; k--){
        sum = sum + a[k];
        if(maxSum < sum) maxSum = sum;
    }
    return maxSum;
}
```

```
int maxRight(int i, int j){
    int maxSum = -10000000;
    int sum = 0;
    for(int k = i; k <= j; k++){
        sum = sum + a[k];
        if(maxSum < sum) maxSum = sum;
    }
    return maxSum;
}
```

```
int maxSub(int i, int j){
    if(i == j) return a[i];
    int mid = (i+j)/2;
    int mL = maxSub(i,mid);
    int mR = maxSub(mid+1,j);
    int mM = maxLeft(i,mid) + maxRight(mid+1,j);
    int maxSum = mL;
    if(maxSum < mR) maxSum = mR;
    if(maxSum < mM) maxSum = mM;
    return maxSum;
}
```

Recursive algorithm

- Count the number of addition ("+") operation $T(n)$

$$T(n) = \begin{cases} 0 & \text{if} \quad n = 1 \\ T(\frac{n}{2}) + T(\frac{n}{2}) + n & \text{if} \quad n > 1 \end{cases}$$

- By induction : $T(n) = n\ logn$

Dynamic programming

General principle

- Division : divide the initial problem into smaller similar problems (subproblems)
- Storing solutions to subproblems : store the solution to subproblems into memory
- Aggregation : establish the solution to the initial problem by aggregating solutions to subproblems stored in the memory

Largest subsequence

- Division :
  - Let s_i be the weight of the largest subsequence of $a_1, \ldots, a_i$ ending at a_i
- Aggregation :
  - $s_1 = a_1$
  - $s_i = \max\{s_{i-1} + a_i, a_i\}, \forall i = 2, \ldots, n$
  - Solution to the original problem is $\max\{s_1, \ldots, s_n\}$
- Number of basic operations is n (best algorithm)

Comparison between algorithms

$\sharp$ operations	$n = 10$	time	$n = 100$	time
$logn$	3.32	3.3×10^{-8} sec.	6.64	$6\times10^{-8} sec.$
$nlogn$	33.2	3.3×10^{-7} sec.	664	6.6×10^{-6} sec.
n^2	100	10^{-6} sec.	10000	10^{-4} sec.
n^3	10^3	10^{-5} sec.	10^6	10^{-2} sec.

$\sharp$ operations	$n = 10^4$	time	$n = 10^6$	time
$logn$	13.3	10^{-6} sec.	19.9	$< 10^{-5} sec.$
$nlogn$	1.33×10^5	$\times10^{-3}$ sec.	1.99×10^7	2×10^{-1} sec.
n^2	10^8	1 sec.	10^12	2.77 hours
n^3	10^12	2.7 hours	10^18	115 days

Outline

Algorithms and complexity

Definition

Informally, an algorithm is any well-defined computational procedure that takes a set of values, as **input**, and produces a set of values, as **output**

- Input
- Output
- Precision
- Finiteness
- Uniqueness
- Generality

Complexity

Criteria for evaluating the complexity of an algorithm

- Memory
- CPU time

Definition

The size of input is defined to be the number of necessary bits for representing it

Definition

The basic operations are the operations which can be performed in a bounded time, and do not depend on the size of the input

We evaluate the complexity of an algorithm in term of the number of basic operations it performs

Complexity

- Worst-case time complexity :
  - The longest execution time the algorithm takes given any input of size n
  - Used to compare the efficiency of algorithms
- Average-case time complexity : execution time the algorithm takes on a random input
- Best-case time complexity : The smallest execution time the algorithm takes given any input of size n

Outline

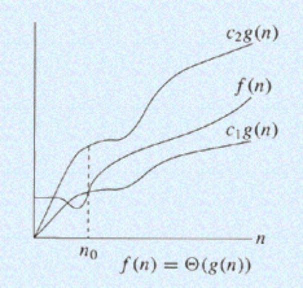
Big-Oh notation

Given a fucntion $g(n)$, we denote :

- $\Theta(g(n)) = \{f(n) : \exists c_1, c_2, n_0 \text{ s.t. } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq n_0\}$

Example :

- $10n^2 - 3n = \Theta(n^2)$



source : http ://www.personal.kent.edu/ rmuhamma/Algorithms/MyAlgorithms/intro.htm

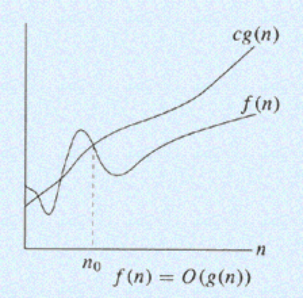
Big-Oh notation

Given a fucntion $g(n)$, we denote :

- $\mathcal{O}(g(n)) = \{f(n) : \exists c, n_0 > 0 \text{ s.t. } f(n) \leq cg(n), \forall n \geq n_0\}$

Example :

- $2n^2 = \mathcal{O}(n^3)$



source : http ://www.personal.kent.edu/ rmuhamma/Algorithms/MyAlgorithms/intro.htm

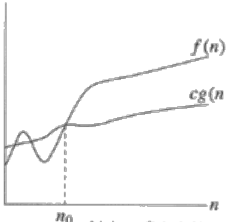
Big-Oh notation

Given a fucntion $g(n)$, we denote :

- $\Omega(g(n)) = \{f(n) : \exists c, n_0 > 0 \text{ s.t. } cg(n) \leq f(n), \forall n \geq n_0\}$

Example :

- $5n^2 = \Omega(n)$



$f(n) = \Omega(g(n))$

source : http ://www.personal.kent.edu/ rmuhamma/Algorithms/MyAlgorithms/intro.htm

Big-Oh notation

- When we say "the time complexity is $\mathcal{O}(f(n))$" : the time complexity in the worst case is $\mathcal{O}(f(n))$
- When we say "the time complexity is $\Omega(f(n))$" : the time complexity in the best case is $\Omega(f(n))$

Little-o notation

Given a fucntion $g(n)$, we denote :

- $o(g(n)) = \{f(n) : \forall$ constant $c > 0, \exists n_0 > 0$ s.t. $0 \leq f(n) < cg(n), \forall n \geq n_0\}$

Example :

- $5n^2 = o(n^3)$

Little-o notation

Given a fucntion $g(n)$, we denote :

- $\omega(g(n)) = \{f(n) : \forall \text{ constant } c > 0, \exists n_0 > 0 \text{ s.t. } 0 \leq cg(n) < f(n), \forall n \geq n_0\}$

Example :

- $5n^2 = \omega(n^{1.5})$

Big-Oh notation

- $$\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty \Rightarrow f(n) = \mathcal{O}(g(n))$$
- $$\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0 \Rightarrow f(n) = \Omega(g(n))$$
- $$0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty \Rightarrow f(n) = \Theta(g(n))$$
- $$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f(n) = o(g(n))$$
- $$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \Rightarrow f(n) = \omega(g(n))$$

Big-Oh notation

- $\mathcal{O}(lgn) = \mathcal{O}(lnn)$
- $lg^a n = o(n^b)$ where a, b are constant
- $n! = o(n^n)$
- $n! = \omega(2^n)$
- $logn! = \Theta(nlogn)$

Big-Oh notation

Example

$A = log_3(n^2)$ vs. $B = log_2(n^3)$

- $A = 2log_3 n = 2lnn/ln3$ where we denote $lnx = log_e(x)$
- $B = 3log_2 n = 3lnn/ln2$
- $\frac{A}{B} = constant \Rightarrow A = \Theta(B)$

Big-Oh notation

Example

$A = n^{lg 4}$ vs. $B = 3^{lg n}$ where we denote $lg x = log_2 x$

- $B = 3^{lg n} = n^{lg 3}$ $(log_b a = log_a b)$
- $\frac{A}{B} = n^{lg(4/3)} \rightarrow \infty$
- $A = \omega(b)$

Example

$A = lg^2 n$ vs. $B = n^{1/2}$ where we denote $lgx = log_2 x$

- $$\lim_{n\to\infty} \frac{A}{B} = \lim_{n\to\infty} \frac{lg^2 n}{n^{1/2}} = 0$$
- $\Rightarrow A = o(B)$

Big-Oh notation

Properties

- $f(n) = \Theta(g(n)) \wedge g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$
- $f(n) = \mathcal{O}(g(n)) \wedge g(n) = \mathcal{O}(h(n)) \Rightarrow f(n) = \mathcal{O}(h(n))$
- $f(n) = \Omega(g(n)) \wedge g(n) = \Omega(h(n)) \Rightarrow f(n) = \Omega(h(n))$
- $f(n) = o(g(n)) \wedge g(n) = o(h(n)) \Rightarrow f(n) = o(h(n))$
- $f(n) = \omega(g(n)) \wedge g(n) = \omega(h(n)) \Rightarrow f(n) = \omega(h(n))$
- $f(n) = \Theta(g(n))$ iff $g(n) = \Theta(f(n))$
- $f(n) = \mathcal{O}(g(n))$ iff $g(n) = \Omega(f(n))$
- $f(n) = o(g(n))$ iff $g(n) = \omega(f(n))$

Outline

Pseudo code

Variables declaration :

- **integer** x, y ;
- **real** u,v ;
- **boolean** a,b ;
- **char** c,d ;
- **datatype** x ;

Pseudo code

Assignment instruction :

- x = expression ;
- x $\leftarrow$ expression ;
- $x \leftarrow x + 3$;

Condition instruction

if condition **then**

    instructions ;

else

    instructions ;

endif ;

Pseudo code

Loop

while condition **do**
    instructions ;
endwhile ;

repeat   instructions ;
until conddition ;

for $i = n_1$ to n_2 [step d]
    instructions ;
endfor

Pseudo code

Case instruction

Case
    condition 1 : statement 1 ;
    condition 2 : statement 2 ;
    . . .
    condition n : statement n ;
endcase

Pseudo code

Functions and Procedures

```
Function name(parameters)
begin
    instructions ;
  return value ;
end
```

```
Procedure name(parameters)
begin
    instructions ;
end
```

Pseudo code

Example : Find the maximal value of an array $A(1:n)$

Function $\max(A(1:n))$
begin
    datatype x ;
    integer i ;
    $x = A[1]$;
    for $i = 2$ to n **do**
        if $x < A[i]$ **then**
            $x = A[i]$;
        endif
    endfor
    return x ;
end

Pseudo code

Algorithm 1: max(A)

$n \leftarrow A.length$;
$x \leftarrow A[1]$;
foreach $i \in 2..n$ **do**
    if $x < A[i]$ **then**
        $x \leftarrow A[i]$;
return x;

Outline

Analysis of algorithms

Experiments studies

- Write a program implementing the algorithm
- Execute the program on a machine with different input sizes
- Measure the actual execution times
- Plot the results

Analysis of algorithms

Shortcomings of experiments studies

- Need to implement the algorithm, sometime difficult
- Results may not indicate the running time of other input not experimented
- To compare two algorithms, it is required to use the same hardware and software environments.

Analysis of algorithms

Asymptotic algorithm analysis

- Use high-level description of the algorithm (pseudo code)
- Determine the running time of an algorithm as a function of the input size
- Express this function with Big-Oh notation

Analysis of algorithms

- Sequential structure : P and Q are two segments of the algorithm (the sequence $P; Q$)
  - $\text{Time}(P; Q) = \text{Time}(P) + \text{Time}(Q)$ or
  - $\text{Time}(P; Q) = \Theta(max(Time(P), Time(Q)))$
- **for** loop : **for** $i = 1$ to m **do** $P(i)$
  - $t(i)$ is the time complexity of $P(i)$
  - time complexity of the **for** loop is $\sum_{i=1}^{m} t(i)$

while (**repeat**) loop

- Specify a function of variables of the loop such that this function reduces during the loop
- To evaluate the running time, we analyze how the function reduces during the loop

Example : binary search

```
Function BinarySearch(T[1..n], x)
begin
    i ← 1 ; j ← n ;
    while i < j do
        k ← (i + j)/2 ;
        case
            x < T[k] : j ← k − 1 ;
            x = T[k] : i ← k ; j ← k ; exit ;
            x > T[k] : i ← k + 1 ;
        endcase
    endwhile
end
```

Example : binary search

Denote

- $d = j - i + 1$ (number of elements of the array to be investigated)
- i^*, j^*, d^* respectively the values of i, j, d after a loop

We have

- If $x < T[k]$ then $i^* = i$, $j^* = (i + j)/2 - 1$, $d^* = j^* - i^* + 1 \leq d/2$
- If $x > T[k]$ then $j^* = j$, $i^* = (i + j)/2 + 1$, $d^* = j^* - i^* + 1 \leq d/2$
- If$x = T[k]$ then $d^* = 1$

Hence, the number of iterations of the loop is $\lceil logn \rceil$

Primitive operations

Function Fib(n)
begin
  $i \leftarrow 0$; $j \leftarrow 1$;
  for $k = 2$ **to** n **do**
  begin
    $j \leftarrow j + i$;
    $i \leftarrow j - i$;
  end
  return j ;
end

Primitive operation is $j \leftarrow j + i$, hence, the running time is $\mathcal{O}(n)$

Primitive operations (be careful ! !)

Procedure PigeonholeSorting($T[1..n]$)
begin
  for $i = 1$ **to** n **do**
    inc($U[T[i]]$) ;
  $i \leftarrow 0$;
  for $k = 1$ **to** s **do**
    while $U[k] > 0$ **do**
      $i \leftarrow i + 1$;
      $T[i] \leftarrow k$;
      $U[k] \leftarrow U[k] - 1$;
    endwhile
  endfor
end

Number of primitive operations is $\sum_{k=1}^{s} U[k] = n$. Hence running time is $\Theta(n)$ (But not correct !)

Primitive operations (be careful ! !)

- Consider the case $T[i] = i^2, \forall i = 1, \ldots, n$

$$U[k] = \begin{cases} 1, & \text{if } k = q^2 \\ 0, & \text{otherwise} \end{cases}$$

- $s = n^2$, the running time is $\Theta(n^2)$ not $\Theta(n)$
- Reason : The primitive operation is not well-chosen. Many null-loop where $U[k] = 0$
- If the primitive operation is the checking instruction $U[k] > 0$, then the running time is $\Theta(n + s) = \Theta(n^2)$