Algorithm Analysis 03

Computer Science Department
Eastern Washington University
Yun Tian (Tony) Ph.D.

Recall

- Big-Oh notation
- Rules of Big-Oh notation
- Growth Rate Function(GRF) with Big-Oh

Today Class

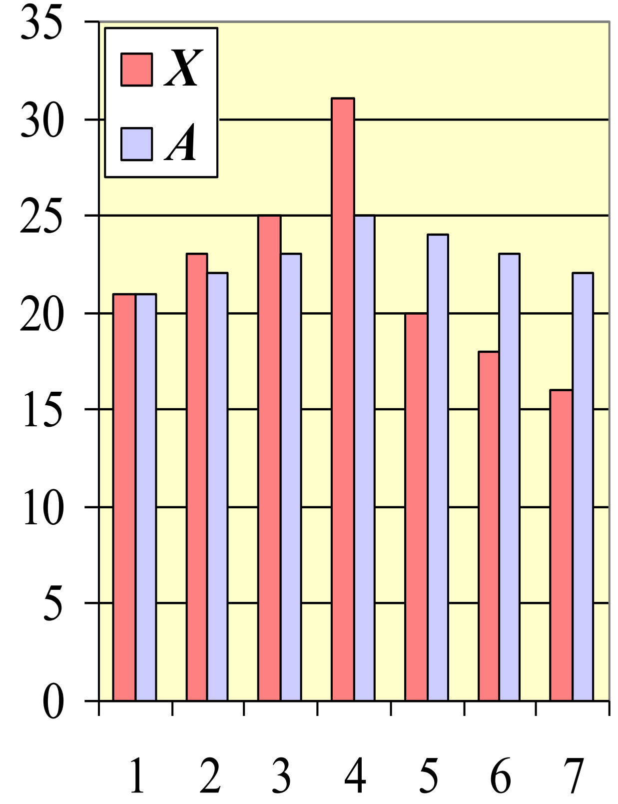
- Analysis Case Study

Computing Prefix Averages

- We further illustrate asymptotic analysis with two algorithms for prefix averages
- The i-th prefix average of an array X is average of the first $(i + 1)$ elements of X:

$$A[i] = (X[0] + X[1] + \ldots + X[i])/(i+1)$$

- Computing the array A of prefix averages of another array X has applications to financial analysis



Prefix Averages (Quadratic)

- The following algorithm computes prefix averages in quadratic time by applying the definition

Algorithm *prefixAverages1*(X, n)	
Input array X of n integers	
Output array A of prefix averages of X	#operations
$A \leftarrow$ new array of n integers	n *initializations*
for (i=0; i <= n – 1; i ++)	$n + 1$ *comparison*
$s \leftarrow X[0]$	n
for (j=1; j<= i; j ++)	$1 + 2 + \ldots + (n - 1)$
$s \leftarrow s + X[j]$	$1 + 2 + \ldots + (n - 1)$
$A[i] \leftarrow s / (i + 1)$	n
return A	1

Add them up: T(n) = n + (n + 1) + n + 2*(1 + 2 + 3.. + n-1) + n + 1

= $n^2 + 3n + 2$ This is the GRF, we can drop the lower-order term

We get the Big-Oh notation, T(n) is $O(n^2)$.

Prefix Averages (Linear)

- The following algorithm computes prefix averages in linear time by keeping a running sum

Algorithm ***prefixAverages2*****(*****X*****,** ***n*****)**	
Input array ***X*** of ***n*** integers	
Output array ***A*** of prefix averages of ***X***	#operations
A ← new array of ***n*** integers	***n***
s ← 0	1
for (i=0; i< =n – 1;i++)	***n+1 comparison***
s ← ***s*** + ***X*****[*****i*****]**	***n***
A**[*****i*****]** ← ***s*** / (***i*** + 1)	***n***
return ***A***	1

Add them up: T(n) = n + 1 + (n + 1) + 2n + 1=4n+3. This is the GRF, we can drop the lower-order term and constant factors,
We get the Big-Oh notation, T(n) is O(n). Algorithm ***prefixAverages2*** runs in ***O*****(*****n*****)** time .

Time Complexity

- Different solutions to a same problem could have different complexity.
- We take the dominant term in our GRF, throw away any constant part of that dominant term,
- And call that the time complexity of the algorithm.

Table of Time Complexity

- $2^n > n!$ or $2^n < n!$? As n goes very big.
- $2^n = 2 * 2 * \ldots\ldots * 2$
  - product of ***n*** 2s
- $n! = n * (n-1) * (n-2) .. * 3 * 2 * 1$
  - product of **n** items, with most of them bigger than 2.
  - So $2^n < n!$, as **n** goes towards infinity.

Table of Time Complexity

-

Big-Oh	Description	Examples
$O(1)$	constant time complexity	addFirst on a linked list (regardless of list representation)
$O(\log_2(n))$	logarithmic	Binary Search
$O(n)$	linear	toString on linked list, linear search
$O(n\log n)$	log linear	merge sort, quick sort
$O(n^2)$	quadratic	insertion, selection, bubble sort
$O(n^3)$	polynomial time	Matrix Multiplication
$O(2^n)$	exponential	Towers of Hanoi, recursive version of Fibonacci
$O(n!)$	permutations, combinations	Lottery, Permutations

Case Study One

- Factorial

```
int factorial_recursion(int n)   //assume n >= 0. Time cost: T(n)
{
   if(n == 0)                        // the exit condition
      return 1;                      // time cost: c
   return factorial_recursion (n-1) * n;  // recursive call, we move the '*n' operation to c above.
                                          //then time cost for last statement: T(n-1)
}
```

Time cost:

T(n) = c+T(n−1)
       = c+c+T(n−2)
       = c+c+...+c + T(1)   //We know T(1) takes c operations also.
       = c+c+c..+c + c   // How many c are here?
       = nc   //drop the constant factor c, so
                              //the recursive factorial algorithm runs in complexity of ***O(n)***

Case Study Two

```
int mystery3(int n, int i, int age) {
   if(n == 0)          1
      return 1;            1
   for(int j = 0; j < 10000; j ++)      10000 + 1
         n *= (i + age);   10000
   return n;  1
}
```

Time cost:

T(n) = 1 + 1 + (10000 + 1) + 10000 + 1

= 20004

//drop the constant factor, so this method mystery3() runs in **O(1)**.

//Constant time complexity.

// no matter how big the problem size is,

// the run time of this alogrithm is considered a constant.

Case Study Three

```
void mystery4(int n) {
   for( int i = n; i > 0; i = i /2)
      printf(i);
}
```

E.g Originally 16 (then / 2)

get 8 (then / 2)

get 4 (then / 2)

get 2 (then / 2)

get 1 (then / 2)

get 0 over

Think in an opposite way from the bottom to top, 1 * 2 * 2 * 2 * 2 = 16.

So if the original input value is **n**, We get similar relationship 1 * 2 * 2 * ....* 2 = n

Case Study Three

```
void mystery4(int n) {
    for( int i = n; i > 0; i = i /2)
        printf(i);
}
```

We repeatedly divide n by 2 until we get a number less than or equal to 1. (Note we use integer division here)

Assume this for loop run x times, we have, $2^x = n$;
Then what is x?

$x = \log_2 n$

Time cost:

$T(n) = \log_2 n$

// Because the most common base for the logarithmic function
// in computer science is 2. In fact, this base is so common,
// we typically leave it off when base is 2. $\log n = \log_2 n$
// so this method mystery4() runs in **O(log n)**.

Case Study Four

```
void mystery4(int n) {
   for( int i = 1; i <= n; i *= 2)
      printf(i);
}
```

Assume this for loop run x times, we have, $2^x = n$;
Then what is x?

$x = \log_2 n$

Time cost:

$T(n) = \log_2 n$

// Because the most common base for the logarithm function
// in computer science is 2. In fact, this base is so common,
// we typically leave it off when base is 2. $\log n = \log_2 n$
// so this method mystery4() runs in **O(log n)**.

Take Home Summary

- Several Case Studies
  - Prefix Average Problem
  - Factorial
  - Reverse Array
  - Constant Time Complexity
  - Logarithm Method

Next Class

- Asymptotic Algorithm Analysis using Big-Oh notation
  - Analysis Case Studies
  - Logarithmic, Linear, Quadratic and more