

Unit 1.2 Analysis

Algorithms

EE/NTHU

Mar. 10, 2020

Evaluating an Algorithm

- Some criteria to judge an algorithm
 - Does it do what we want it to do?
 - Does it work correctly according to the original specifications of the task?
 - Is there documentation that describes how to use it and how it works?
 - Are procedures created in such a way that they perform logical sub-functions?
 - Is the code readable?

Definition 1.2.1. Space/Time complexity

The **space complexity** of an algorithm is the amount of memory it needs to run to completion. The **time complexity** of an algorithm is the amount of computer time it needs to run to completion.

- Performance evaluation can be divided into two phases:
 - Performance analysis: a priori estimates,
 - Performance measurement: a posteriori testing.
- Three simple examples for space and time complexities analysis.

Algorithm Examples

Algorithm 1.2.2. Expression

```
// Evaluate an arithmetic expression.
// Input:  $x, y, z$ 
// Output: value of the expression.
1 Algorithm expr( $x, y, z$ )
2 {
3     return  $x + y + y \times z + (x + y - z) / (x + y) + 4.0$ ;
4 }
```

Algorithm 1.2.3. Simple Sum

```
// Simple summation of  $n$ -element array  $A[1 : n]$ .
// Input:  $A[1 : n]$ , int  $n > 0$ 
// Output:  $\sum A[i], 1 \leq i \leq n$ .
1 Algorithm Sum( $A, n$ )
2 {
3      $Sum := 0$ ;
4     for  $i := 1$  to  $n$  do
5          $Sum := Sum + A[i]$ ;
6     return  $Sum$ ;
7 }
```

Space Complexity

Algorithm 1.2.4. Recursive Sum

```
// Recursive summation of  $n$ -element array  $A[1 : n]$ .
// Input:  $A[1 : n]$ , int  $n > 0$ 
// Output:  $\sum A[i], 1 \leq i \leq n$ .
1 Algorithm RSum( $A, n$ )
2 {
3     if ( $n \leq 0$ ) then return 0; // Termination check.
4     else return  $A[n] + \text{RSum}(A, n - 1)$ ;
5 }
```

- The memory space needed for the preceding algorithms consists two parts:
 - A **fixed part**, c , that is independent of the size of the problem.
 - Function instructions, constants, simple variables (such as indexing variables).
 - The **variable part**, S_P , that depends on the particular problem.
 - Space for the referenced variables, recursion stack space, etc.
 - The **total space** $S(P)$ for an algorithm P is

$$S(P) = c + S_P(\text{instance characteristic}). \quad (1.2.1)$$

where c is a constant.

- For Algorithm **Expression** the memory space needed are for variables x, y, z , and the result. Thus, no memory is needed that is specific to the instance of the problem, i.e., $S_P(\text{instance characteristic}) = 0$.
- For Algorithm **Sum**, $S_{\text{Sum}}(n) = (n + 3)$.
 - n for array A , and one for each variable: n, i and Sum .
- For Algorithm **RSum**, $S_{\text{RSum}}(n) = 3(n + 1)$.
 - Each recursive call needs to store formal parameters, local variables, and return address.
 - For this problem, it needs to store pointer to A, n and the return address. (assume it takes 3 words)
 - The number of recursive calls is $n + 1$. Thus, total memory space needed is at least $3(n + 1)$.

Time Complexity

- The **time complexity** $T(P)$ of an algorithm is the time required to execute an algorithm.
 - In a general sense, the **compile time** should be included. But, the compile time does not depend on the size of the problem and, thus, is not the focus of the analysis.
 - The execution time should include all operations. Yet, this would make the analysis difficult.
- The time complexity is simplified to count the number of **program steps** when the algorithm execute, t_P .
 - In a loose sense, a program step is an expression.
- As in the following example, one can add an variable **count** to the algorithm **Sum** to count the number of program steps.
- From the example, the number of program steps for an array with n elements, the total number of program steps executed is $2n + 3$. Thus $t_{\text{Sum}} = 2n + 3$.

Time Complexity, II

Algorithm 1.2.5. Sum – Program Step Counting

```
// Modified version to count the number of steps.
// Input:  $A[1 : n]$ ,  $\text{int } n > 0$ 
// Output:  $\sum A[i]$ ,  $1 \leq i \leq n$ .
1 Algorithm Sum( $A, n$ ) // count is a global variable with initial value of 0.
2 {
3      $\text{Sum} := 0$ ;
4      $\text{count} := \text{count} + 1$ ; // for assignment
5     for  $i := 1$  to  $n$  do {
6          $\text{count} := \text{count} + 1$ ; // for loop control
7          $\text{Sum} := \text{Sum} + A[i]$ ;
8          $\text{count} := \text{count} + 1$ ; // for assignment
9     }
10     $\text{count} := \text{count} + 1$ ; // for loop termination
11     $\text{count} := \text{count} + 1$ ; // for return
12    return  $\text{Sum}$ ;
13 }
```

- Same algorithm as Algorithm (1.2.3) with lines 4, 6, 8, 10, 11 added
- After execution, global variable *count* has the number of program steps executed.

Time Complexity, III

- The RSum algorithm can also be modified to count the number of program steps as the following page.
- The number of program steps for an array A with n , $n > 0$, elements is

$$t_{\text{RSum}}(n) = 2 + t_{\text{RSum}}(n - 1)$$

- Including the case of $n = 0$, we have the following recurrence relationship:

$$t_{\text{RSum}}(n) = \begin{cases} 2 & \text{if } n = 0, \\ 2 + t_{\text{RSum}}(n - 1) & \text{if } n > 0. \end{cases}$$

- This recursive formula can expanded for $n > 0$ as

$$\begin{aligned} t_{\text{RSum}}(n) &= 2 + t_{\text{RSum}}(n - 1) \\ &= 2 + 2 + t_{\text{RSum}}(n - 2) \\ &\vdots \\ &= 2n + t_{\text{RSum}}(0) \\ &= 2n + 2 \end{aligned}$$

- Thus, Algorithms sum (1.2.3) and Rsum (1.2.4) have very similar time complexities.

Algorithm 1.2.6. RSum – Program Step Counting

```
// Modified version to count the number of steps.
// Input:  $A[1 : n]$ , int  $n > 0$ 
// Output:  $\sum A[i]$ ,  $1 \leq i \leq n$ .
1 Algorithm RSum( $A, n$ ) // count is a global variable with initial value of 0.
2 {
3     count := count + 1; // for if statement
4     if ( $n \leq 0$ ) then {
5         count := count + 1; // for return statement
6         return 0;
7     }
8     else {
9         count := count + 1; // for the expression and return statements
10        return  $A[n] + \text{RSum}(A, n - 1)$ ;
11    }
12 }
```

- This algorithm is the same as Algorithm (1.2.4) with lines 3, 5, 9 added.

Time Complexity, V

Definition 1.2.7. Input Size

The **input size** of a problem is defined to be the number of words (or the number of elements) needed to describe the instance of the problem.

- For the algorithm **Sum**(A, n) the input size is $(n + 1)$, n for the number of elements of the array, and 1 for the value of n .
- The following algorithm adds two $m \times n$ matrices, A and B , together to form a resulting matrix, C .

Algorithm 1.2.8. Matrix Addition

```
//  $m \times n$  matrix addition.
// Input:  $m \times n$  matrices  $A, B$ , int  $m, n > 0$ 
// Output:  $m \times n$  matrix  $C = A + B$ .
1 Algorithm MAdd( $A, B, C, m, n$ )
2 {
3     for  $i := 1$  to  $m$  do
4         for  $j := 1$  to  $n$  do
5              $C[i, j] := A[i, j] + B[i, j]$ ;
6 }
```

Time Complexity, VI

- Adding **count** to count the number of program steps as the following.


Algorithm 1.2.9. Matrix Addition – Counting Steps

```
// Modified version of  $m \times n$  matrix addition.
// Input:  $m \times n$  matrices  $A, B$ , int  $m, n > 0$ 
// Output:  $m \times n$  matrix  $C = A + B$ .
1 Algorithm MAdd( $A, B, C, m, n$ ) // count is a global variable with 0 initial value.
2 {
3     for  $i := 1$  to  $m$  do {
4          $count := count + 1$ ; // loop- $i$  control
5         for  $j := 1$  to  $n$  do {
6              $count := count + 1$ ; // loop- $j$  control
7              $C[i, j] := A[i, j] + B[i, j]$ ;
8              $count := count + 1$ ; // element addition
9         }
10         $count := count + 1$ ; // loop- $j$  termination
11    }
12     $count := count + 1$ ; // loop- $i$  termination
13 }
```

- Time complexity is $2mn + 2m + 1$
- Input size is $2mn + 2$

Time Complexity – Table Approach

- An alternative approach to find algorithm complexity is the table approach
- For example



Statement	s/e	freq.	Total steps
1 Algorithm Sum(A, n)	0	—	0
2 {	0	—	0
3 $Sum := 0$;	1	1	1
4 for $i := 1$ to n do	1	$n + 1$	$n + 1$
5 $Sum := Sum + A[i]$;	1	n	n
6 return Sum ;	1	1	1
7 }	0	—	0
Total			$2n + 3$

where **s/e** is step per execution,
freq. is the frequency of execution.

- Algorithm Sum(A, n) has the time complexity of $2n + 3$.

Table Approach, II

- RSum example

Statement	s/e	frequency		Total steps	
		$n = 0$	$n > 0$	$n = 0$	$n > 0$
1 Algorithm RSum (A, n)	0	—	—	0	0
2 {	0	—	—	0	0
3 if ($n \leq 0$) then	1	1	1	1	1
4 return 0;	1	1	0	1	0
5 else return					
6 $A[n] + \text{RSum}(A, n - 1);$	$1 + x$	0	1	0	$1 + x$
7 }	0	—	—	0	0
Total				2	$2 + x$

$$x = t_{\text{RSum}}(n - 1)$$

- Thus,

$$t_{\text{RSum}}(n) = \begin{cases} 2 & \text{if } n = 0, \\ 2 + t_{\text{RSum}}(n - 1) & \text{if } n > 0. \end{cases}$$

Table Approach, III

- MAdd example

Statement	s/e	freq.	total steps
1 Algorithm MAdd (A, B, C, m, n)	0	—	0
2 {	0	—	0
3 for $i := 1$ to m do	1	$m + 1$	$m + 1$
4 for $j := 1$ to n do	1	$m(n + 1)$	$mn + m$
5 $C[i, j] := A[i, j] + B[i, j];$	1	mn	mn
6 }	0	—	0
Total			$2mn + 2m + 1$

- Thus, $t_{\text{MAdd}}(n) = 2mn + 2m + 1$.

Fibonacci Number

- Fibonacci number is defined as

$$f_0 = 0, \quad f_1 = 1, \quad f_n = f_{n-1} + f_{n-2}, \quad n \geq 2. \quad (1.2.2)$$

- The following algorithm calculates f_n using iterative approach.

Algorithm 1.2.10. Fibonacci

```
// Compute the  $n$ -th Fibonacci number.
// Input: int  $n \geq 0$ 
// Output:  $f_n$ .
1 Algorithm Fibonacci( $n$ )
2 {
3     if ( $n \leq 1$ ) then return  $n$ ; //  $f_0$  or  $f_1$ , just return  $n$ .
4     else {
5          $fim2 := 0$ ;  $fim1 := 1$ ; //  $fim2 = f_{i-2}$ ,  $fim1 = f_{i-1}$ .
6         for  $i := 2$  to  $n$  do {
7              $fi := fim1 + fim2$ ; //  $fi = f_{i-1} + f_{i-2}$ .
8              $fim2 := fim1$ ;  $fim1 := fi$ ; // Update  $f_{i-2}$  and  $f_{i-1}$ .
9         }
10        return  $fi$ ; //  $f_n = f_i$ .
11    }
12 }
```

Fibonacci Number, II

Statement	s/e	frequency		Total steps	
		$n \leq 1$	$n \geq 2$	$n \leq 1$	$n \geq 2$
1 Algorithm Fibonacci (n)	0	—	—	0	0
2 {	0	—	—	0	0
3 if ($n \leq 1$) then	1	1	1	1	1
4 return n ;	1	1	0	1	0
5 else {	0	—	—	0	0
6 $fim2 := 0$; $fim1 := 1$;	2	0	1	0	2
7 for $i := 2$ to n do {	1	0	n	0	n
8 $fi := fim1 + fim2$;	1	0	$n - 1$	0	$n - 1$
9 $fim2 = fim1$; $fim1 = fi$;	2	0	$n - 1$	0	$2n - 2$
10 }	0	—	—	0	0
11 return fi ;	1	0	1	0	1
12 }	0	—	—	0	0
13 }	0	—	—	0	0
Total				2	$4n + 1$

- Thus,

$$t_{\text{Fibonacci}} = \begin{cases} 2, & n \leq 1, \\ 4n + 1, & n \geq 2. \end{cases}$$

- Note that Eq. (1.2.2) can be implemented using a recursive function.
 - However, this recursive function has a much larger time complexity.
 - You are encouraged to try it out.

- The time complexity – the execution time – of an algorithm depends on the input.
 - Thus, it is usually expressed as a function of the input size.
 - It can be expressed as a function of part of the input size.
 - For example, t_{MAdd} as a function of m , number of rows, only.
 - If such complexity is of interest to a user.
- In evaluating the time complexity of an algorithm, the number of steps is not well defined.
 - It can be a simple comparison, an addition, a multiplication, or even a complex expression.
 - Thus, the exact number is not very important.
 - The growth of the time complexity as the input size grows is usually of more interest.
- The asymptotic complexity will be studied more later.

Dynamic Store Algorithm

- In **C**, array size is fixed. To handle data without prior knowledge of its size, dynamically allocated array should be used.

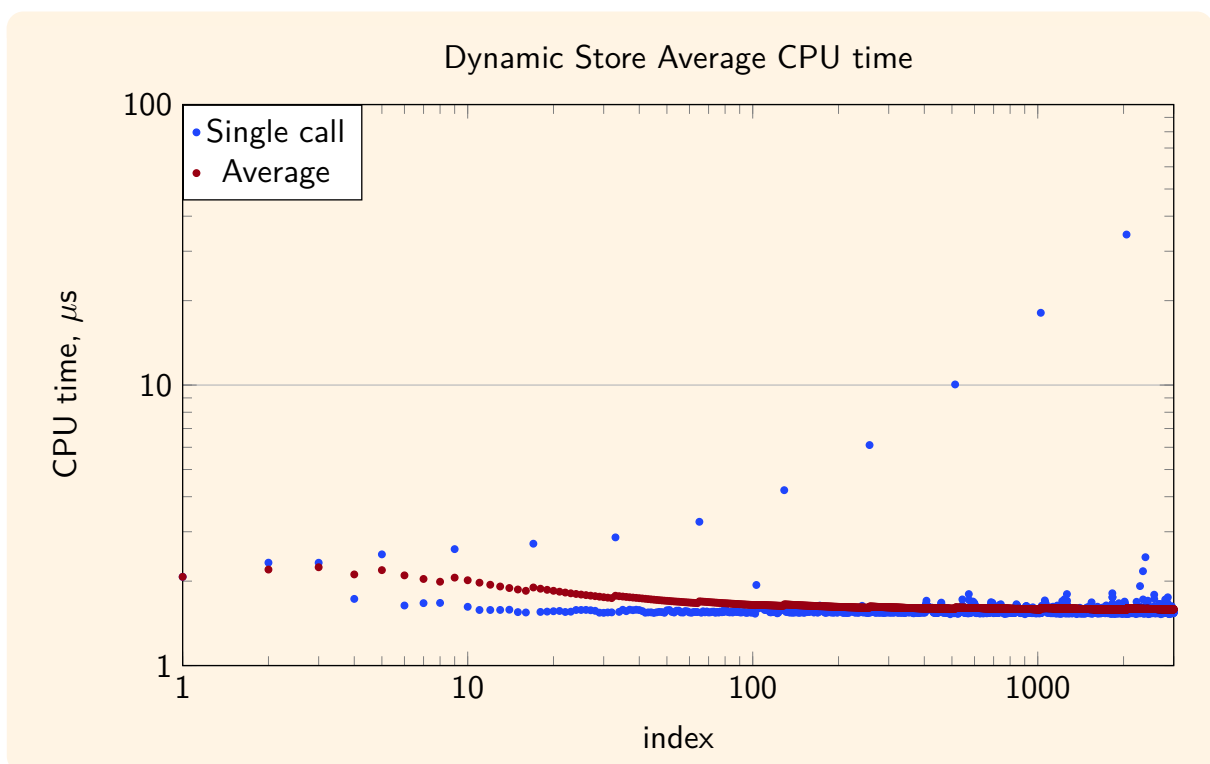
Algorithm 1.2.11. Dynamic Store

```
// Store item into a dynamic array A of size.
// Input: A[1 : size], item, int size and index
// Output: A[index] := item.
1 Algorithm Dynamic_Store(A, size, index, item)
2 {
3     if (size = 0) then { // Initial call.
4         size := 1; A := malloc(size × sizeof(typeA)); // Allocate A.
5     }
6     else if (index > size) then { // Array A is full. Double A.
7         size := 2 × size;
8         B := malloc(size × sizeof(typeA));
9         for i := 1 to index - 1 do B[i] := A[i]; // Copy old data.
10        free(A);
11        A := B; // Pointer assignment.
12    }
13    A[index] := item; // Store into array A.
14    index := index + 1;
15 }
```

Dynamic Store Analysis

- All function parameters are assumed to be called by reference.
- Before the first call to `Dynamic_Store` algorithm, variable *size* should be initialized to 0 and *index* to 1.
- When the algorithm is called, one array storage operation is needed most of the time.
 - In this case, the complexity is $\Theta(1)$.
- However, when $index = 2^k + 1$, $k = 0, 1, 2, \dots$, then $2^k + 1$ array storage operations are needed.
 - Let $n = index$, in this case, n operations are needed.
 - The complexity is $\Theta(n)$.
- Overall complexity is $\mathcal{O}(n)$.
- Since `Dynamic_Store` must be used in sequence, the worst-case complexity overestimates the real complexity.
- **Amortized analysis** should be used for tighter bound.
Three methods available:
 - **Aggregate analysis**
 - **Accounting method**
 - **Potential method**

Dynamic Store CPU Time



- `Dynamic_Store` CPU time is most negligible except when $index = 2^k + 1$.
- However, the average CPU time is completely negligible.

Aggregate Analysis

- The **aggregate analysis** performs the algorithm n times to get $T(n)$ operations, then the average performance of the algorithm is then $T(n)/n$.
- For the **Dynamic_Store**($A, size, index, item$) algorithm the cost of $index = i$, c_i is

$$c_i = \begin{cases} i & \text{if } i = 2^k + 1, k \in \mathbb{N}, \\ 1 & \text{otherwise.} \end{cases} \quad (1.2.3)$$

<i>index</i>	1	2	3	4	5	6	7	8	9
<i>size</i>	1	2	4	4	8	8	8	8	16
c_i	1	2	3	1	5	1	1	1	9
$\sum c_i$	1	3	6	7	12	13	14	15	24

- Total cost for n **Dynamic_Store** calls is

$$T(n) = \sum_{i=1}^n c_i \leq n + \sum_{j=1}^{\lfloor \lg n \rfloor} 2^j < n + 2n = 3n. \quad (1.2.4)$$

- Thus, the amortized cost of a single call is $T(n)/n = 3$.
- The amortized complexity of the algorithm is $\mathcal{O}(1)$.

The Accounting Method

- The amortized analysis performs a sequence of n calls of the algorithm to find the average cost.
- The **actual cost** c_i of the algorithm may vary for different instance i .
- The **amortized cost** \hat{c}_i can be anything but to approach the actual cost over n calls, the following relationship must hold for all $n > 0$.

$$\sum_{i=1}^n \hat{c}_i \geq \sum_{i=1}^n c_i. \quad (1.2.5)$$

- The accounting method is then to select a amortized cost \hat{c}_i and show that Eq. (1.2.5) holds.

- The smaller $\left(\sum_{i=1}^n \hat{c}_i - \sum_{i=1}^n c_i \right)$ the more accurate amortized cost is.

The Accounting Method, II

- For the `Dynamic_Store` algorithm example
- Choose $\hat{c}_i = 3$, we have

<i>index</i>	1	2	3	4	5	6	7	8	9
<i>size</i>	1	2	4	4	8	8	8	8	16
c_i	1	2	3	1	5	1	1	1	9
$\sum c_i$	1	3	6	7	12	13	14	15	24
\hat{c}_i	3	3	3	3	3	3	3	3	3
$\sum \hat{c}_i$	3	6	9	12	15	18	21	24	27
$\sum \hat{c}_i - \sum c_i$	2	3	3	5	3	5	7	9	3

- When $\sum \hat{c}_i - \sum c_i > 0$, we have net credits for future operations.
- It can be shown that $\sum_{i=1}^n \hat{c}_i - \sum_{i=1}^n c_i \geq 3$ for all $n \geq 2$.
- Thus, the amortize cost per operation is 3 and the amortized complexity is $\mathcal{O}(1)$.

The Potential Method

- The potential method associates a non-negative **potential function**, Φ_i , with the i -th operation of the algorithm such that

$$\hat{c}_i = c_i + \Phi_i - \Phi_{i-1} \quad (1.2.6)$$

The amortized cost at the i -th operation is the actual cost plus the potential difference between those two operation.

- The potential function represents the energy barrier for each operation.
- Thus,

$$\begin{aligned} \sum_{i=1}^n \hat{c}_i &= \sum_{i=1}^n (c_i + \Phi_i - \Phi_{i-1}) \\ &= \sum_{i=1}^n c_i + \Phi_n - \Phi_{n-1} + \Phi_{n-1} - \Phi_{n-2} \cdots + \Phi_1 - \Phi_0 \\ &= \sum_{i=1}^n c_i + \Phi_n - \Phi_0 \end{aligned} \quad (1.2.7)$$

The Potential Method, II

- Note that Eq. (1.2.5) still needs to be satisfied.
- Thus,

$$\Phi_n \geq \Phi_0, \quad \text{for all } n \geq 1. \quad (1.2.8)$$

where Φ_0 can be chosen arbitrarily, and is usually set to be 0.

- Again, the average amortized cost represents the amortized complexity of the algorithm.
- Take the **Dynamic_Store** algorithm as an example, note that $index > size/2$, thus we can choose the following potential function.

$$\Phi_i = 2i - size_i, \quad (1.2.9)$$

where $i = index$ and $\Phi_0 = 0$.

<i>index</i>	1	2	3	4	5	6	7	8	9
<i>size</i>	1	2	4	4	8	8	8	8	16
c_i	1	2	3	1	5	1	1	1	9
Φ_i	1	2	2	4	2	4	6	8	2
\hat{c}_i	2	3	3	3	3	3	3	3	3

- Note that $\Phi_i \geq 0$.

The Potential Method, III

- In the case that $index \leq size$ no **malloc** is needed and $size_i = size_{i-1}$.

$$\begin{aligned} \hat{c}_i &= c_i + 2i - size_i - 2(i-1) + size_{i-1} \\ &= 1 + 2i - 2i + 2 = 3. \end{aligned} \quad (1.2.10)$$

Note that c_i is given in Eq. (1.2.3).

- In the case that $index > size$ when calling **Dynamic_Store** we have $size_i = 2 \times size_{i-1} = 2(i-1)$.

$$\begin{aligned} \hat{c}_i &= c_i + 2i - size_i - 2(i-1) + size_{i-1} \\ &= i + 2i - 2i + 2 - 2i + 2 + i - 1 = 3. \end{aligned} \quad (1.2.11)$$

- Thus, we have the amortized cost per operation is $\hat{c}_i = 3$.
- The amortized complexity of the algorithm is $\mathcal{O}(1)$.

Binary Counter

- The m -bit incrementing binary counter algorithm is shown below.

Algorithm 1.2.12.

```
// Increment  $m$ -bit binary array  $D[m-1:0]$ .  
// Input: binary array  $D[m-1:0]$ , int  $m > 0$   
// Output:  $D = D + 1$ .  
1 Algorithm BinCount( $D, m$ )  
2 {  
3      $i := 0$ ; // Loop index  
4     while ( $i < m$  and  $D[i] = 1$ ) do { // Stop for smallest  $i$ ,  $D[i] = 0$   
5          $D[i] := 0$ ; //  $D[i] = 1$ , set it to 0  
6          $i := i + 1$ ; // next  $i$   
7     }  
8     if ( $i < m$ ) then  $D[i] := 1$ ; //  $D[i]$  was 0, set to 1.  
9 }
```

- In this algorithm, the **while** loop on lines 4-7 determines the cost of the operation, but it is not a constant.
- Worst-case complexity is $\mathcal{O}(m)$ due to the **while** loop on lines 4-7.
- How about the average-case complexity?

Binary Counter – Example

- Example of BinCount(D, m) partial execution result with $m = 5$ is shown below (Assuming $D[m-1:0]$ are all 0's initially.)

$D[4]$	$D[3]$	$D[2]$	$D[1]$	$D[0]$	c_i	$\sum c_i$
0	0	0	0	1	1	1
0	0	0	1	0	2	3
0	0	0	1	1	1	4
0	0	1	0	0	3	7
0	0	1	0	1	1	8
0	0	1	1	0	2	10
0	0	1	1	1	1	11
0	1	0	0	0	4	15
0	1	0	0	1	1	16
0	1	0	1	0	2	18
0	1	0	1	1	1	19
0	1	1	0	0	3	22
0	1	1	0	1	1	23
0	1	1	1	0	2	25
0	1	1	1	1	1	26
1	0	0	0	0	5	31

Binary Counter – Aggregate Analysis

- Let the number of bits that change states be the cost of operation, c_i .
- The aggregate analysis execute the algorithm n times to find the total cost of operation and then the average can be found.
- Note that bit $D[0]$ changes state on every call.
- Bit $D[1]$ changes state every other time.
- Bit $D[2]$ changes state every fourth time.
- Hence, we have

$$\sum_{i=1}^n c_i = n + n/2 + n/4 + \cdots + n/2^m < 2n. \quad (1.2.12)$$

- Thus, the total amortized cost is $T(n) = \mathcal{O}(n)$
- And the amortized cost per operation is $T(n)/n = \mathcal{O}(1)$.

Binary Counter – Accounting Method

- In accounting method, we need find \hat{c}_i that satisfies Eq. (1.2.5).

$D[4]$	$D[3]$	$D[2]$	$D[1]$	$D[0]$	c_i	$\sum c_i$	\hat{c}_i	$\sum \hat{c}_i$
0	0	0	0	1	1	1	2	2
0	0	0	1	0	2	3	2	4
0	0	0	1	1	1	4	2	6
0	0	1	0	0	3	7	2	8
0	0	1	0	1	1	8	2	10
0	0	1	1	0	2	10	2	12
0	0	1	1	1	1	11	2	14
0	1	0	0	0	4	15	2	16
0	1	0	0	1	1	16	2	18
0	1	0	1	0	2	18	2	20
0	1	0	1	1	1	19	2	22
0	1	1	0	0	3	22	2	24
0	1	1	0	1	1	23	2	26
0	1	1	1	0	2	25	2	28
0	1	1	1	1	1	26	2	30
1	0	0	0	0	5	31	2	32

- $\hat{c}_i = 2$ is a choice and the amortized cost per operation is $\mathcal{O}(1)$.

Binary Counter – Potential Method

- In potential method, we need to find the potential function that satisfies Eqs. (1.2.7) and (1.2.8), then the amortized cost can be found using Eq. (1.2.9).
- Define the potential function as

$$\Phi_i = \sum_{j=0}^{m-1} D[j]. \quad (1.2.13)$$

That is Φ_i is the number of set bits ($D[i] = 1$).

- Let r_i be the number of bits reset to 0 for the i -th operation, then

$$c_i = r_i + 1. \quad (1.2.14)$$

- Note that r_i is simply the number iteration for the **while** loop on **lines 5-8** of Algorithm (1.2.12), and the extra 1 comes from **line 9**.
- Thus for the i -th operation,

$$\Phi_i = \Phi_{i-1} - r_i + 1. \quad (1.2.15)$$

- And

$$\hat{c}_i = c_i + \Phi_i - \Phi_{i-1} = r_i + 1 + \Phi_{i-1} - r_i + 1 - \Phi_{i-1} = 2. \quad (1.2.16)$$

- Thus, the amortized cost per operation is $\mathcal{O}(1)$.

Binary Counter – Potential Method, II

- Potential method in 5-bit binary counter example.

$D[4]$	$D[3]$	$D[2]$	$D[1]$	$D[0]$	c_i	Φ_i	\hat{c}_i
0	0	0	0	1	1	1	2
0	0	0	1	0	2	1	2
0	0	0	1	1	1	2	2
0	0	1	0	0	3	1	2
0	0	1	0	1	1	2	2
0	0	1	1	0	2	2	2
0	0	1	1	1	1	3	2
0	1	0	0	0	4	1	2
0	1	0	0	1	1	2	2
0	1	0	1	0	2	2	2
0	1	0	1	1	1	3	2
0	1	1	0	0	3	2	2
0	1	1	0	1	1	3	2
0	1	1	1	0	2	3	2
0	1	1	1	1	1	4	2
1	0	0	0	0	5	1	2

- In amortized analysis a sequence of n operations are performed to find the worst-case total operations.
- The time complexity of a single operation is then the total operation cost divided by the number of operation, n .
- Three methods are available:
 - Aggregate analysis,
 - More systematic.
 - Accounting method,
 - Usually the amortized cost is assumed and proven to be correct.
 - Potential method,
 - Need to find the potential function.
 - A tool to prove the amortized cost.

Summary

- Space and time complexities
- Algorithm examples
- Time complexity
 - Counting number of steps
 - Table approach.
- Amortized analysis
 - Aggregate analysis
 - Accounting method
 - Potential method