Data Structures & Algorithms (COMP2113)

Lecture # 14

Algorithm Analysis | Part05

Calculating Running Time

Course Instructor

Dr. Aftab Akram

PhD CS

Assistant Professor

Department of Information Sciences, Division of Science & Technology

University of Education, Lahore

Constant Running Time

Example 3.9 We begin with an analysis of a simple assignment to an integer variable.

```
a = b;
```

Because the assignment statement takes constant time, it is $\Theta(1)$.

Running Time of `for` loop

Example 3.10 Consider a simple **`for`** loop.

```
sum = 0;
for (i=1; i<=n; i++)
   sum += n;
```

The first line is $\Theta(1)$. The **`for`** loop is repeated n times. The third line takes constant time so, by simplifying rule (4) of Section 3.4.4, the total cost for executing the two lines making up the **`for`** loop is $\Theta(n)$. By rule (3), the cost of the entire code fragment is also $\Theta(n)$.

Running Time of Several **for** loops

Example 3.11 We now analyze a code fragment with several **for** loops, some of which are nested.

```
sum = 0;
for (i=1; i<=n; i++)       // First for loop
   for (j=1; j<=i; j++)    //   is a double loop
      sum++;
for (k=0; k<n; k++)        // Second for loop
   A[k] = k;
```

This code fragment has three separate statements: the first assignment statement and the two **for** loops. Again the assignment statement takes constant time; call it c_1. The second **for** loop is just like the one in Example 3.10 and takes $c_2 n = \Theta(n)$ time.

Running Time of Several **for** loops

The first **`for`** loop is a double loop and requires a special technique. We work from the inside of the loop outward. The expression **`sum++`** requires constant time; call it c_3. Because the inner **`for`** loop is executed i times, by simplifying rule (4) it has cost $c_3 i$. The outer **`for`** loop is executed n times, but each time the cost of the inner loop is different because it costs $c_3 i$ with i changing each time. You should see that for the first execution of the outer loop, i is 1. For the second execution of the outer loop, i is 2. Each time through the outer loop, i becomes one greater, until the last time through the loop when $i = n$. Thus, the total cost of the loop is c_3 times the sum of the integers 1 through n. From Equation 2.1, we know that

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2},$$

which is $\Theta(n^2)$. By simplifying rule (3), $\Theta(c_1 + c_2 n + c_3 n^2)$ is simply $\Theta(n^2)$.

Nested `for` loops

Example 3.12 Compare the asymptotic analysis for the following two code fragments:

```
sum1 = 0;
for (i=1; i<=n; i++)      // First double loop
   for (j=1; j<=n; j++)   //    do n times
      sum1++;

sum2 = 0;
for (i=1; i<=n; i++)      // Second double loop
   for (j=1; j<=i; j++)   //    do i times
      sum2++;
```

Nested **`for`** loops

In the first double loop, the inner **`for`** loop always executes n times. Because the outer loop executes n times, it should be obvious that the statement **`sum1++`** is executed precisely n^2 times. The second loop is similar to the one analyzed in the previous example, with cost $\sum_{j=1}^{n} j$. This is approximately $\frac{1}{2}n^2$. Thus, both double loops cost $\Theta(n^2)$, though the second requires about half the time of the first.

Nested `for` loops-Another Example

Example 3.13 Not all doubly nested `for` loops are $\Theta(n^2)$. The following pair of nested loops illustrates this fact.

```
sum1 = 0;
for (k=1; k<=n; k*=2)     // Do log n times
   for (j=1; j<=n; j++)   // Do n times
      sum1++;

sum2 = 0;
for (k=1; k<=n; k*=2)     // Do log n times
   for (j=1; j<=k; j++)   // Do k times
      sum2++;
```

Nested **`for`** loops-Another Example

When analyzing these two code fragments, we will assume that n is a power of two. The first code fragment has its outer **`for`** loop executed $\log n + 1$ times because on each iteration k is multiplied by two until it reaches n. Because the inner loop always executes n times, the total cost for the first code fragment can be expressed as $\sum_{i=0}^{\log n} n$. Note that a variable substitution takes place here to create the summation, with $k = 2^i$. From Equation 2.3, the solution for this summation is $\Theta(n \log n)$. In the second code fragment, the outer loop is also executed $\log n + 1$ times. The inner loop has cost k, which doubles each time. The summation can be expressed as $\sum_{i=0}^{\log n} 2^i$ where n is assumed to be a power of two and again $k = 2^i$. From Equation 2.8, we know that this summation is simply $\Theta(n)$.

Running Time of **`while`** loop

- **`While`** loops are analyzed in a manner similar to **`for`** loops.
- The cost of an **`if`** statement in the worst case is the greater of the costs for the **`then`** and **`else`** clauses.
- This is also true for the average case, assuming that the size of n does not affect the probability of executing one of the clauses (which is usually, but not necessarily, true).
- For **`switch`** statements, the worst-case cost is that of the most expensive branch.
- For subroutine calls, simply add the cost of executing the subroutine.

Binary Search

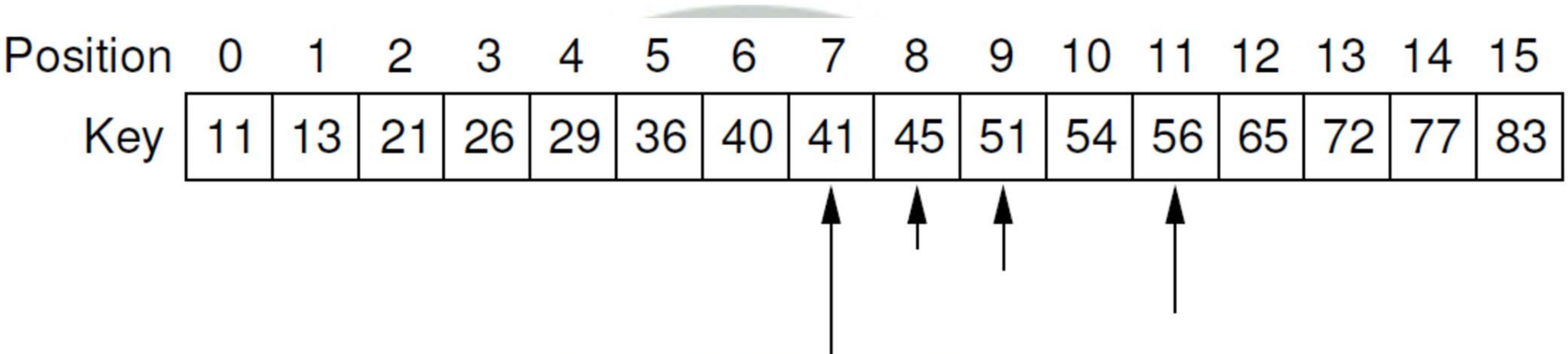

Figure 3.4 An illustration of binary search on a sorted array of 16 positions. Consider a search for the position with value $K = 45$. Binary search first checks the value at position 7. Because $41 < K$, the desired value cannot appear in any position below 7 in the array. Next, binary search checks the value at position 11. Because $56 > K$, the desired value (if it exists) must be between positions 7 and 11. Position 9 is checked next. Again, its value is too great. The final search is at position 8, which contains the desired value. Thus, function **`binary`** returns position 8. Alternatively, if K were 44, then the same series of record accesses would be made. After checking position 8, **`binary`** would return a value of n, indicating that the search is unsuccessful.

Estimating Running Time for Binary Search

```
// Return the position of an element in sorted array "A" of
// size "n" with value "K".  If "K" is not in "A", return
// the value "n".
int binary(int A[], int n, int K) {
  int l = -1;
  int r = n;             // l and r are beyond array bounds
  while (l+1 != r) {  // Stop when l and r meet
    int i = (l+r)/2;  // Check middle of remaining subarray
    if (K < A[i]) r = i;     // In left half
    if (K == A[i]) return i; // Found it
    if (K > A[i]) l = i;     // In right half
  }
  return n; // Search value not in A
}
```

Estimating Running Time for Binary Search

To find the cost of this algorithm in the worst case, we can model the running time as a recurrence and then find the closed-form solution. Each recursive call to **`binary`** cuts the size of the array approximately in half, so we can model the worst-case cost as follows, assuming for simplicity that n is a power of two.

$$\mathbf{T}(n) = \mathbf{T}(n/2) + 1 \text{ for } n > 1; \quad \mathbf{T}(1) = 1.$$

If we expand the recurrence, we find that we can do so only $\log n$ times before we reach the base case, and each expansion adds one to the cost. Thus, the closed-form solution for the recurrence is $\mathbf{T}(n) = \log n$.

Next Lecture

- In next lecture, we discuss topics like Analyzing Problems, some common misunderstanding.