Lecture 12: Runtime

Introduction

When designing data structures and algorithms, it is crucial to write efficient functions, particularly in terms of time complexity. The *runtime* of a function refers to the number of computational steps required to complete a task. Efficient functions minimize these steps.

How Computers Execute Code

To understand runtime, it's important to recognize how computers execute code. Every statement in a program is translated into *machine language commands*, which are fundamental instructions a computer understands. The number of machine instructions required to execute a statement determines its execution time, referred to as the *processing cost* of the statement. Therefore, a function's runtime is the sum of the processing cost of each statement multiplied by the number of times that statement executes during the function's execution.

Summation Properties

Summations (or *series*) help express the total runtime of an algorithm. A summation is written as:

$$\sum_{i=a}^{b} f(i) = f(a) + f(a+1) + \cdots + f(b)$$

where a and b define the summation's range and $f(i)$ is the function being summed. Useful summation properties include:

- **Splitting a Summation**

  $$\sum_{i=a}^{b} f(i) = \sum_{i=a}^{c} f(i) + \sum_{i=c+1}^{b} f(i)$$

  where $a \leq c < b$.

- **Sum/Difference & Scalar Products**

  $$\sum_{i=a}^{b} c(f(i) \pm g(i)) = c\left(\sum_{i=a}^{b} f(i) \pm \sum_{i=a}^{b} g(i)\right)$$

  where c is a number.

- **Sum of Ones**

  $$\sum_{i=a}^{b} 1 = (b - a + 1)$$

- **Consecutive Integer Sum**

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

  where n is a positive integer.

These properties assist in analyzing the runtime of loops and recursive functions.

Runtime Function & Factor

A *runtime function* expresses an algorithm's execution time as a function of the input size (or *runtime factor*)—the specific aspect of the input that influences statement execution. It is computed as:

$$T(n) = \sum_{i=1}^{m} c_i \cdot t_i$$

where:

- m is the number of unique statements in the function,
- c_i is the processing cost of the ith statement,
- t_i is the number of times the ith statement executes,
- n represents the size of the input data.

Since exact processing costs vary across systems, abstract cost values are used.

Example:

```
ulg C(ulg x)
{
  ulg n = 0;

  while(x > 0)
  {
    x = x / 10;
    n += 1;
  }
  return n;
}
```

```
bool P(ulg n)
{
  for(ulg i = 2;i*i <= n;i += 1)
  {
    if(n % i == 0)
    {
      return false;
    }
  }
  return (n > 1);
}
```

```
void R(Array<int>& dt,ulg n,ulg m)
{
  while(n < m)
  {
    int t = dt[n];
    dt[n] = dt[m];
    dt[m] = t;
    n += 1;
    m -= 1;
  }
}
```

where '`ulg`' is an abstract data type for unsigned long and *Array* is a container class for an array. The runtime analysis of each algorithm is:

- `C()` - The loop executes for each digit of x; hence, the runtime factor is the number of digits in the parameter.
- `P()` - The loop checks if n is divisible by integers at most its square root; hence, the runtime factor is the parameter's value.
- `R()` - The loop swaps values between n and m; hence, the runtime factor is the difference of the integer parameters.

Worst-Case, Average-Case, & Best-Case Analysis

When analyzing runtime, different scenarios must be considered since the input varies the statement executions. The scenarios are:

- **Best-case**: the function executes the minimum number of steps.
- **Average-case**: the function executes an average number of steps.
- **Worst-case**: the function executes the maximum number of steps.

The worst-case scenario is often used because it provides an upper bound on runtime.

Example:
For each algorithm, its worst-case scenario is:

- `C()` - All cases are the same.
- `P()` - The worst-case scenario occurs when the parameter is prime.
- `R()` - The worst-case scenario occurs when integer parameters are the end indices of the array parameter.

Analyzing Function Runtime with a Runtime Table

Constructing a *runtime table*—a table that records the cost and number of executions (time) for each distinct statement in an algorithm—simplifies determining an algorithm's runtime function. The cost of a statement is represented as an abstract value, typically 0 or 1, depending on the aspects of the analyzed algorithm. The time is expressed as either a fixed integer or a function of the algorithm's runtime factor. When the time is an expression involving the runtime factor, the floor and ceiling functions are used to ensure proper calculations.

Example:
The runtime tables for each algorithm is:

`C()`:

line	cost	time
03	c_1	1
05	c_2	$n+1$
07	c_3	n
08	c_4	n
10	c_5	1

`P()`:

line	cost	time
03	c_1	1
03	c_2	$\lfloor\sqrt{n}\rfloor$
05	c_3	$\lfloor\sqrt{n}\rfloor - 1$
07	c_4	0
03	c_5	$\lfloor\sqrt{n}\rfloor - 1$
10	c_6	1

`R()`:

line	cost	time
03	c_1	$\lceil\frac{n}{2}\rceil + 1$
05	c_2	$\lceil\frac{n}{2}\rceil$
06	c_3	$\lceil\frac{n}{2}\rceil$
07	c_4	$\lceil\frac{n}{2}\rceil$
08	c_5	$\lceil\frac{n}{2}\rceil$
09	c_6	$\lceil\frac{n}{2}\rceil$

- Algorithm `C()` loop's condition (line 05) is evaluated once for each digit of x, plus once more to exit, resulting in $n+1$ evaluations; meanwhile, its body executes n times—for each true evaluation of the condition. Therefore, the runtime function is
$$T(n) = 3n + 3$$
if all costs are 1.
- Algorithm `P()` loop's condition (line 03) is evaluated once for each integer from 2 to $\lfloor\sqrt{n}\rfloor + 1$, resulting in $\lfloor\sqrt{n}\rfloor$ evaluations. When n is prime (worst-case scenario), the return statement (line 07) never executes; whereas, the rest of body statements run $\lfloor\sqrt{n}\rfloor - 1$ times. Therefore, the runtime function is
$$T(n) = 3\lfloor\sqrt{n}\rfloor$$
if all costs are 1.
- Algorithm `R()` loop's condition (line 03) is evaluated $\lceil\frac{n}{2}\rceil + 1$ times since the endpoints approach each other until they overlap each other 1 step each time; meanwhile, its body statements run $\lceil\frac{n}{2}\rceil$ times. Therefore, the runtime function is
$$T(n) = 6\left\lceil\frac{n}{2}\right\rceil + 1$$
if all costs are 1.

Order of Growth & Big-O Notation

Asymptotic notations are used to compare algorithms by providing a mathematical boundary for their runtime functions. These notations define an upper bound, a lower bound, or both, helping to analyze the efficiency of algorithms as input size grows.

- **Big-Oh**: upper bound on runtime growth denoted $O(f(n))$ which is
$\{g(n) : 0 < g(n) \le cf(n) \text{ whenever } n \ge n_0 \text{ for some } c > 0, n_0 > 0\}$
- **Big-Omega**: lower bound on runtime growth denoted $\Omega(f(n))$ which is
$\{g(n) : 0 < cf(n) \le g(n) \text{ whenever } n \ge n_0 \text{ for some } c > 0, n_0 > 0\}$
- **Big-Theta**: tight bound (both upper and lower bound) denoted $\Theta(f(n))$ which is
$\{g(n) : c_1 f(n) \le g(n) \le c_2 f(n) \text{ whenever } n \ge n_0 \text{ for some } c_2 \ge c_1 > 0, n_0 > 0\}$

Example:

- $O(1)$: constant time (independent of input size).
- $O(logn)$: logarithmic time (e.g., binary search).
- $O(n)$: linear time (e.g., iterating over an array).
- $O(n^2)$: quadratic time (e.g., nested loops).
- $O(2^n)$: exponential time (e.g., brute-force recursive algorithms).

In addition, although a function may have multiple big-oh bounds, the tightest bound is preferred. For instance, $3n + 5 = O(n^2)$ is valid, but $3n + 5 = O(n)$ is tighter and more accurate.