Algorithm Analysis

CSE 2020 Computer Science II

Learning Objectives

- Analyze growth rates of programs
- Describe the best, worst, and average case analysis
- Explain the definition of upper bound of growth rates (Big-Oh notation)
- Calculate Big-Oh of programs
- Compare the complexity of programs based on Big-Oh

Algorithms

- "An algorithm is a sequence of computational steps that transform the input into the output"
- Example: Sorting Problem
  - Input: A sequence of n numbers a1 , a2 , ..., an
  - Output: A permutation (reordering) {a1′ , a2′ , ..., an′} of the input sequence such that a1′ <= a2′ <= …… <=an′
- Example: Searching Problem
  - Input: A sequence of n numbers a1 , a2 , ..., and an element x to be search
  - Output: the index of the element x if present, else, return -1

How good is your algorithm?

- Algorithm analysis means analyzing the execution times of algorithms, or time complexity of algorithms.
- NOT the concrete execution time of an algorithm with specific input.
- Focus on theoretical analysis that is independent of computers, programming languages, and specific input.
- Focus on the growth rate of the running time.
- Asymptotic Analysis
  - the upper bound of growth rate, **Big-Oh**
  - the lower bound of growth rate, Big-Omega
  - same growth rate, Big-Theta

Growth Rate of Running Time

- Growth rate of running time is the time complexity based on the input size n, that is, a function of the input size n, $T(n)$.
- With growth rate, we can see how fast an algorithm’s execution time increases as the input size increases.
    - Preferred – when n increases, the execution time increases slower
- Calculate growth rate – count the number of basic operations

Growth Rate Example 1

- Compute the sum of an array $\sum_{i=0}^{n-1} a[i]$

```
int sum (int a[], int n)
{
    int result = 0;
    for (int i = 0; i < n; i++)
        result = result + a[i];
    return result;
}
```

- $T(n) = t_1 n + \ t_2$
- linear growth rate

Growth Rate Example 2

- Compute geometric series sum $\sum_{i=0}^{n} x^i$

```
long long int geom_sum (int x, int n)
{
    long long int result = 0;
    for (int i = 0; i <= n; i++)
    {
        long long int xpow = 1;
        for (int j = 0; j < i; j++)
            xpow = xpow * x;
        result = result + xpow;
    }
    return result;
}
```

- $T(n) = t_1 n^2 + t_2 n + t_3$
- quadratic growth rate

Growth Rate Example 3

- Compute geometric series sum $\sum_{i=0}^{n} x^i$ with Horner's rule

```
long long int geom_sum (int x, int n)
{
    long long int result = 0;
    for (int i = 0; i <= n; i++)
    {
        result = result * x + 1;
    }
    return result;
}
```

- $T(n) = t_1 n + t_2$
- linear growth rate

Growth Rate Example 4

- find maximum

```
int findMax (int a[], int n)
{
    int max = a[0];
    for ( int i = 1; i < n; i++)
    {
        if ( a[i] > max )
            max = a[i];
    }
    return max;
}
```

- $T(n) = t_1 n + t_2$

Growth Rate Example 5

- Sequential Search

```
int seqtSearch (int a[], int n, int key)
{
    int i = 0;
    while ( i < n )
    {
        if ( a[i] == key ) return i;
        else i++;
    }
    return -1;
}
```

- The key matches a[0], $T(n) = t_1$
- The key is not in a[], $T(n) = t_1 n + t_2$

Best, Worst, and Average Cases

- For the same input size, an algorithm’s execution time may vary, depending on the input.
- Best case analysis means analyzing algorithms based on the input that results in the shortest execution time.
- Worst case analysis means analyzing algorithms based on the input that results in the longest execution time.
- Best and worst case analysis are not representative.
- Average case analysis determine the average execution time among all possible input of the same size, ideal but difficult to perform.
- Normally, use worst case analysis,
  - easy to perform,
  - an algorithm will never be slower than worst case.

Upper Bound, Big-Oh

- Upper bound indicates the upper or highest growth rate that an algorithm can have.
- $T(n) = O(f(n))$ the upper bound of an algorithm growth rate $T(n)$ is $f(n)$
- **Definition of Big-Oh:** $\boldsymbol{T(n) = O(f(n))}$ **if there exist positive constants** $\boldsymbol{c}$ **and** $\boldsymbol{n_0}$ **such that** $\boldsymbol{T(n) \leq c(f(n))}$ **for all** $\boldsymbol{n \geq n_0}$
- $\boldsymbol{T(n)}$ is asymptotically smaller than or equal to $\boldsymbol{f(n)}$

Big-Oh Example

- $T(n) = 5n+7 \;\leq\; 5n+7n = 12\text{n} \;\text{ for n} \geq 1$

  $T(n) \leq cn \text{ for n} \geq 1 \; and \; c = 12$

  $T(n) = \; \text{O}(\text{n})$
- $T(n) = 3n^2+10n+100$

  $\leq 3n^2+10n^2+100n^2 = 113n^2 \text{ for n } \geq 1$

  $T(n) \leq cn^2 \text{ for n} \geq 1 \; and \; c = 113$

  $T(n) = \; \text{O}(n^2)$
- $T(n) = 100n^3 + \; 100n \; + \; 100$

  $\leq 100n^3+100n^3+100n^3 = 300n^3 \text{ for n } \geq 1$

  $T(n) \leq cn^3 \text{ for n} \geq 1 \text{ and c} = 300$

  $T(n) = \; \text{O}(n^3)$

Big-Oh Example (cont.)

- $T(n) = (n^2 + \ 100) \log 5 \ n^5$

$= (n^2 + \ 100) (\log 5 \ + \ logn^5)$

$= (n^2 + \ 100) (\log 5 \ \ + 5\log n)$

$\leq (n^2 + \ 100n^2) (\log n \ \ + 5\log n)$ for $\mathrm{n} \geq 5$

$= 101n^2 \ * 6\log n \ = 606n^2 \ \log n$

$T(n) \leq \ cn^2 \ \log n$ for $\mathrm{n} \geq 5$ and $\mathrm{c} = 606$

$T(n) = \ \mathrm{O}(n^2 \ \log n)$

Big-Oh Example (cont.)

- $T(n) = t_1 n \ + \ t_2 \leq t_1 n \ + \ t_2 n \ = (t_1 + t_2)n \ for \ n \geq 1$

  $T(n) \leq cn$ for n ≥ 1 $and \ c = t_1 + t_2$

  $T(n) = \mathrm{O(n)}$

- $T(n) = t_1 n^2 + \ t_2 n \ + \ t_3 \leq \ t_1 n^2 + \ t_2 n^2 + \ t_3 \ n^2$

  $= (t_1 + t_2 + t_3)n^2 \ \ for \ n \geq 1$

  $T(n) \leq cn^2$ for n ≥ 1 $and \ c = t_1 + t_2 + \ t_3$

  $T(n) = O(n^2)$

Growth Rate to Big-Oh

- $T(n) \rightarrow$ Big-Oh: ignore **constants** and **lower-order terms** in growth rate function
- If an algorithm takes constant running time regardless of the input size n, $T(n) = c$, we say $T(n) = O(1)$, constant time
- We always seek to define the running time of an algorithm with the TIGHTEST possible upper bound.
  - $T(n) = 5n + 7 = O(n^2)$ is not the tightest

Big-Oh of Code Segments

- Single for loop O(n)

```
for (int i = 0; i < n; i++) { } O(n)
```

- Nested for loops O(n^2)

```
for (int i = 0; i < n; i++){
    for (int j = 0; j < n; j++){
        ……
}}
```

- Count the number of iterations in loops and the number of basic operations in each iteration, such as comparison ->T(n) ->Big-Oh

Big-Oh

• $O(1)$	Constant
• $O(logn)$	Logarithmic
• $O(n)$	Linear
• $O(nlogn)$	Log-linear
• $O(n^2)$	Quadratic
• $O(n^3)$	Cubic
• $O(2^n)$	Exponential
• $O(n!)$	Factorial

Comparing Algorithms

- Given two growth rate functions f(n) and g(n), determine if one grows faster than the other.

$$\lim_{n \to \infty} \frac{f(n)}{g(n)}$$

- ∞, then g(n) is in O(f(n)) because f(n) grows faster.
- 0, then f(n) is in O(g(n)) because g(n) grows faster.
- $c \neq 0$, then both grow at the same rate

Binary Search Example

```
int binarySearch(int arr[], int l, int r, int x)
{
    if (r >= l) {
        int mid = l + (r - l) / 2;
        if (arr[mid] > x)
            return binarySearch(arr, l, mid - 1, x);
        else if (arr[mid] < x)
            return binarySearch(arr, mid + 1, r, x);
        else
            return mid;
    }
    return -1;
}
```

Binary Search Example

$$T(n) = T\left(\frac{n}{2}\right) + c$$

$$= T\left(\frac{n}{2*2}\right) + c + c$$

$$= T\left(\frac{n}{2*2*2}\right) + c + c + c$$

$$= T\left(\frac{n}{2^k}\right) + kc$$

assuming $n = 2^k \Rightarrow k = log_2 n$

$$= T(1) + c log_2 n \Rightarrow O(\log n)$$