SCC.121: ALGORITHMS AND COMPLEXITY
Big-O Notation

Emma Wilson, Ibrahim Aref

Today's Lecture

Aim: Look at using big-O notation in practice

Learning objectives:

- To know how to calculate the complexity of example algorithms in big-O notation
- To be able to calculate the complexity of algorithms using big-O notation without counting each operation

Outline

- **Big O of example algorithms discussed in previous lectures**
- Examples – big O without counting the number of operations
- General rules – big O without counting the number of operations

The big O notation in general

- **Examples:**
  - T(n) = $C_1 \times N + C_0$ → O(N)
  - T(n) = $C_2 \times N^2 + C_1 \times N + C_0$ → $O(N^2)$
  - T(n) = $C_3 \times N^3 + C_2 \times N^2 + C_1 \times N + C_0$ → $O(N^3)$
  - T(n) = $C_k \times N^k + C_{k-1} \times N^{k-1} + \cdots + C_1 \times N + C_0$ → $O(N^k)$
- More examples:
  - T(n) = $C_2 \times N + C_1 \log N + C_0$ → O(N) N is dominant term
  - T(n) = $C_2 \times N^{1000} + C_1 2^N + C_0$ → $O(2^N)$ 2^N is dominant term

Case 1: T(N) is Constant

- What is the overall program time?
  - **Best Case:** T(N) = 20
    - **T(N) = Constant** → *O*(1)
  - **Worst Case:** T(N) = 20
    - **T(N) = Constant** → *O*(1)

Case 1: T(N) is Constant

```
double avg5(int theArray[])
{
      int total = 0;

      for (int i=0; i<5; i++)

            total += theArray[i];

      double avg = ((double) total) / 5.0;

      return avg;
}
```

Case 2: T(N) is Linear

- What is the overall program time?
  - **Best Case:** T(N) = **3N+1**
    - **$T(N) = C_1 \times N + C_2 \rightarrow O(N)$**
    - **$C_1 = 3$ and $C_2 = 1$ are constant**
  - **Worst Case:** T(N) = **4N**
    - **$T(N) = C_1 \times N + C_2 \rightarrow O(N)$**
    - **$C_1 = 4$ and $C_2 = 0$ are constant**

Case 2: T(N) is Linear

```
int findMin(int theArray[], int N)
{

        int smallest_i = 0; //Assume smallest value at index 0

        for (int i=1; i<N; i++)

                if (theArray[i] < theArray[smallest_i])

                        smallest_i = i;

        return smallest_i;
}
```

Case 3: T(N) is Logarithmic

- What is the overall program time?
  - **Best Case:** T(N) = $3 \log_2 \mathrm{N} + 3$
    - **T(N) = C_1 x $\log_2$ N + C_2** → $O(\log N)$
    - **C_1 = 3 and C_2 = 3 are constant**
  - **Worst Case:** T(N) = $3 \log_2 \mathrm{N} + 3$
    - **T(N) = C_1 x $\log_2$ N + C_2**
    - **C_1 = 3 and C_2 = 3 are constant** → $O(\log N)$

Case 3: T(N) is Logarithmic

```
int logBaseTwoN(int N)
{
        int count = 0;

        while (N > 1) {

                count++;

                N = N/2;
        }
        return count;
}
```

Case 4: T(N) is Quadratic

- What is the overall program time?
  - **Best Case:** $T(N) = 3N^2+4N+3$
    - **$T(N) = C_1 \times N^2 + C_2 \times N + C_3 \rightarrow O(N^2)$**
    - **$C_1 = 3$ and $C_2 = 4$ and $C_3 = 3$ are constant**
  - **Worst Case:** $T(N) = 3N^2+4N+3$
    - **$T(N) = C_1 \times N^2 + C_2 \times N + C_3 \rightarrow O(N^2)$**
    - **$C_1 = 3$ and $C_2 = 4$ and $C_3 = 3$ are constant**

Case 4: T(N) is Quadratic

```
void quadratic(int N)
{
     int count = 0;

      for (int i=0; i<N; i++)

          for (int j=0; j<N; j++)

              count++;
}
```

Linear Search

- What is the overall program time?
  - **Best Case:** T(N) = 4
    - **T(N) = Constant →** $O(1)$
  - **Worst Case:** T(N) = 3N+3
    - **T(N) = C_1 x N + C_2 →** $O(N)$
    - **C_1 and C_2 are constant**
  - **Average case: T(N) =** $\left(\frac{3}{2}P + 3 - 3\mathrm{P}\right)\mathrm{N} + \left(\frac{5}{2}P + 3 - 3P\right)$
  - **T(N) = C_1 x N + C_2 →** $O(N)$
  - **C_1 and C_2 are constant**

Linear Search

```
int isInArray(int theArray[], int N, int iSearch)
{

	for (int i = 0; i < N; i++)
		if (theArray[i] == iSearch)
			return 1;

	return 0;
}
```

Outline

- Big O of example algorithms discussed in previous lectures
- **Examples – big O without counting the number of operations**
- General rules – big O without counting the number of operations

Example#1

- **Example#1:** What is the worst case time complexity (in the Big O notation) of the following code fragment? O(1)

```
// Here c is a constant

for (int i = 1; i <= c; i++){

    // some O(1) expressions

}
```

Example#2

- **Example#2:** What is the worst case time complexity (in the Big O notation) of the following code fragment? O(max(N,M))

```
int a = 0, b = 0;

for (i = 0; i < N; i++) {
        a = a + rand();
}

for (j = 0; j < M; j++) {
        b = b + rand();
}
```

$T_1(n)$ is O(N)

$T_2(n)$ is O(M)

Example#3

- **Example#3:** What is the worst case time complexity (in the Big O notation) of the following code fragment? O(max(N,N))=O(N)

```
int a = 0, b = 0;

for (i = 0; i < N; i++) {
    a = a + rand();
}

for (j = 0; j < N; j++) {
    b = b + rand();
}
```

$T_1(n)$ is O(N)

$T_2(n)$ is O(N)

Example#4

- **Example#4:** What is the worst case time complexity (in the Big O notation) of the following code fragment if $M \ll N$? O(max(N,M))=O(N)

```
int a = 0, b = 0;

for (i = 0; i < N; i++) {
      a = a + rand();
}

for (j = 0; j < M; j++) {
      b = b + rand();
}
```

$T_1(n)$ is O(N)

$T_2(n)$ is O(M)

Example#5

- **Example#5:** What is the worst case time complexity (in the Big O notation) of the following code fragment?

```
int a = 0;
for (i = 0; i < N; i++) {

    for (j = 0; j < N; j++) {

        a = a + i + j;
    }
}
```

$T_1(n)$ is O(N)

Example#6

- **Example#6:** What is the worst case time complexity (in the Big O notation) of the following code fragment?

```
int a = 0;
for (i = 0; i < N; i++) {

}
```

$T_2(n)$ is O(N)

Example#6

- **Example#6:** What is the worst case time complexity (in the Big O notation) of the following code fragment? $O(N) \times O(N) = O(N^2)$

```
int a = 0;
for (i = 0; i < N; i++) {

    for (j = 0; j < N; j++) {

        a = a + i + j;
    }
}
```

$T_2(n)$ is O(N)

$T_1(n)$ is O(N)

Example Question

- What is the worst case time complexity (in the big O notation) of the following code fragment?

```
for (i = 0; i < N²; i++) {

        for (j = 0; j < N; j++)
{

        \\ some O(1)
expressions
        }
}
```

What is the worst case time complexity (in the big O notation) of the following code fragment?

ⓘ Start presenting to display the poll results on this slide.

Example Solution

- What is the worst case time complexity (in the big O notation) of the following code fragment?

```
for (i = 0; i < N²; i++) {
        for (j = 0; j < N; j++)
{
        \\ some O(1)
expressions
        }
}
```

$T_2(n)$ is $O(N^2)$

$T_1(n)$ is $O(N)$

Overall: $O(N^2) \times O(N) = O(N^3)$

Example#7: Non-independent loops (Approach#1)

- **Example#5:** What is the worst case time complexity (in the Big O notation) of the following code fragment?

i=0	i=1	i=2	…	i=N-1
N	**N-1**	**N-2**	**…**	**1**

$$T(N) = 1 + 2 + 3 + \cdots + N = \frac{N(N+1)}{2} = \frac{1}{2}N^2 + \frac{1}{2}N \rightarrow O(N^2)$$

```
int a = 0;
for (i = 0; i < N; i++) {
        for (j = N; j > i; j--) {
                a = a + i + j;
        }
}
```

Example#7: Non-independent loops (Approach#2)

General Rule:

```
for (i = a; i <= b; i++) {
        //O(1) instructions
}
```

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

```
int a = 0;
for (i = 0; i < N; i++) {
        for (j = N; j > i; j--) {
                a = a + i + j;
        }
}
```

Example#7: Non-independent loops (Approach#2)

General Rule:

```
for (i = a; i <= b; i++) {
        //O(1) instructions
}
```

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

```
int a = 0;
for (i = 0; i < N; i++) {
        for (j = N; j >= i+1; j--) {
                a = a + i + j;
        }
}
```

Example#7: Non-independent loops (Approach#2)

General Rule:

```
for (i = a; i <= b; i++) {
        //O(1) instructions
}
```

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

```
int a = 0;
for (i = 0; i < N; i++) {
        for (j = i+1; j <= N; j++) {
                a = a + i + j;
        }
}
```

Example#7: Non-independent loops (Approach#2)

$$T(N) = \sum_{i=0}^{N-1} \sum_{j=i+1}^{N} 1 = \sum_{i=0}^{N-1} (N-i) = \sum_{i=0}^{N-1} N - \sum_{i=0}^{N-1} i = N^2 - \sum_{i=0}^{N-1} i = N^2 - \frac{N(N-1)}{2} = \frac{1}{2}N^2 + \frac{1}{2}N$$

$$\sum_{j=i+1}^{N} 1 = N - i$$

$$\sum_{i=0}^{N-1} N = N \sum_{i=0}^{N-1} 1 = N \times N = N^2$$

$$\sum_{i=0}^{N-1} i = \frac{N(N-1)}{2}$$

```
int a = 0;
for (i = 0; i < N; i++) {
        for (j = i+1; j <= N; j++) {
                a = a + i + j;
        }
}
```

Example#8: Non-independent nested loops

```
exampleEight(int n){
int count = 0;
for (int i=0; i<n; i++)
    for (int j=n-5; j<n; j++)
        for (int k=j; k<n; k++)
            count++;
}
```

$$T(n) = \sum_{i=0}^{n-1} \sum_{j=n-5}^{n-1} \sum_{k=j}^{n-1} 1 = 15n$$

Big-O: Question

Algorithms A and B spend exactly $\mathbf{T_A(n) = 0.1n^2 \log_{10} n}$ and $\mathbf{T_B(n) = 2.5n^2}$ microseconds, respectively, for a problem of size n.

Q1) Which algorithm is better in the Big-O sense?

Q2) Find out a problem size n_0 such that for any larger size $n > n_0$ the chosen algorithm outperforms the other slido

Algorithms A and B spend exactly $TA(n) = 0.1n^2 \log_{10}(n)$ and $TB(n) = 2.5n^2$ microseconds, respectively, for a problem of size n. Which algorithm is better in the Big-O sense?

ⓘ Start presenting to display the poll results on this slide.

Algorithms A and B spend exactly $T_A(n) = 0.1n^2 \log_{10}(n)$ and $T_B(n) = 2.5n^2$ microseconds, respectively, for a problem of size n. What is a problem size n0 such that for any larger size n > n0 the chosen algorithm outperforms the other

ⓘ Start presenting to display the poll results on this slide.

Big-O: Solution q1

Algorithms A and B spend exactly $\mathbf{T_A(n) = 0.1n^2 \log_{10} n}$ and $\mathbf{T_B(n) = 2.5n^2}$ microseconds, respectively, for a problem of size n.

Q1) Which algorithm is better in the Big-O sense?

$$\text{constant} < \log n < n^c (\text{where } 0 < c < 1) < n < n \log n < n^2 < n^3 \ldots < 2^n < 3^n < \ldots < n!$$

- $\mathbf{T_A(n) = 0.1n^2 \log_{10} n = c*n^2*\log_{10} n}$ **O(n² log n)**
- $\mathbf{T_B(n) = 2.5n^2 = c*n^2}$ **O(n²)**
- Constant grows slower than log n
- So Algorithm B is better in the big-O sense

Big-O: Solution q2

Algorithms A and B spend exactly $\mathbf{T_A(n) = 0.1n^2 \log_{10} n}$ and $\mathbf{T_B(n) = 2.5n^2}$ microseconds, respectively, for a problem of size n.

Q2) Find out a problem size n_0 such that for any larger size $n > n_0$ the chosen algorithm outperforms the other

- Algorithm B outperforms algorithm A when $\mathbf{T_B(n) < T_A(n)}$
- $2.5n^2 < 0.1n^2 \log_{10} n$
- Rearranging: $25 < \log_{10} n$, equivalently $\log_{10} n > 25$
- Recap: $\mathbf{b^c = a}$ is equivalent to $\mathbf{\log_b a = c}$
- $n > 10^{25}$
- So, $n_0 = 10^{25}$

Outline

- Big O of example algorithms discussed in previous lectures
- Examples – big O without counting the number of operations
- **General rules – big O without counting the number of operations**

Single loops with O(1) instructions

Loop running constant times: **O(1)**

- Loop runs constant times, performing O(1) operations at each iteration
- Time complexity = c*O(1) = O(1)

```
// c is a constant
for (int i = 0; i <= c; i++) {
        //O(1) instructions
}
```

Loop incrementing/ decrementing by constant c: **O(n)**

- Loop runs n/c times, performing O(1) operations at each iteration
- Time complexity = 1/c *O(n)* O(1) = O(n)

```
// c is a constant
for (int i = 0; i <= n; i+=c)
{
        //O(1) instructions
}
```

Loop divided/ multiplied by constant c: **O(log n)**

- Loop runs $\log_c(n)$ times, performing O(1) operations at each iteration
- Time complexity = $\log_c(n)$ * O(1) = O(log n)

```
// c is a constant
for (int i = 1; i <= n; i*=c)
{
        //O(1) instructions
}
```

Single loops with O(f(n)) instructions

Loop running constant times:

- Loop runs constant times, performing O(1) operations at each iteration
- Time complexity = c*O(f(n)) = **O(f(n))**

```
// c is a constant
for (int i = 0; i <= c; i++) {
       //O(f(n)) instructions
}
```

Loop incrementing/ decrementing by some constant c:

- Loop runs n/c times, performing O(f(n)) operations at each iteration
- Time complexity = 1/c *O(n)* O(f(n)) = **O(n*f(n))**

```
// c is a constant
for (int i = 0; i <= n; i+=c)
{
       //O(f(n)) instructions
}
```

Loop divided/ multiplied by some constant c:

- Loop runs $\log_c(n)$ times, performing O(f(n)) operations at each iteration
- Time complexity = $\log_c(n)$ * O(f(n)) = O(log n*f(n))

```
// c is a constant
for (int i = 1; i <= n; i*=c)
{
       //O(f(n)) instructions
}
```

Sequential Loops

- Want to find the dominant term – i.e. which loop takes the longest
- Can consider sequential terms separately. No product as not nested.

```
int a = 0, b = 0;

for (i = 0; i < N; i++) {
      a = a + rand();
}
```

O(N)

```
for (j = 0; j < M; j++) {
      b = b + rand();
}
```

O(M)

What is the worst case complexity in the big O notation?

- **O(max(N,M))**

What if M=N?

- **O(N)**

What if M ≪ N?

- **O(N)**

Nested Loops

- Complexity of nested loops equal to the number of times innermost statement executed*complexity of statement
- Complexity of inner loop*complexity of outer loop
- Care needed if loops are non-independent (we considered 2 approaches: counting operations and using Sigma)

```
for (int i = 0; i < n; i = i+1 )\\
{
        for (int j = 0; j < n; j = j + 1)
        {
                \\some O(f(n)) expressions
        }
}
```

Example: Inner loop runs n times for every iteration of outer loop

- Total number of nested loop iterations: $O(n)*O(n) = O(n^2)$
- At each iteration nested loop doing an $O(f(n))$ operation
- Overall time complexity = $O(f(n))*O(n^2)$ = $\mathbf{O(n^2 * f(n))}$

Care with general rules – check code!

```
// c is a constant
for (int i = 0; i <= n; i*=c)
{
        //O(f(n)) instructions
}
```

```
// c is a constant
for (int i = n;i>-1; i/=2) {
        //O(f(n)) instructions
}
```

Summary

Today's lecture: looked at using big O notation

- The growth of functions is usually described using the big-O notation
- If you are asked to find the BIG O of an algorithm, you **DO NOT** need to first count the operations and then calculate the BIG O
- For sequential codes. Example:
  - O(max(N,M))
- For nested codes, example:
  - $O(N) \times O(N) = O(N^2)$
- Care needed when loops are not independent

- **Next Lecture:** Big Ω and Θ notations