# Time Complexity for Loops

CSE 2320 – Algorithms and Data Structures
Alexandra Stefan

University of Texas at Arlington

# Summary & Conventions

- On Time complexity of for loops
  - Time complexity: T(N), T(N,M)
  - Big-Oh briefly
  - Tricky loops

- Terminology and notation:
  - 1 -> N will stand for 1,2,3,4,…(N-1), N
  - $\log_2 N = \lg N$
  - Use interchangeably:
    - *Runtime* and *time complexity*
    - *Record* and *item*

# How many times does the instruction *printf("B ")* execute? Worksheet

(The number of iterations of printf("B ") will determine the time complexity of this code. We will discuss time complexity formally later on.)

```
// Example A1.
for (i = 1; i<=7; i++) {
      printf("\nA: ");
      for(k=1; k<=5; k = k+1)
            printf("B ");
}

// Example A2. Replace hardcoded value 7 and 5 with variables N and M
for (i = 1; i<=N; i++) {
      printf("\nA: ");
      for(k=1; k<=M; k = k+1)
            printf("B ");
}
```

# How many times does the instruction *printf("B ")* execute? Answers

(The number of iterations of printf("B ") will determine the time complexity of this code. We will discuss time complexity formally later on.)

```
// Example A1.
for (i = 1; i<=7; i++) {
      printf("\nA: ");
      for(k=1; k<=5; k = k+1)
            printf("B ");      ////////  Answer: 35 = 7*5
}
```

```
//Example A2. Replace hardcoded value 7 and 5 with variables N and M.
for (i = 1; i<=N; i++) {
      printf("\nA: ");
      for(k=1; k<=M; k = k+1)
            printf("B ");    /////// Answer:  N*M
}
```

Since the expression that gives the count now depends on N and M, lets be more formal and use the notation : **T(N,M) = Θ(N*M)**

**We use T(N,M) to indicate that it is a function of N and M, and**

**use Θ to indicate that it is not the exact count of instructions, but the *time complexity* of the code.**

# How many times does the instruction *printf("B ")* execute? Worksheet

```
// Example A3.
for (i = 1; i<=5; i++) { // i goes up to 5
      printf("\nA: ");
      for(k=i; k<=5; k = k+1)  // here k starts at value i
            printf("B ");
}
```

*What is the output of this code?*

*Use a table to help you count the repetitions of printf(B ").*

# How many times does the instruction *printf("B ")* execute? Answer

```
// Example A3.      T(N) = ….
for (i = 1; i<=5; i++) { // i goes up to 5
      printf("\nA: ");
      for(k=i; k<=5; k = k+1)  // here k starts at value i
            printf("B ");
}
```

Output of Example A3:
A: B B B B B
A: B B B B
A: B B B
A: B B
A: B
A:
A:

The table on the right helps us count the number of executions of printf("B ").

| **i** | **k** takes values: | Repetitions of printf(B) (count of different values of k) |
|---|---|---|
| 1 | 1,2,3,4,5 | 5 |
| 2 | 2,3,4,5 | 4 |
| 3 | 3,4,5 | 3 |
| 4 | 4,5 | 2 |
| 5 | 5 | 1 |

Total: 5+4+3+2+1 = 5*(5+1)/2

(This is a summation of part of an arithmetic Series.)

# How many times does the instruction *printf("B ")* execute? Worksheet

Example A4.

Replace hardcoded value 5 with variable N.

Goal: e able to generalize and understand the loop's pattern of behavior.

```
for (i = 1; i<=N; i++) {
      printf("\nA: ");
      for(k=i; k<=N; k = k+1)
            printf("B ");
}
```

| **i** | **k** takes values: | Repetitions of printf(B) (count of different values of k) |
|---|---|---|
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |

# How many times does the instruction *printf("B ")* execute? Answer

Example A4.

Replace hardcoded value 5 with variable N.

Goal: e able to generalize and understand the loop's pattern of behavior.

```
for (i = 1; i<=N; i++) {
      printf("\nA: ");
      for(k=i; k<=N; k = k+1)
            printf("B ");
}
```

| i | k takes values: | Repetitions of printf(B) (count of different values of k) |
|---|---|---|
| 1 | 1,2,3,…,N | N |
| 2 | 2,3,…,N | N-1 |
| 3 | 3,…,N | N-2 |
| … | …. | … |
| i | (N+1-i) | N+1-i |
| …. | … | … |
| N-1 | (N-1),N | 2 |
| N | N | 1 |

Total: $N + (N-1) + \ldots + 3 + 2 + 1 = N*(N+1)/2 = (N^2+N)/2$

(This is a summation of part of an arithmetic Series.)

# Table Method for Time Complexity

```
// Example D.      T(N) = ….
for (i = 0; i<N; i++)
            for(k=1; k<N; k = k*2)
                        printf("A");
```

# Table Method – for Time Complexity

```
// Example D.      T(N) = ....
for (i = 0; i<N; i++)
        for(k=1; k<N; k = k*2)
                printf("A");
```

How many powers of 2 smaller than N are there?
$2^0, 2^1, \ldots, 2^p$ ( where $2^p < N <= 2^{p+1}$ )
We want p. Simplify by assuming: $N = 2^p$, and k<=N.
=> p = lgN. Here: $\lg N = \log_2 N$
Because of our simplification we may be off by a constant (i.e. the exact answer is (p-1) or (p+1)).
The missing '-1' or '+1' will generate lower-order terms => will not change the dominant term or the Θ .

Self check:
Use ((lgn )- 2) in the last columns and verify that the dominant term and Θ are still the same.

| i | k takes values: | Iterations of 'for k' for this i |
|---|---|---|
| 0 | 1, 2, 4, 8, 16 ,…, $2^p$ where $2^p$ is the largest power of 2 smaller than N | lgn |
| 1 | 1, 2, 4, 8, 16 ,…, $2^p$ | lgn |
| … | | |
| i | **1, 2, 4, 8, 16 ,…, $2^p$** | lgn |
| … | … | … |
| n-3 | 1, 2, 4, 8, 16 ,…, $2^p$ | lgn |
| n-1 | 1, 2, 4, 8, 16 ,…, $2^p$ | lgn |
| Total : | $\lg n + \lg n + \ldots + \lg n$ | |
| | $= n \lg n$ | |
| | $= \Theta(n \lg n)$ | |

# Time complexity

Practice

```
// Example C (source: Dr. Bob Weems)  T(N) = …
for (i = 0; i<N; i++)
        for(k=1; k<N; k = k+k)
                printf("A");
```

---

```
// Example D.     T(N) = ….
for (i = 0; i<N; i++)
        for(k=1; k<N; k = k*2)
                printf("A");
```

---

```
// Example E.     T(N) = ….
for (i = 0; i<N; i++)
        for(k=N; k>=1; k = k/2)
                printf("A");
```

---

```
// Example F.    T(N) = ….
for (i = 0; i<N; i++)
        for(k=1; k<N; k = k+2)
                printf("A");
```

# Time complexity Answers

Here we can easily solve without a table because the number of k values does not depend on i, so it is the same for every iteration of the outer loop.

```
// Example C (source: Dr. Bob Weems)  T(N) = …
for (i = 0; i<N; i++)                          → N
        for(k=1; k<N; k = k+k)                 → k: 1,2,4,8,16,32 …, last power of 2 smaller than N
                printf("A");
```

Geometric progression (powers of 2) => lgN values (not exact)

**Θ(NlgN)**

---

```
// Example D.     T(N) = ….
for (i = 0; i<N; i++)                          → N
        for(k=1; k<N; k = k*2)                 → k: 1,2,4,8,16,32 …, last power of 2 smaller than N
                printf("A");
```

Geometric progression (powers of 2) => lgN values (not exact)

**Θ(NlgN)**

---

```
// Example E.     T(N) = ….
for (i = 0; i<N; i++)                          → N
        for(k=N; k>=1; k = k/2)                → k: N, N/2, N/4, N/8, …, 8, 4, 2,1
                printf("A");
```

Geometric progression (/2) => (1+lgN) values (not exact)

**Θ(NlgN)**

---

```
// Example F.     T(N) = ….
for (i = 0; i<N; i++)                          → N
        for(k=1; k<N; k = k+2)                 → k:1,3,5,7,9, …, last odd smaller than N
                printf("A");
```

Arithmetic progression (+2) => N/2 values (not exact)

**Θ($N^2$)**

# Time complexity: T(n) notation

## Worksheet

We will use T(n) to refer to the time complexity of a piece of code that depends on some value, n. **If the code depends on more values, those will also be arguments of T**. E.g.: **T(n,m).**

**Also n may depend on other values.**

// Write the Θ (Theta) time complexity

```
for (i=left; i<right; i++)
    printf("A");
for (i=0; i<p; i++) {
    for (k=0; k<r; k++)
        printf("B");
    }
```

# Time complexity: T(n) notation

## Answer

We will use T(n) to refer to the time complexity of a piece of code that depends on some value, n. If the code depends on more values, those will also be arguments of T. E.g.: T(n,m).

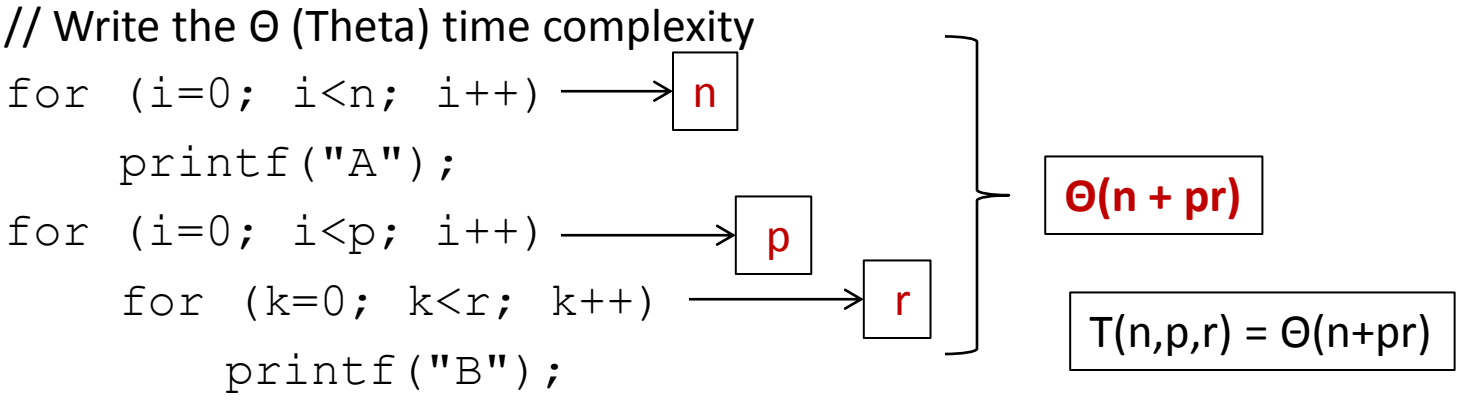
// Write the Θ (Theta) time complexity

```
for (i=left; i<right; i++)
    printf("A");
for (i=0; i<p; i++) {
    for (k=0; k<r; k++)
        printf("B");
    }
```

'for i' loop depends on value (right-left) =>
Let **n = (right-left)**
Now the code piece depends on: n,p and r =>
For its time complexity use: T(n,p,r)
Solve: **T(n,p,r) = Θ(n+pr)**

# Loops: Sequential vs Nested and Multiple Variables

We keep both the n term and the pr term because we assume n,p,r to be independent, positive integers. (We cannot and must assume that n will be less than pr. In some cases n can be much bigger than pr and it will be the dominant term.)



# Dominant Terms for Functions of Multiple Variables.

- Give the Theta for the function below

$27n^4m + 6n^3 + 7nm^2 + 100 = \Theta(\ldots\ldots\ldots\ldots\ldots\ldots)$

# Dominant Terms for Functions of Multiple Variables.

- Give the Theta for the function below

$27n^4m + 6n^3 + 7nm^2 + 100 = \Theta ( \mathbf{n^4m + nm^2} )$

*m and n are independent. (m can be much larger than n.)*

# Adding Terms (table method) for 'dependent' loops

Cannot multiply. The values that k takes depend on i => the number of iterations of 'for k' is different for every i => MUST explicitly add them.

```
for (i = 0; i<(n-1); i++)
        for(k=i+1; k<n; k = k+1)
                printf("A");
```

Steps:

- Table
- Summation (based on table)
- Closed form
- Dominant term:
- Θ (Theta): **$\Theta(n^2)$**

$$\text{formula}: 1+2+3+\ldots+y=\frac{y(y+1)}{2}$$

| i | k takes values: | Iterations of 'for k' for this i |
|---|---|---|
| 0 | 1 -> (n-1) | n-1 |
| 1 | 2 -> (n-1) | n-2 |
| … | | |
| i | (i+1) -> (n-1) | n-i-1 |
| … | … | … |
| n-3 | (n-2) -> (n-1) | 2 |
| n-2 | (n-1) -> (n-1) | 1 |

Total : $(n-1)+(n-2)+\ldots.+2+1$

$$=\frac{(n-1)n}{2}$$

$$=\frac{n^2-n}{2}$$

**If you add** the individual terms (counts of repetitions of the 'for k' loop), **you must NOT multiply by n anymore**.
The addition is already factoring in the contribution of the outer loop (each row corresponds to an i).

# Adding Terms (table method) for 'simple' code

The table method can be applied to the simple code (with 'independent' loops):

```
for (i = 0; i<n; i++)
        for(k=1; k<n; k = k+1)
                printf("A");
```

Steps:

- Table
- Summation (based on table)
- Closed form
- Dominant term:
- Θ (Theta): **$\Theta(n^2)$**

| **i** | **k** takes values: | # Iterations of 'for k' for this i |
|---|---|---|
| 0 | 1 -> (n-1) | n-1 |
| 1 | 1 -> (n-1) | n-1 |
| … | … | … |
| **i** | **1 -> (n-1)** | **n-1** |
| … | … | … |
| n-3 | 1 -> (n-1) | n-1 |
| n-1 | 1 -> (n-1) | n-1 |

Total : $(n-1)+(n-1)+....+(n-1)$

$= n(n-1)$

$= n^2 - n$

**Again, since you add** the individual terms, **you must NOT multiply by n anymore**.
The addition is already factoring in the contribution of the outer loop (each row corresponds to an i).

# Compact solution

This answer provides most of the information given with the table.

Note: for exams and homework, you must be able to do the Table method.

```
// Example G.     T(n) = ….
for (i = 0; i<n; i++)
            for(k=1; k<n; k = k+1)
                        printf("A");
```

n → n-1 → $\Theta(n^2)$

summation

closed form

$\Theta$

---

```
// Example H. (similar to selection sort processing)    T(n) = ….
for (i = 0; i<(n-1); i++)
            for(k=i+1; k<n; k = k+1)
                        printf("A");
```

$(n-1) + (n-2) + \ldots 2 + 1 = [n(n-1)]/2 = \Theta(n^2)$

---

```
// Example I. (similar to insertion sort processing) T(n) = ….
for (i = 1; i<n; i++)
            for(k=i-1; k>=0; k = k-1)
                        printf("A");
```

$1 + 2 + \ldots + (n-1) + (n-2) = [n(n-1)]/2 = \Theta(n^2)$

# Function Call Inside Loop

The time complexity for a function call is NOT 1, but the complexity derived from its code.

Assume that `foo` has time complexity: $T_{foo}(n) = \Theta(n)$

```
// Example 1. T(n) = n* Θ(n) = Θ(n²)
for (i = 0; i<n; i++)        ---> n
        foo(n);              ---> n
```

$\Theta(n^2)$

```
// Example 2. MUST use table method:
// => T(n) = n* Θ(n) = Θ(n²)
for (i = 0; i<n; i++)
        foo(i); // --> Θ(i)
```

| i | foo(i) takes: | Simplify to |
|---|---|---|
| 0 | Θ(1) | 1 |
| 1 | Θ(1) | 1 |
| … | … | … |
| i | Θ(i) | i |
| … | … | … |
| n-2 | Θ (n-2) | n-2 |
| n-1 | Θ (n-1) | n-1 |

Total: $1+2+\ldots+(n-2)+(n-1)$

$$= \frac{(n-1)n}{2}$$

$$= \frac{n^2 - n}{2}$$

Steps (for example 2):

- Table
- Summation (based on table)
- Closed form
- Dominant term:
- Θ (Theta): **Θ(n²)**

$$1+2+3+\ldots+y = \frac{y(y+1)}{2}$$

# Challenging Example

Source: code section from the LU decomposition code: "Notes 2: Growth of functions" by Dr. Weems.

```
1. for (i=1; i<=n-1; i++) {
2.   …
3.   for (k=i+1; k<=n; k++)   {
4.     temp=ab[ell][k];
5.     ab[ell][k]=ab[i][k];
6.     ab[i][k]=temp;

7.     for (t=i+1; t<=n; t++)
8.       ab[t][k] -= ab[t][i]*ab[i][k];
   }
   …
}
```

| i | k (loop 3) | t (loop 7) | Line 8 (nested 3&7) |
|---|---|---|---|
| 1 | | | |
| 2 | | | |
| … | | | |
| i | | | |
| … | | | |
| n-2 | | | |
| n-1 | | | |
| total | | | |

# Challenging Example

Source: code section from the LU decomposition code: "Notes 2: Growth of functions" by Dr. Weems.

```
1. for (i=1; i<=n-1; i++) {
2.   …
3.   for (k=i+1; k<=n; k++)   {
4.     temp=ab[ell][k];
5.     ab[ell][k]=ab[i][k];
6.     ab[i][k]=temp;

7.     for (t=i+1; t<=n; t++)
8.       ab[t][k] -= ab[t][i]*ab[i][k];
   }
  …
}
```

| i | k (loop 3) | t (loop 7) | Line 8 (nested 3&7) |
|---|---|---|---|
| 1 | 2<br>↓(n-1)<br>n | 2 -> n<br>…<br>2 -> n | $(n-1)^2$ |
| 2 | 3<br>↓(n-2)<br>n | 3 -> n<br>…<br>3 -> n | $(n-2)^2$ |
| … | … | … | … |
| **i** | **i+1**<br>**↓(n-i)**<br>**n** | **i+1 -> n** | **$(n-i)^2$** |
| … | … | … | … |
| n-2 | n-1<br>↓(2)<br>n | n-1 -> n<br>…<br>n-1 -> n | $(2)^2$ |
| n-1 | n<br>↓(1)<br>n | n -> n<br>…<br>n -> n | $(1)^2$ |
| total | | | $\sum_{y=1}^{n-1} y^2 = \Theta(n^3)$ |

See the approximation of summations by integrals example for solution to:

$$\sum_{y=1}^{n-1} y^2 = \Theta(n^3)$$

# Starting a business?

- Facebook: more than 2.07 billion monthly active users (in 2017)
- Assume:
  - If you start a business that has the potential to grow this much, and
  - Currently you have 30 users
  - You need to buy software and have 2 offers:
    - $N^2$
    - 1000N, also a bit more expensive
  - Switching from one software to the other later on, is undesired (disruption of service, uncertainty, increased cost …)
- Which one do you choose?

# Comparing growth of functions

- Comparing **linear**, **N lg N**, and **quadratic complexity**.

| N | N lg N | $N^2$ |
|---|---|---|
| $10^6$ (1 million) | ≈ 20 million | $10^{12}$ (one trillion) |
| $10^9$ (1 billion) | ≈ 30 billion | $10^{18}$ (one quintillion) |
| $10^{12}$ (1 trillion) | ≈ 40 trillion | $10^{24}$ (one septillion) |

- Quadratic time algorithms become impractical (too slow) much faster than linear and **N lg N** algorithms.
- Of course, what we consider "impractical" depends on the application.
  - Some applications are more tolerant of longer running times.

| N | lgN |
|---|---|
| 1000 | ≈ 10 |
| $10^6 = 1000^2$ | ≈ 2*10 |
| $10^9 = 1000^3$ | ≈ 3*10 |
| $10^{12} = 1000^4$ | ≈ 4*10 |

# Θ (Theta) made simple

- For any function we look at the 'fastest growing term' or the dominant term.
  - E.g. for $f(n) = 15n^3 + 7n^2+3n+20$, the dominant term is $15n^3$ .
  - Functions of multiple variables may have more than one dominant term!

  Consider $f(n,m) = 27n^4m + 6n^3 + 7nm^2 + 100$

  (Ask yourself: What if m is small and n is large? What if n is small and m is large?)

- Use Θ,Theta, for the dominant term but without the constant.
  - This is a oversimplification of Θ. We will study Θ formally in future lectures.
- Notation: **$f(n) = \Theta(n^2)$**
  - if the dominant term of $f(n)$ is $n^2$.
- Given a function, e.g. $f(n) = 15n^3 + 7n^2+3n+20$, find Theta:
  - find the dominant term: $15n^3$
  - remove the constant: $n^3$
  - $f(n) = \Theta(n^3)$

# CLRS - reference

- Book reference subchapters (the first number is the chapter number):
  - 1.2 Efficiency
  - Problem 1-1
  - See the pseudocode conventions in 2.1
  - In 2.2 see section “Order of growth”
  - 2.1 covers Insertion sort and discusses detailed instruction count part of that. You can revisit this subchapter after we talk about insertion sort.
  - 2.3 we will cover later on.

# Motivation for Big-Oh Notation

- Given an algorithm, we want to find a function that describes the time *performance* of the algorithm.
- Computing the number of instructions in detail is NOT desired:
  - It is complicated and the details are not important
  - The number of machine instructions and runtime depend on factors other than the algorithm:
    - Programming language
    - Compiler optimizations
    - Performance of the computer it runs on (CPU, memory)

    (There are some details that we would actually **NOT** want this function to include, because they can make a function unnecessarily complicated.)
- When comparing two algorithms we want to see which one is better for very large data. This is called the asymptotic behavior
  - It is not important what happens for small size data.
  - Asymptotic behavior = rate of growth = order of growth
- The Big-Oh notation describes the asymptotic behavior and greatly simplifies algorithmic analysis.