ECE430.217 Data Structures

# Algorithm Analysis

## Weiss Book Chapter 2


**Byoungyoung Lee**

**https://compsec.snu.ac.kr**

**byoungyoung@snu.ac.kr**



Lecture slides were prepared based on Douglas W. Harder's notes (**dwharder@alumni.uwaterloo.ca).**

# Outline

In this topic, we will examine code to determine the run time of various operations

We will introduce machine instructions

We will calculate the run times of:

- Operators **`+`**, **`-`**, **`=`**, **`+=`**, **`++`**, etc.
- Control statements **`if`**, **`for`**, **`while`**, **`do-while`**, **`switch`**
- Functions
- Recursive functions

# Motivation

The goal of algorithm analysis is to take a block of code and determine the **asymptotic run time** or **asymptotic memory requirements** based on various parameters

- Given an array of size $n$:
  - Selection sort requires $\Theta(n^2)$ time
  - Merge sort, quick sort, and heap sort all require $\Theta(n \ln(n))$ time
- However:
  - Merge sort requires $\Theta(n)$ additional memory
  - Quick sort requires $\Theta(\ln(n))$ additional memory
  - Heap sort requires $\Theta(1)$ memory

# Motivation

The asymptotic behavior of algorithms indicates the ability to scale

- Suppose we are sorting an array of size $n$

Selection sort has a run time of $\Theta(n^2)$

- $2n$ entries requires $(2n)^2 = 4n^2$
  - Four times longer to sort
- $10n$ entries requires $(10n)^2 = 100n^2$
  - One hundred times longer to sort

# Motivation

Merge/Quick/Heap sorting algorithms have $\Theta(n \ln(n))$ run times

– $2n$ entries require $(2n) \ln(2n) = (2n) (\ln(n) + 1) = 2(n \ln(n)) + 2n$

– $10n$ entries require $(10n) \ln(10n) = (10n) (\ln(n) + 1) = 10(n \ln(n)) + 10n$

In each case, it requires $\Theta(n)$ more time

However:

– Merge sort will require twice and 10 times as much memory

– Quick sort will require one or four additional memory locations

– Heap sort will not require any additional memory

# Motivation

To properly investigate the determination of run times asymptotically:

- We will begin with machine instructions
- Discuss operations
- Control statements
  - Conditional statements and loops
- Functions
- Recursive functions

# Machine Instructions

Given any processor, it is capable of performing only a limited number of operations

These operations are called *instructions*

For example, consider the operation `a += b;`

- Assume that the compiler has already has the value of the variable `a` in register `D1` and perhaps `b` is a variable stored at the location stored in address register `A1`, this is then converted to the single instruction

```
ADD (A1), D1
```

# Operators

Because each machine instruction can be executed in a fixed number of cycles, we may assume each operation requires a fixed number of cycles

– The time required for any operator is $\Theta(1)$ including:

- Retrieving/storing variables from memory
- Variable assignment `=`
- Integer operations `+ - * / % ++ --`
- Logical operations `&& || !`
- Bitwise operations `& | ^ ~`
- Relational operations `== != < <= => >`
- Memory allocation and deallocation `new delete`

# Operators

Of these, memory allocation and deallocation are the slowest by a significant factor

- Roughly over 100 times slowdown
- They require communication with the operation system

# Blocks of Operations

Each operation runs in $\Theta(1)$ time and therefore any fixed number of operations also run in $\Theta(1)$ time, for example:

```
// Swap variables a and b
int tmp = a;
a = b;
b = tmp;

// Update a sequence of values
++index;
prev_modulus = modulus;
modulus = next_modulus;
next_modulus = modulus_table[index];
```

# Blocks in Sequence

Suppose you have now analyzed a number of blocks of code run in sequence

```
template <typename T>
void update_capacity( int delta ) {
    T *array_old = array;
    int capacity_old = array_capacity;
    array_capacity += delta;
    array = new T[array_capacity];

    for ( int i = 0; i < capacity_old; ++i ) {
        array[i] = array_old[i];
    }

    delete[] array_old;
}
```

$\Theta(1)$

$\Theta(n)$

$\Theta(1)$

To calculate the total run time, add the entries: $\Theta(1 + n + 1) = \Theta(n)$

# Blocks in Sequence

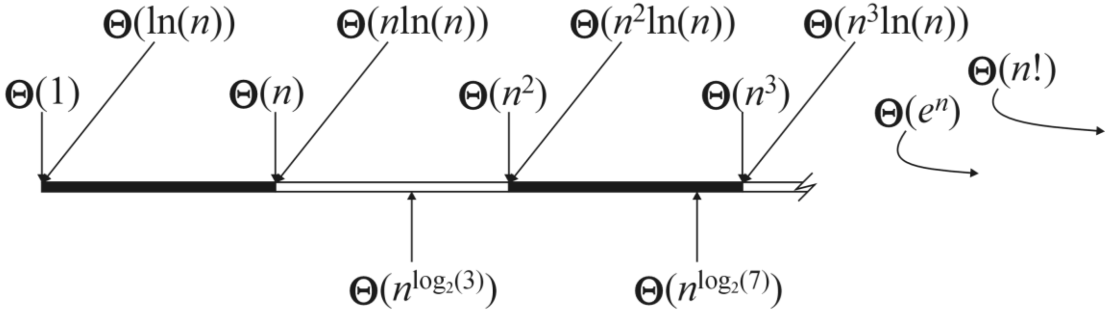
Other examples include:

- Run three blocks of code which are $\Theta(1)$, $\Theta(n^2)$, and $\Theta(n)$

  Total run time $\Theta(1 + n^2 + n) = \Theta(n^2)$
- Run two blocks of code which are $\Theta(n \ln(n))$, and $\Theta(n^2)$

  Total run time $\Theta(n \ln(n) + n^2) = \Theta(n^2)$

Recall this linear ordering from the previous topic




- When considering a sum, take the dominant term

# Control Statements

Statements which potentially alter the execution of instructions

- Conditional statements

  **`if`**, **`switch`**
- Condition-controlled loops

  **`for`**, **`while`**, **`do-while`**
- Count-controlled loops

  **`for (i=0; i++; i<=N)`**
- Collection-controlled loops

  **`for (auto i: int_counters)`** **`# C++11`**

# Control Statements

Given

```
if ( condition ) {
    // true body
} else {
    // false body
}
```

The run time of a conditional statement is:

- the run time of the condition (the test), plus
- the run time of the body which is run

# Condition-controlled Loops

The initialization, condition, and increment statements are usually $\Theta(1)$

For example,

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

# Condition-controlled Loops

If the body does not depend on the variable (in this example, `i`), then the run time of

```
for ( int i = 0; i < n; ++i ) {
    // code which is Θ(f(n))
}
```

is $\mathbf{\Theta}(n\,\mathrm{f}(n))$

If the body is $\mathbf{O}(\mathrm{f}(n))$, then the run time of the loop is $\mathbf{O}(n\,\mathrm{f}(n))$

# Condition-controlled Loops

```
int sum = 0;
for ( int i = 0; i < n; ++i ) {
    sum += 1;       // Θ(1)
}
```

This code has run time

$$\Theta(n \cdot 1) = \Theta(n)$$

# Condition-controlled Loops

```
int sum = 0;
for ( int i = 0; i < n; ++i ) {
    for ( int j = 0; j < n; ++j ) {
        sum += 1;       // Θ(1)
    }
}
```

The previous example showed that the inner loop is $\Theta(n)$, thus the outer loop is

$$\Theta(n \cdot n) = \Theta(n^2)$$

# Analysis of Repetition Statements

Suppose with each loop, we use a linear search an array of size $m$:

```
for ( int i = 0; i < n; ++i ) {
      // search through an array of size m
      // O( m );
}
```

The inner loop is $\mathbf{O}(m)$ and thus the outer loop is

$$\mathbf{O}(n\ m)$$

# Conditional Statements

Consider this example

```
void Disjoint_sets::clear() {
    if ( sets == n ) {
        return;
    }

    max_height = 0;
    num_disjoint_sets = n;

    for ( int i = 0; i < n; ++i ) {
        parent[i] = i;
        tree_height[i] = 0;
    }
}
```

$\mathbf{\Theta}(1)$

$\mathbf{\Theta}(1)$

$\mathbf{\Theta}(n)$

$$\mathrm{T}_{\text{clear}}(n) = \begin{cases} \Theta(1) & sets = n \\ \Theta(n) & \text{otherwise} \end{cases}$$

# Analysis of Nested Loops

For example,

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

$$\Theta\left(1+\sum_{i=0}^{n-1}\left(1+i\right)\right)=\Theta\left(1+n+\sum_{i=0}^{n-1}i\right)=\Theta\left(1+n+\frac{n\left(n-1\right)}{2}\right)=\Theta\left(n^2\right)$$

# Serial Statements

Suppose we run one block of code followed by another block of code

Such code is said to be run ***serially***

If the first block of code is $\mathbf{O}(\mathrm{f}(n))$ and the second is $\mathbf{O}(\mathrm{g}(n))$, then the run time of two blocks of code is

$$\mathbf{O}(\ \mathrm{f}(n) + \mathrm{g}(n)\ )$$

# Serial Statements

Consider the following two problems:

- 1) search through a random list of size $n$ to find the maximum entry
- 2) search through a random list of size $n$ to find if it contains a particular entry

What is the proper means of describing the run time of these two algorithms?

# Serial Statements

Searching for the maximum entry requires that each element in the array be examined, thus, it must run in $\mathbf{\Theta}(n)$ time

Searching for a particular entry may end earlier: for example, the first entry we are searching for may be the one we are looking for, thus, it runs in $\mathbf{O}(n)$ time

# Serial Statements

Therefore:

- if the leading term is big-$\mathbf{\Theta}$, then the result must be big-$\mathbf{\Theta}$, otherwise
- if the leading term is big-$\mathbf{O}$, we can say the result is big-$\mathbf{O}$

For example,

$$\mathbf{O}(n) + \mathbf{O}(n^2) + \mathbf{O}(n^4) = \mathbf{O}(n + n^2 + n^4) = \mathbf{O}(n^4)$$

$$\mathbf{O}(n) + \mathbf{\Theta}(n^2) = \mathbf{\Theta}(n^2)$$

$$\mathbf{O}(n^2) + \mathbf{\Theta}(n) = \mathbf{O}(n^2)$$

$$\mathbf{O}(n^2) + \mathbf{\Theta}(n^2) = \mathbf{\Theta}(n^2)$$

# Functions

A function invocation is a bit complex. We must consider:

- deal with arguments
- jump to the function
- execute the function
- deal with the return value
- clean up

# Functions

For simplicity, we will assume that the overhead required to make a function call and to return is $\Theta(1)$.

- This is in fact true as function calls/returns require a fixed cost
- You will learn more in computer architecture or operating system courses

Given a function $\mathrm{f}(n)$ (the run time of which depends on $n$) we will associate the run time of $\mathrm{f}(n)$ by some function $\mathrm{T}_{\mathrm{f}}(n)$

- We may write this to $\mathrm{T}(n)$

# Recursive Functions

A recursive function is a function calling itself.

For example, we could implement the factorial function recursively:

```
int factorial( int n ) {
    if ( n <= 1 ) {
        return 1;
    } else {
        return n * factorial( n - 1 );
    }
}
```

$$\Theta(1)$$

$$\mathrm{T}_!(n-1)+\Theta(1)$$

$$\mathrm{T}_!(n) = \begin{cases} \boldsymbol{\Theta}(1) & n \le 1 \\ \mathrm{T}_!(n-1) + \boldsymbol{\Theta}(1) & n > 1 \end{cases}$$

# Recursive Functions

The analysis of the run time of this function yields a recurrence relation:

$$\mathrm{T}_!(n) = \mathrm{T}_!(n - 1) + \boldsymbol{\Theta}(1) \qquad \mathrm{T}_!(1) = \boldsymbol{\Theta}(1)$$

Thus, $\mathrm{T}_!(n) = \boldsymbol{\Theta}(n)$

# Cases

As well as determining the run time of an algorithm, because the data may not be deterministic, we may be interested in:

- Best-case run time
- Average-case run time
- Worst-case run time

In many cases, these will be significantly different

# Cases: Example

Suppose you search a list linearly (to look for a particular element) :

We will count the number of comparisons

- Best case:
  - The first element is the one we're looking for: $\mathbf{O}(1)$
- Worst case:
  - The last element is the one we're looking for, or it is not in the list: $\mathbf{O}(n)$
- Average case?
  - We need some information about the list...

# Average Cases: Example #1

Assume the case we are looking for is in the list and equally likely distributed

If the list is of size $n$, then there is a $1/n$ chance to be in the $k^{\text{th}}$ location

Thus, we sum

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

Thus, the average case is $\mathbf{O}(n)$

# Average Cases: Example #2

Suppose we have a different distribution:

- there is a 50% chance that the element is the first
- for each subsequent element, the probability is reduced by ½

We could write:

$$\sum_{k=1}^{n} k\frac{1}{2^k} < \sum_{k=1}^{\infty} k\frac{1}{2^k} = 2$$

Thus, the average case is $\mathbf{O}(1)$

Power series [ edit ]

**Low-order polylogarithms** [ edit ]

Finite sums:

- $\sum_{k=0}^{n} z^k = \frac{1-z^{n+1}}{1-z}$, (geometric series)
- $\sum_{k=1}^{n} kz^k = z\frac{1-(n+1)z^n+nz^{n+1}}{(1-z)^2}$
- $\sum_{k=1}^{n} k^2z^k = z\frac{1+z-(n+1)^2z^n+(2n^2+2n-1)z^{n+1}-n^2z^{n+2}}{(1-z)^3}$
- $\sum_{k=1}^{n} k^mz^k = \left(z\frac{d}{dz}\right)^m \frac{1-z^{n+1}}{1-z}$

https://en.wikipedia.org/wiki/List_of_mathematical_series

# Summary

In these slides we have looked at:

- The run times of
  - Operators
  - Control statements
  - Functions
  - Recursive functions
- We have also defined best-, worst-, and average-case scenarios

We will be considering all of these each time we inspect any algorithm used in this class