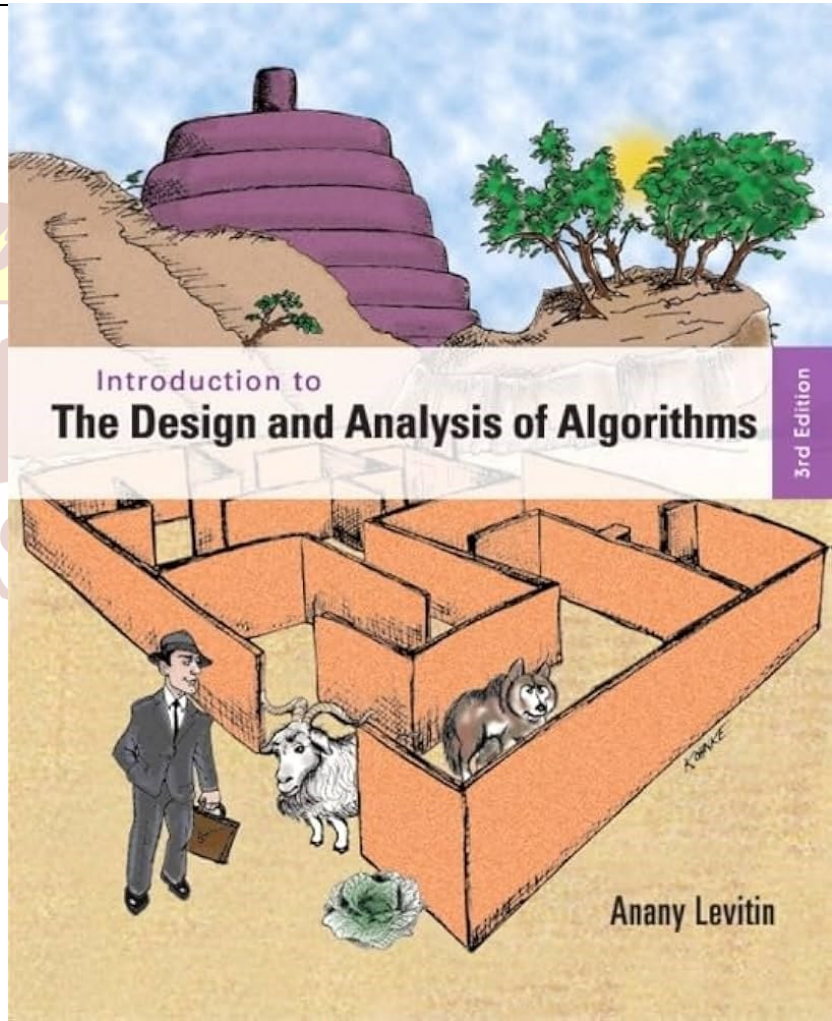
CS 07340 Design and Analysis of Algorithms

Week 2 Overview

- Fundamentals of the Analysis of Algorithm Efficiency
  - Big O, Ω, Θ



Fundamentals of the Analysis of Algorithm Efficiency

Runtime is usually measured not in a unit of time but a unit of number of steps/operations necessary to complete for the algorithm to come to a conclusion.

We will start with a **`SequentialSearch`** from our book (really a **`FindByValue`** as opposed to **`FindByIndex`**) in an array (an array is an ADT, but also used as the name of the data structure). First, we look at the pseudo code, and then we will try to count the number of steps for some examples.

ALGORITHM *SequentialSearch*$(A[0..n-1], K)$

//Searches for a given value in a given array by sequential search
//Input: An array $A[0..n-1]$ and a search key K
//Output: The index of the first element in A that matches K
// or -1 if there are no matching elements
$i \leftarrow 0$
while $i < n$ **and** $A[i] \neq K$ **do**
    $i \leftarrow i + 1$
if $i < n$ **return** i
else return -1

Example: For the array $\{0,1,2,3,4,5,6,7,8,9\}$, count the number of operations + in **`SequentialSearch(A[0..9], 7)`**.

Example: For the array $\{0,1,2,3,4,5,6,7,8,9\}$, count the number of operations + in **`SequentialSearch(A[0..9], 0)`**.

Example: For the array $\{0,1,2,3,4,5,6,7,8,9\}$, count the number of operations + in **`SequentialSearch(A[0..9], 9)`**.

How do these case scale with the size of the given array? What do you think should be the expected runtime? Does it even make sense to ask this question?

Example: The given code does nothing useful, but we can count the number of operations as a function of n. Count each of the operations:

- Assignment with constant
- Comparison
- Assignment with algebra op

Note that counts may depend on the value of i.

```
i = 0;
while(i < n){

    sum = sum * 2;
    sum++;

    for(j = 0; j < i; j++){
        product = i * j;
        sum += product;
    }

    i++;
}
```

Inner loop: $4i + 2$

Outer loop: $3 + \text{inner loop}$

While loop: $\sum_{i=0}^{n-1}(4i + 5) = 2n^2 + 3n$

All: $1 + (n + 1) + \text{while loop}$

So in total the number of steps is $2n^2 + 4n + 2$. For very large n this will be more or less a parabola $\approx 2n^2$ (quadratic). This algorithm (again, which does nothing useful) takes a quadratic number of steps in n for given n. When doubling the input size, the runtime quadruples!

Example: Count the minimum and the maximum number of steps the following algorithm takes. Why can't we count exactly here?

```
void sort(List<Integer> A) {
    for (int i = 0; i < A.size(); i++) {
        int smallestElementIndex = i;
        for (int j = i + 1; j < A.size(); j++)
            if (A.get(j) < A.get(smallestElementIndex))
                smallestElementIndex = j;
        swap(A, smallestElementIndex, i);
    }
}
```

Asymptotic Notation

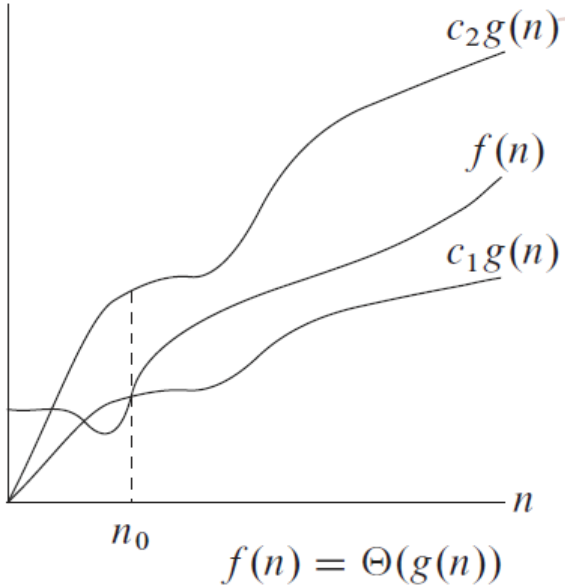
We use O, Θ, and Ω as mathematical notations to describe the asymptotic complexity of an algorithm (or function).

Asymptotic notation in mathematics refers to the growth of functions as the argument goes to infinity. You may have seen something like $\lim_{n\to\infty} \frac{an^2+bn+c}{n^2} = a$. That means that asymptotically, for large n, $an^2 + bn + c$ behaves like an^2. We can write $an^2 + bn + c = \Theta(n^2)$: up to a constant (in this case a), this function behaves like n^2 for **large** n. Sometimes such behavior only starts when n is past a certain value (large N_0), but asymptotically that does not make a difference – we are going to infinity with n.

Θ-Notation

We say that some algebraic expression / function $f(n)$ is $\Theta\big(g(n)\big)$ if $f(n)$ is essentially the same as $g(n)$ for large n, up to constant scalars. This is a ***tight bound***.

$$\Theta\big(g(n)\big) = \{f(n) \mid 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n > N_0\}$$



Think of it as an analog of the Sandwich Theorem (or Squeeze Theorem) from Calculus I. In Calculus terms it means $c_1 \leq \lim_{n\to\infty} \frac{f(n)}{g(n)} \leq c_2$, that is, for large n these two functions are roughly scalar multiples and grow at the same rate.

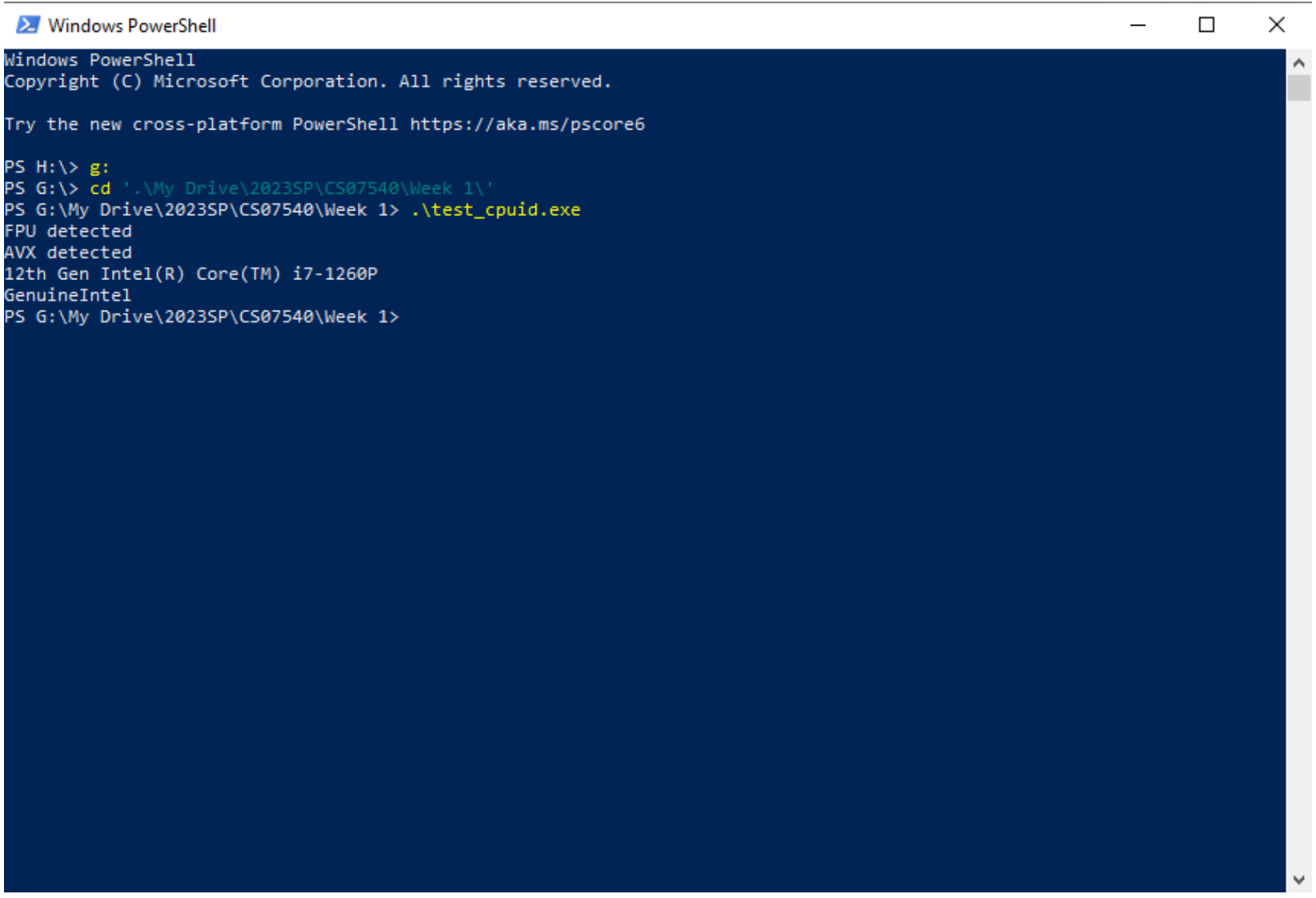
EXAMPLE FOR Θ

One of the fundamental ***abstract data types*** is an ***array***. It holds an ordered collection of items accessible by an integer index. It is usually implemented as a contiguous range of memory containing (or pointing to) the data and hence supports offset calculation. We are going to look at a C++/ASM example implementing the CPUID call on x86 CPUs. This call returns four 32-bit registers which we write to an array of length four (for capabilities of the CPU). The two stubs shown are a (partial) class declaration and its implementation. We will see the arrays as variable names initialized with type, name, and length:

```
uint32_t cpuid_output[4];  ← Array of type uint32_t and length 4
std::string name[13];      ← Array of type string and length 13
```

```
#if GCC
#ifdef __i386__
#define ARCH_X86
#endif
#endif

class cpuid {
    public:
    cpuid();
    ~cpuid();
    void call_cpuid_asm(uint32_t EAX, uint32_t ECX, uint32_t (&EAX_EBX_ECX_EDX)[4]);
    void extract_x86_flags(uint32_t ECX, uint32_t EDX);
    void extract_x86_extended_flags(uint32_t EBX);
    std::string get_brand_string();
    std::string get_vendor_string();

    bool has_fpu() const;
    …
    bool has_avx2() const;

    private:
    std::string brand_string;
    void set_brand_string();
    std::string vendor_string;
    void set_vendor_string();

    bool m_has_fpu = false;
    …
    bool m_has_avx2 = false;
};
```

```
void cpuid::set_brand_string()
{
    #ifdef ARCH_X86
    // store EAX, EBX, ECX, and EDX
    uint32_t cpuid_output[3][4];
    call_cpuid_asm(0x80000000, 0, cpuid_output[0]);
    if(cpuid_output[0][0] >= 0x80000004) {
        // retrieve brand string in EAX, EBX, ECX, EDX
        char brandString[48] = {0, };
        call_cpuid_asm(0x80000002, 0, cpuid_output[0]);
        call_cpuid_asm(0x80000003, 0, cpuid_output[1]);
        call_cpuid_asm(0x80000004, 0, cpuid_output[2]);
        // DWORD/uint32_t to char conversion
        for(int i=0; i<3; i++)
            for(int j=0; j<4; j++)
                *((uint32_t*) brandString + 4*i+j) = cpuid_output[i][j];
        this->brand_string = std::string(brandString);
    }
    #endif
}
```


Windows PowerShell
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.
Try the new cross-platform PowerShell https://aka.ms/pscore6
PS H:\> g:
PS G:\> cd '.\My Drive\2023SP\CS07540\Week 1\'
PS G:\My Drive\2023SP\CS07540\Week 1> .\test_cpuid.exe
FPU detected
AVX detected
12th Gen Intel(R) Core(TM) i7-1260P
GenuineIntel
PS G:\My Drive\2023SP\CS07540\Week 1>

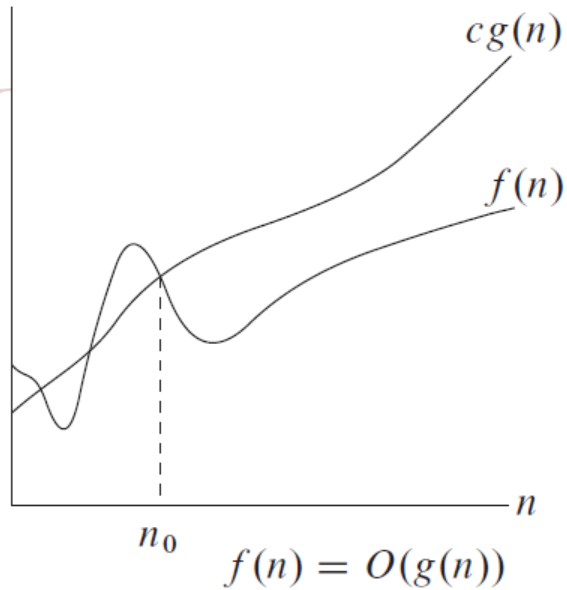
When we access an array to retrieve an element at index **`idx`**, the program uses the base address of the array (which is the address of **`Array[0]`**) and adds **`idx*size_of(data_type)`** to find the (virtual) memory address of the element at index **`idx`**. Hence it takes two operations (multiplication and addition) to retrieve an element from an array. With $c_1 = 1$ and $c_2 = 2$ we can say that retrieval in an array of length n, say $f(n)$, is bounded below and above by constants. Therefore, ***retrieval in any array*** (regardless of size n) ***is*** $\Theta(1)$ – constant time (if the array fits into physical main memory).

O-Notation

Θ-notation is not always attainable. We may not have a good grasp at the correct complexity. If we know a worst-case scenario, we can still create an ***asymptotic upper bound***.

$$O\big(g(n)\big) = \{f(n) \mid 0 \leq f(n) \leq c \cdot g(n) \text{ for all } n > N_0\}$$

means that $g(n)$ is an upper bound for these functions f. It does not mean it is a good bound (such as a least upper bound) or even of the correct order (n^2 versus n^3). We strive to use O only when it is a **tight** upper bound. Note that Θ-bounds are also O-bounds.



EXAMPLE FOR O

Suppose we want to find the position of a particular value in an array with **FINDINDEX** (or return **FAIL** if the value does not exist). The following Java-like pseudocode should help us determine the number of steps it would take at most to accomplish this task.

```
FindIndex(Array, value)
  for(int idx = 0; idx < n; i++) {
    if(Array[idx] == value)
      return idx;
    }
  return fail;
```


In best case, the first element (at index 0) is the correct element. In worst case, we have to go through the whole array. Hence **FINDINDEX** for an array has an upper bound n and is $O(n)$. Is this the best upper bound we can be sure of?

If the array is ordered, can we do better than $O(n)$?

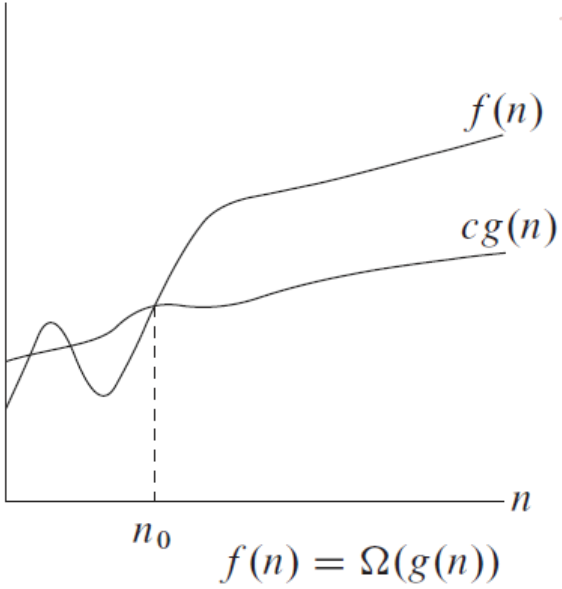
Yes, with a heap-like approach. If it is known that the array is ordered smallest to largest, then we can test the median (middle) index first. It will tell us whether the element we are looking for is in the first half or the second half of the array. That means we can ignore half of the array! Continuing recursively, we can find and element in $c \cdot \log_2(n)$ steps. ***For ordered arrays, find will be an*** $O(\log(n))$ ***algorithm!*** (Why do we write $O(\log(n))$ and not $O(\log_2(n))$?)

Ω-Notation

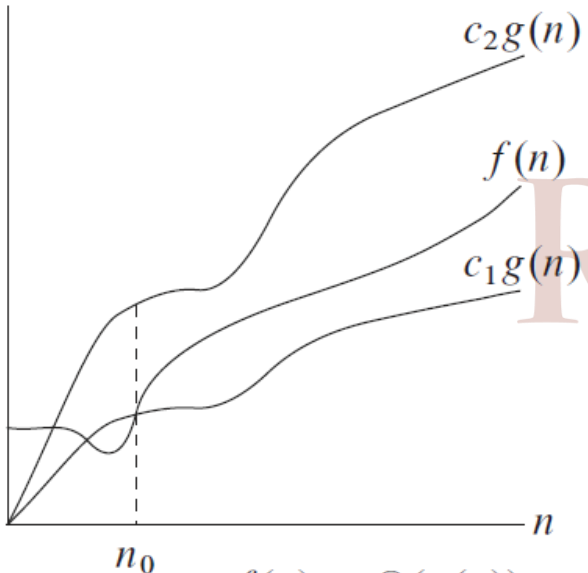
Given asymptotic upper bounds, it is natural to ask whether there are ***asymptotic lower bounds***. This is the holy grail!

$$O\big(g(n)\big) = \{f(n) \mid 0 \leq c \cdot g(n) \leq f(n) \text{ for all } n > N_0\}$$

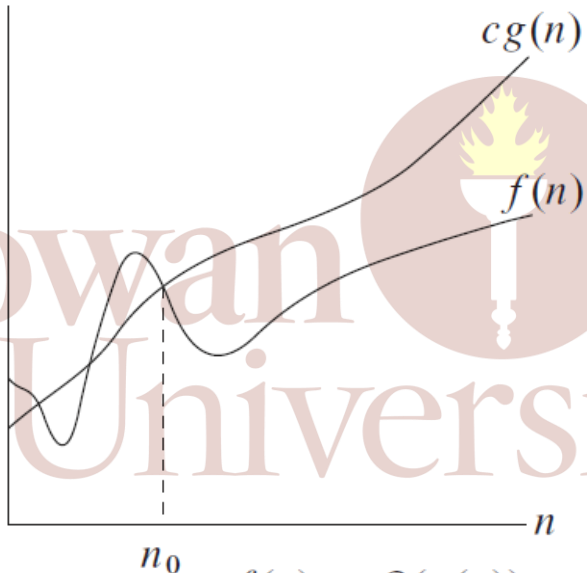
The three notations are linked in that a function is $\Theta\big(g(n)\big)$ exactly if it is both $O\big(g(n)\big)$ and $\Omega\big(g(n)\big)$.



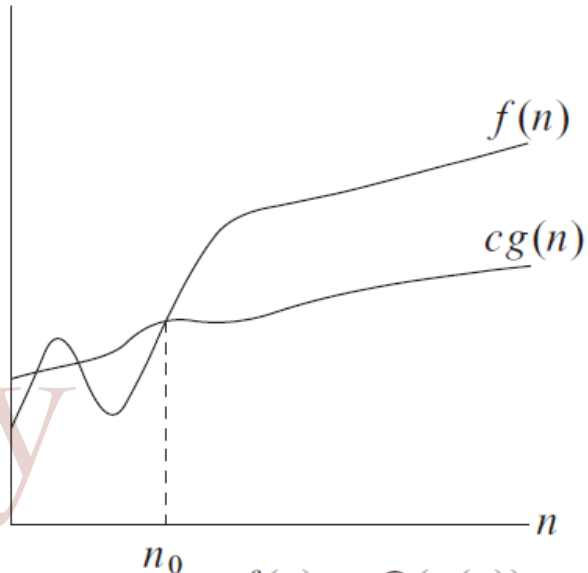
Finally, the O and Ω notations have "little" companions that indicate that the expression may not be tight.



$f(n) = \Theta(g(n))$



$f(n) = O(g(n))$



$f(n) = \Omega(g(n))$

What do these functions or expressions refer to? The notations are linked to specific algorithms! Algorithms are just a means to an end, so we need to distinguish between the problem we are trying to solve and the algorithm we are going to use. Think

- Problem: Find a minimal spanning tree
- Algorithm: Prim's Algorithm (which is just one specific algorithm)

Let's use an example.

- Problem: Sorting a list of values.
- Algorithm: Selection Sort

Selection Sort Example

Selection sort is an in-place comparison sorting algorithm. It has an $O(n^2)$ time complexity, which makes it inefficient on large lists, and generally performs worse than the similar insertion sort. Selection sort is noted for its simplicity and has performance advantages over more complicated algorithms in certain situations, particularly where auxiliary memory is limited. 📖

```java
import java.util.ArrayList;
import java.util.List;
import static java.util.Collections.swap;

public class SelectionSort {
    void sort(List<Integer> A) {
        for (int i = 0; i < A.size(); i++) {
            int smallestElementIndex = i;
            for (int j = i + 1; j < A.size(); j++)
                if (A.get(j) < A.get(smallestElementIndex))
                    smallestElementIndex = j;
            swap(A, smallestElementIndex, i);
        }
    }

    public static void main (String[] args){
        SelectionSort obj = new SelectionSort();
        List<Integer> A = new ArrayList<>(List.of(126,0,3, 215, 12, 22, 11));
        System.out.println("Given array");
        System.out.println(A);
        obj.sort(A);
        System.out.println("Sorted array");
        System.out.println(A);
        }
    }
```

If we start with a list of size n, we can analyze the loop structure and count the number of iterations. For $i = 0$ we get an inner loop of size n. For $i = 1$ we get $n - 1$, and so on. Since

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

we get an asymptotic runtime of $\left(\frac{1}{2}n^2 - \frac{1}{2}n\right)$ steps. Note that the n^2 term dominates for large n.

Formally we say that the worst-case runtime $T_A(X)$ of algorithm A with input X ($|X| = n$) is

$$T_A(n) = \max_{|X|=n} T_A(X)$$

Given a particular problem P (such as sorting), we first need an algorithm A that solves P before we can talk about runtime. Our algorithm is selection sort. With A being selection sort, the runtime T_A is $\frac{1}{2}(n^2 - n)$ for any input X of size n. The worst-case complexity of a problem P is the worst-case running time of the fastest algorithm for solving it.

$$T_P(n) = \min_{A \text{ solves } P} T_A(n)$$

$$T_P(n) = \min_{A \text{ solves } P} \left(\max_{|X|=n} T_A(X) \right)$$

When we describe an algorithm A that solves P in $O\big(f(n)\big)$ time, we have an ***upper bound*** on the complexity of P.

$$T_P(n) \leq T_A(n) = O\big(f(n)\big)$$

We have

- P is sorting
- A is selection sort (which is a particular implementation of sorting)
- $f(n) = \frac{1}{2}n^2 - \frac{1}{2}n \sim n^2$

Hence an upper bound on the complexity of sorting n objects is n^2, written $O(n^2)$.

But how do we know whether the algorithm we used to solve our problem is any good? Good compared to what? It would be nice to have both upper and lower tight bounds to evaluate an algorithm.

- $O(\quad)$ describes the upper bound of the complexity.
- $\Omega(\quad)$ describes the lower bound of the complexity.
- $\Theta(\quad)$ describes the exact bound of the complexity (upper and lower are the same, up to scalars and constants).
- $o(\quad)$ describes the upper bound excluding the exact bound.

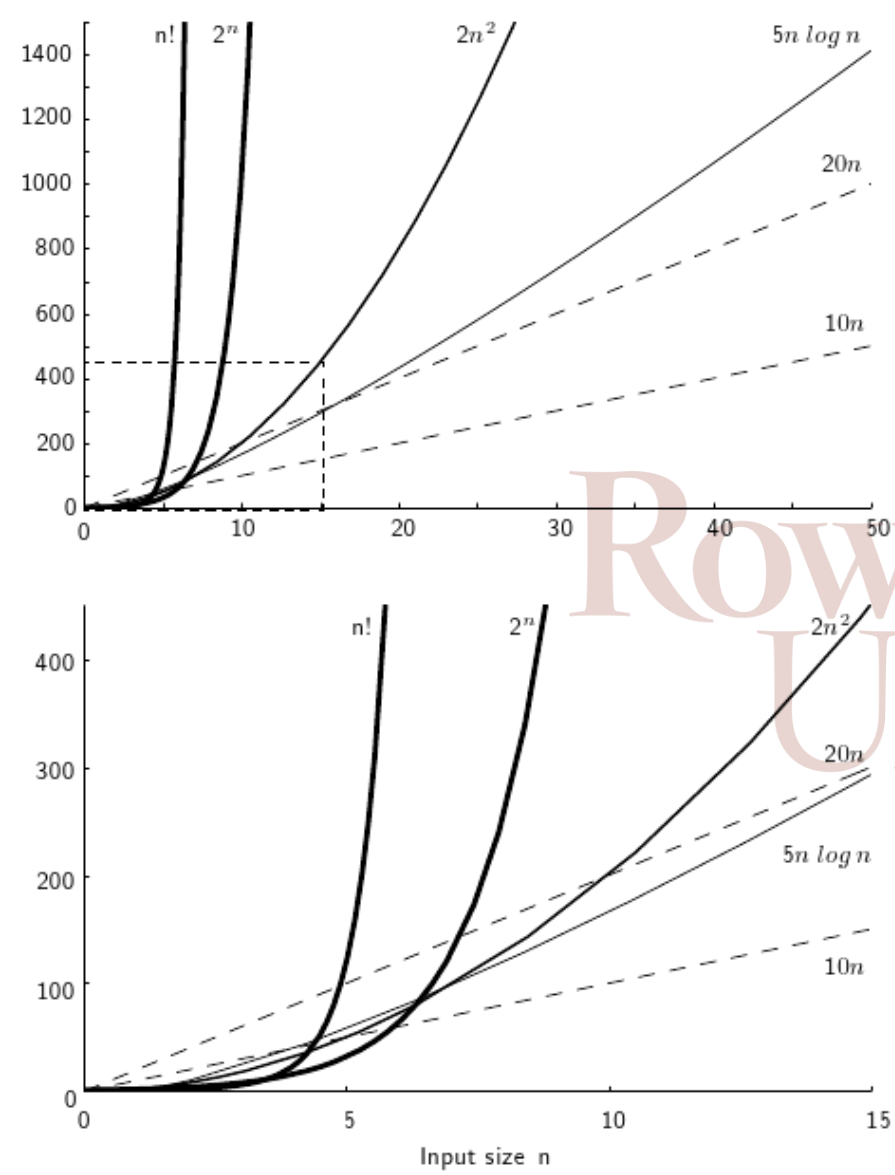
If we have $f(n) = \frac{1}{2}n^2 - \frac{1}{2}n$, then

- $f(n)$ is $O(n^3)$ and also $O(n^2)$ but not $O(n)$
- $f(n)$ is $\Omega(n)$, but we can also argue for $\Omega(n^2)$ for larger n
- $f(n)$ is $\Theta(n^2)$
- $f(n)$ is $o(n^3)$

Usually, when we write $O(\quad)$, we really want $\Theta(\quad)$.

TABLE 2.1 Values (some approximate) of several functions important for analysis of algorithms

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		



There is a tremendous difference between the growth of 2^n and $n!$, yet both are often referred to as "exponential-growth functions" (or simply "exponential"). Strictly speaking, only the former should be referred to as such since $n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$ (Stirling's Formula).

Algorithms that require an exponential number of operations are practical for solving only problems of very small sizes.

Sorting Complexity

Selection sort has complexity $O(n^2)$. Is this the best upper bound? No. We will see ***heap sort***, which has complexity $O(n \log n)$ (as does ***merge sort***). All of these are general comparison sorting algorithms without additional assumptions on the data. 📖

We are going to prove the worst-case runtime of comparison-based sorting algorithms is $\Omega(n \log n)$ later in the semester.

Worst-Case, Best-Case, and Average-Case Efficiencies

The worst-case efficiency of an algorithm is its efficiency for the worst-case input of size n – it runs the longest among all possible inputs of that size. It is the easiest to estimate since we assume worst performance for each subproblem. **This will be our focus**.

Example: Finding the minimum in an array of size n has worst performance $O(n)$, since we cannot be sure to have found the minimum until we looked at all n elements.

Example: Finding the minimum in an ordered array of size n has exact performance $\Theta(1)$, since we the minimum is $\text{Array}[0]$. Constant time. It does not get any better than that.

The analysis of the best-case efficiency is not nearly as important as that of the worst-case efficiency. However, a best-case analysis will give us a minimum runtime to expect from our algorithm.

In a typical environment, the worst case does not happen all the time. Rather, we expect some sort of "averaging out" of worst cases and best cases. This field of complexity analysis is called Amortized Analysis (*CS07540 Advanced Design and Analysis of Algorithms*). ***Amortized analysis*** is not average-case analysis – probability is not involved; an amortized analysis guarantees the average performance of each operation in the worst case for a sequence of operations.

It is the job of the algorithm ***analyst*** to figure out as much information about an algorithm as possible, and it is the job of the ***programmer*** who implements an algorithm to apply that knowledge to develop programs that solve problems effectively.

Amortized Analysis Example

A ***dynamic table*** is a list data structure with random access (array) and variable size. It provides an interface for getting, adding, and removing items. Random access arrays have $O(1)$ run-time cost, just like arrays. However, a dynamic table needs to be resized when its size limit is reached. That is a costly operation. We want to account for the $O(1)$ run-time cost balanced with the $O(n)$ cost of resizing. This is an example of aggregate analysis. Do we know any dynamic array/resizable array structures?

- **`ArrayList`** in **`java.util`**
- **`std::vector`** in C++ STL
- `[]` in Python

Instead, vector containers may allocate some extra storage to accommodate for possible growth, and thus the container may have an actual capacity greater than the storage strictly needed to contain its elements (i.e., its size). Libraries can implement different strategies for growth to balance between memory usage and reallocations, but in any case, reallocations should only happen at logarithmically growing intervals of size so that the insertion of individual elements at the end of the vector can be provided with ***amortized constant time*** complexity (see push_back).

Constructing objects with predefined properties often involves an unknown number of results at both compile time and run time. In order to collect these results, a dynamic table/list/array/set is the appropriate data structure. It may need to resize.

- What would be an appropriate strategy?
- What does that mean for insert performance?

A possible strategy is to double the size of the array each time it needs to be resized. If we insert with **`push_back`**, then each insert is $O(1)$ until we need to resize the array (which means reallocate memory and copy the existing entries with $O(2 \cdot \text{size})$). So what is the "cost" of n insert operations? For simplicity in the calculations, let $n = 2^k$ be a power of 2.

$$
\begin{aligned}
(1 + 2 + 4 + 8 + \cdots + n) + (n) &= \left(\sum_{i=0}^{k} 2^i \right) + 2^k \\
&= (2^{k+1} - 1) + 2^k \\
&= \mathbf{3n - 1}
\end{aligned}
$$

So even with copying, on average over n, the cost is still $O(1)$. 😊

A bad strategy is to increase the size one at a time:

$$(1+2+3+4+5+\cdots)+(n) = \left(\sum_{i=0}^{n+1} i\right)+(n)$$

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

$$= \frac{1}{2}n^2+\frac{5}{2}n+1$$

On average over n insert operations, this would be

$$\frac{\frac{1}{2}n^2+\frac{5}{2}n+1}{n} = O(n)$$

☹ The first strategy is obviously better.

Runtime Analysis through Recurrence

Generating functions are a mathematical tool for analyzing sequences of numbers. It uses formal power series (*Calculus II*) and functions (*Analysis*). Generating functions were introduced by de Moivre in 1730, in order to solve the general linear recurrence problem. He also created an approximation for $\log(n!)$ by showing $\lim_{n\to\infty} n! = \sqrt{2\pi}\left(\frac{n}{e}\right)^n$. This is what will actually help us determine the runtime of **HEAPSORT** later.

Let F_n denote the n-th Fibonacci number with $F_0 = 0$ and $F_1 = 1$. For $n \geq 2$, the recurrence relation is

$$F_n = F_{n-1} + F_{n-2}$$

Define a formal power series (*Calculus II*)

$$F(x) = \sum_{n=0}^{\infty} F_n x^n$$

The goal will be to find out that

$$F(x) = \frac{x}{1 - x - x^2}$$

and then use partial fraction decomposition (*Calculus II*) to rewrite the power series.

We use the recurrence as follows:

$$\begin{aligned} F(x) &= F_0 + F_1 x + \sum_{n=2}^{\infty} (F_{n-2} + F_{n-1}) x^n \\ &= x + x^2 \sum_{n=2}^{\infty} F_{n-2} x^{n-2} + x \sum_{n=1}^{\infty} F_{n-1} x^{n-1} \\ &= x + x^2 \cdot F(x) + x \cdot F(x) \end{aligned}$$

Instead of a recurrence between the coefficients of a function, we have a recurrence of the function. We can solve it using algebra to find a closed form of $F(x)$:

$$F(x) = \frac{x}{1 - x^2 - x}$$

The roots are $x_{1,2} = -\frac{1}{2}\left(1 \pm \sqrt{5}\right)$, called $-\varphi$ and $-\psi = {}^{1}/_{\varphi}$.

Using partial fraction decomposition, we find that

$$F(x) = \frac{1}{\sqrt{5}}\left(\frac{\psi}{x+\psi} - \frac{\varphi}{x+\varphi}\right)$$

We will rewrite each of those two fractions as geometric series, which will afford us to read off the coefficients directly. Recall that for $|x| < 1$,

$$\frac{1}{1-x} = \sum_{n=0}^{\infty} x^n$$

Hence

$$\frac{\psi}{x+\psi} = \frac{1}{1+\frac{x}{\psi}} = \frac{1}{1-\varphi x} = \sum_{n=0}^{\infty} \varphi^n x^n$$

and

$$-\frac{\varphi}{x+\varphi} = -\frac{1}{\frac{x}{\varphi}+1} = -\frac{1}{1-\psi x} = -\sum_{n=0}^{\infty} \psi^n x^n$$

This means we can write $F(x)$ as

$$F(x) = \frac{1}{\sqrt{5}}\left(\sum_{n=0}^{\infty} \varphi^n x^n - \sum_{n=0}^{\infty} \psi^n x^n\right) = \frac{1}{\sqrt{5}} \sum_{n=0}^{\infty} (\varphi^n - \psi^n) x^n$$

and therefore the n-th Fibonacci number is $\boldsymbol{F_n = \frac{1}{\sqrt{5}}(\varphi^n - \psi^n)}$.

The calculation of the n-th Fibonacci number through recursion is $\Theta(\varphi^n)$, so it is exponential. However, the direct formula can be used in $\Theta(\log(n))$ through ***fast exponentiation***.

Quiz Topics

- Step Counting
- Big-O Notation