DATA STRUCTURE

Dr. Rania Baashirah

r.baashirah@ubt.edu.sa

Mobile: +966-559009202

CHAPTER 2: Algorithm Analysis

Review

What is Data?

Data Types.

Why we use data type?

What is Data Structure.

Format for storing and organizing data.

Abstract data structure.

What is Algorithm.

Properties of Algorithms.

Introduction: Effective Algorithm

- An algorithm has to be effective to solve the problem in manageable time.
- Example:
- If we have two algorithms: A1, A2 to solve the same problem P
- How do we compare the two algorithms:
  - A1 is faster and takes less time to solve the problem than A2, so it is more effective.
  - Time is not the full story, there is space too
- This lecture gives us the tools to analyze the asymptotic complexity of algorithms.

Introduction: Effective Algorithm

- Algorithms $\mathbf{A1}$ and $\mathbf{A2}$ solve the same problem P.
- How do we know $\mathbf{A1}$ is faster than $\mathbf{A2}$?
- Implement both of them, run them on every possible input $\mathbf{Ii}$, time their execution
- Which is faster?
  - The one with the shortest average execution time?
  - The one with the shortest worst-case execution time?
  - The one with the shortest best-case execution time?
- Can we always run algorithm on all possible inputs?

Algorithm execution – Time Analysis

- Given input I_i from list of inputs.
- **Best-Case Execution Time (BCET):** The shortest time that the algorithm takes to solve the problem using certain input I_i.
- **Worst-Case Execution Time (WCET):** The longest time that the algorithm takes to solve the problem using certain input I_i.
- **Average-Case Execution Time (ACET):** The expected time that the algorithm takes to solve the problem on average using certain input I_i.

Algorithm execution – Time Analysis

- Which of the analysis metrics BCET, WCET, ACET is most useful?
  - Generally concentrate on WCET.
  - If WCET is manageable then algorithm is good.
- For certain algorithms, the WCET is very large and there are no better algorithms to solve same problem.
  - Use ACET as a metric.

Solve an Example:

- Sum the natural numbers 1 to n (Arithmetic series).
  - We know that
  - $SUM = 1 + 2 + \cdots + n = \frac{n(n+1)}{2}$
  - Which algorithm is faster A1 or A2 ?

<u>A1</u>

```
Function SUM(n: ℕ): ℕ
    result := n(n+1)/2
    return result
```

<u>A2</u>

```
Function SUM(n: ℕ): ℕ
  result := 0
  For i := 1 to n   do
     result := result + i

  return result
```

Implementing both Algorithms in python

- Execution time of Algorithm A2 Implementation when n=1,000,000

```
Sum is 50005000 required 0.0018950 seconds
Sum is 50005000 required 0.0018620 seconds
Sum is 50005000 required 0.0019171 seconds
Sum is 50005000 required 0.0019162 seconds
Sum is 50005000 required 0.0019360 seconds
```

About 2 ms

- Execution time of Algorithm A1 Implementation when n=(10,000; 100,000; 1,000,000; 10,000,000; and 100,000,000)

```
Sum is 50005000 required 0.00000095 seconds
Sum is 5000050000 required 0.00000191 seconds
Sum is 500000500000 required 0.00000095 seconds
Sum is 50000005000000 required 0.00000095 seconds
Sum is 5000000050000000 required 0.00000119 seconds
```

Less than 1 μs

Example – Time Analysis

- Let's assume the following.
  - CPU can execute only one operation at a time.
  - Any operation (Add, multiply, compare, assign…etc) *takes same constant time* **s**.
- What is the time T(A1)?
  - From the algorithm → 1 multiply + 1 divide + 1 add + 1 assign + 1 return
  - **T(A1) = 5s**
- What is the time T(A2)?
  - Inside loop → (2 assign + 1 condition + 1 add) repeated n times
  - T(A2) = 1 assign + n * (2s + s + s) + 1 assign + 1 return
  - **T(A2) = 3s + n(4s) = 3+4n**

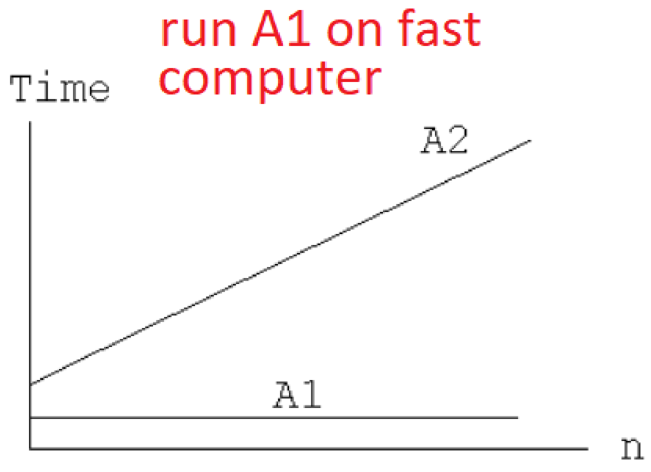
Example – Time Analysis

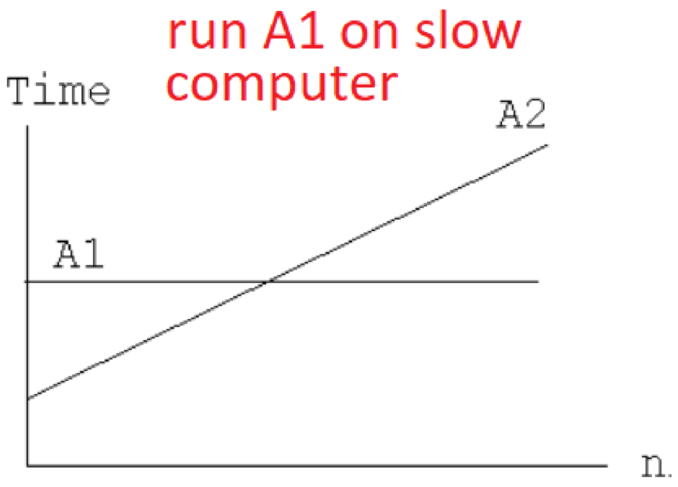
- Since s is a constant we can just consider it to be 1 time unit.
  - Also we can call it a **Step**.
  - So **s** is number of steps to be taken to execute the algorithm.
- We say that A1 has a constant growth (does not depend on n).
  - T(A1) = 5 → it takes 5 steps.
- We say that A2 has a linear growth in terms of n.
  - T(A2) = f(n) where f(n) = 4n + 3 → it takes 4n + 3 steps.

- **Now, can say: 4n + 3 > 5 ?** → in all cases?
- Answer: yes T(A2) > T(A1) when n >= 1

Example – Time Analysis

- When analyzing algorithms, we make simplistic assumptions.
  - Each instruction takes a constant time to execute.
- But in modern hardware:
  - We have multi-core computers.
  - There are accelerators e.g., caches, pipelines, branch predictors, etc.
  - Some instructions can be variable-latency e.g., memory load.





Example – Time Analysis

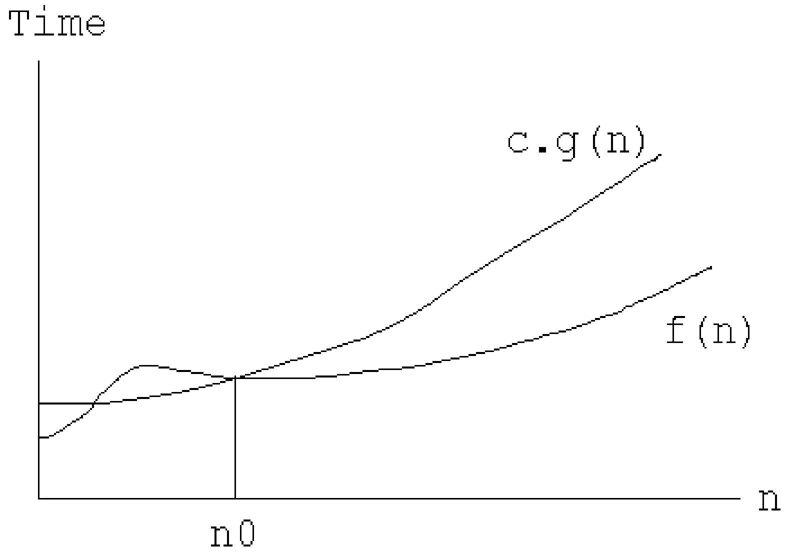
- We analyze algorithm performance when its input (n) is very large.
  - In mathematical algorithms, the input n may be just a number.
  - In algorithms deal with lists, the input n may be the number of elements.
  - In text search algorithms, the input n may be the number of letters.
  - And so on….
  - Small inputs most of the time spent as a **start-up time** in such cases.
  - The point that analyzing become interesting is called **n0**.
- So, We perform **asymptotic analysis** of algorithms.
  - Which means analyze algorithms when their input is very large i.e., they are **difficult instances.**

What is Big-Oh

- ***Order of magnitude*** is often called Big-O notation and written as a function $O(f(n))$.
- It provides a useful approximation to the actual number of steps in the computation.
- The function $f(n)$ provides a representation of number of steps based on n.
- We examine time and space.
- Example:
  - In some Algorithm A if the number of steps $f(n) = n^2 + 5n + 1$ then $O(f(n)) = O(n^2)$

Asymptotic Analysis – Big Oh

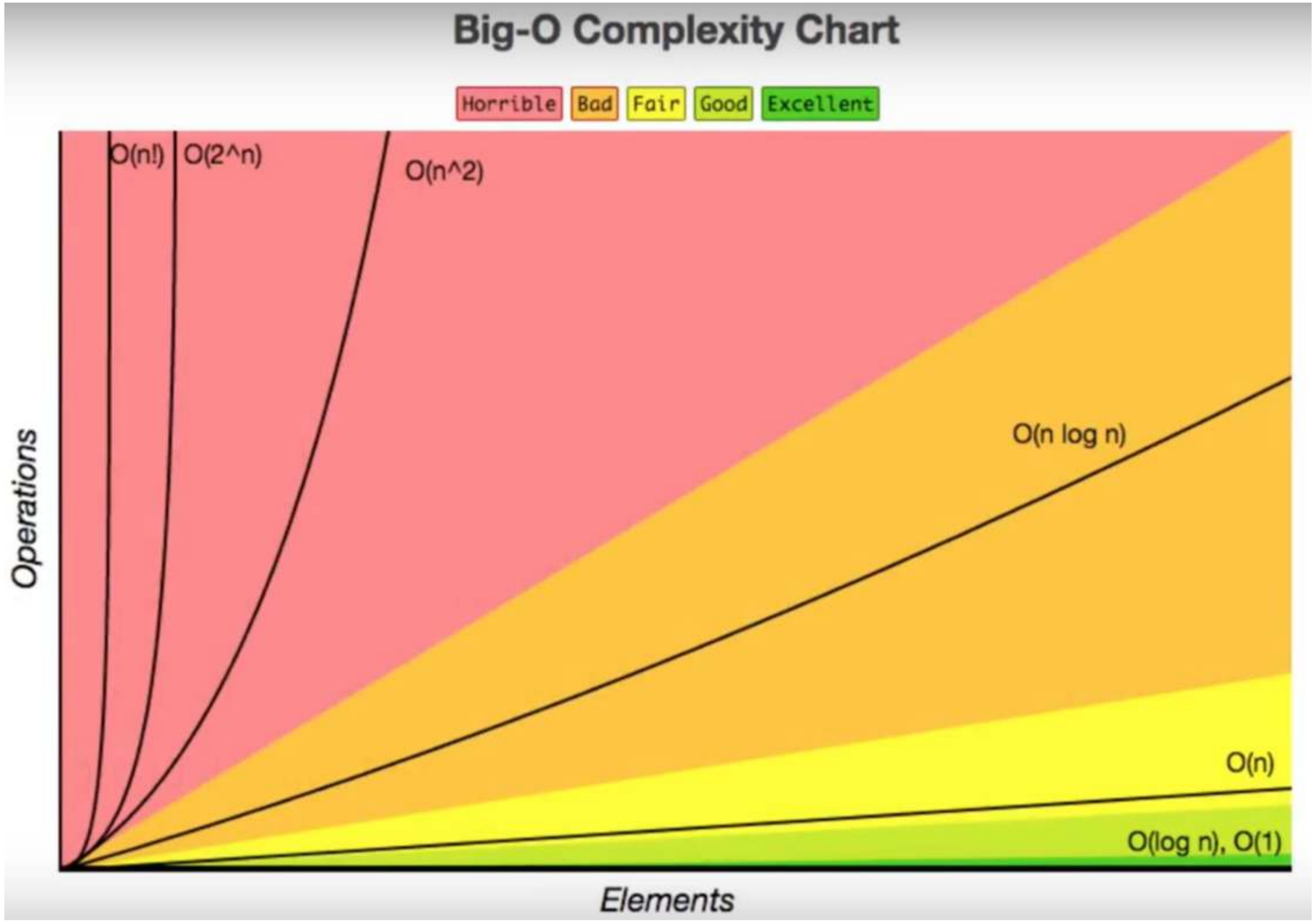
- Algorithm's execution time grows as a function of the input *n*.
- Given a function f(n) = 4n + 3, what is the minimal function g(n) that is always greater than f(n)?
  - we compare ***when the input n is very large (n >= n_0)***.
- We say that " g grows slower than f " if:
  - Written as: f(n) = O(g(n))
  - Read as: f is "**big Oh of**" g
  - Also: g is "asymptotically dominate f"
  - Also: g is an upper bound on f



Asymptotic Analysis – Big Oh

- We measure the algorithm in the Worst case.
- Ignores constants $5n = O(n)$
- Certain terms dominate the others, ignore low-order terms
  - $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(2^n) < O(n!)$

Asymptotic Analysis – Big Oh



Asymptotic Analysis – Big Oh

- Example of constant time:
  1. x = 5 + (15 * 20) → O(1)
  2. x = 5 + (15 * 20)

     y = 15 – 2

     print x + y;

     g(n) = 3 * O(1) = O(1) (drop constant)
- Example of linear time:
  1. For x in range (0, n) → O(n)

     print x; → O(1)

     g(n) = O(n) + O(1) = O(n)

Asymptotic Analysis – Big Oh

- Example of quadratic time:

  1. for x in range (0, n) → O(n)

         for y in range (0, n) → O(n)

             print x * y; → O(1)

     $g(n) = O(n) * O(n) * O(1) = O(n^2)$

Exercise (next class)

```
x = 5 + (15 * 20)
for x in range (0, n)
    print x * y;
for x in range (0, n)
    for y in range (0, n)
        print x * y;
```

What is the g(n)?

Exercise

- Given algorithm with time $f(n)=3n^2+4n+5$
- Simply g(n) will be equal to the highest degree of f(n), in this case 2
- $g(n) = n^2$ where $n >= 1$ ($n_0 = 1$)

More examples:

$3n^2+4n+5 \neq O(n)$

$3n^2+4n+5 = O(n^2)$

$3n^2+4n+5 = O(n^3)$

$3n^2+4n+5 = O(n^4)$

Asymptotic Analysis – Big Omega

- **Big Oh notation (worst case)**
  - $f(n) = O(g(n))$
  - $f(n)$ is big Oh of $g(n)$
- **Big Omega notation (best case)**
  - $f(n) = \Omega(g(n))$
  - $f(n)$ is big Omega of $g(n)$
  - $g(n)$ is faster than $f(n)$
  - $g(n)$ is the Lowest degree of $f(n)$
- Relation
  - $f(n) = O(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$

More examples:

$3n^2+4n+5 \neq O(n)$

$3n^2+4n+5 = O(n^2)$

$3n^2+4n+5 = O(n^3)$

$3n^2+4n+5 = O(n^4)$

More examples:

$3n^2 +4n+5 = \Omega(1)$

$3n^2 +4n+5 = \Omega(n)$

$3n^2 +4n+5 = \Omega(n^2)$

$3n^2 +4n+5 \neq \Omega(n^3)$

Asymptotic Analysis – Big Theta

- **Big Theta notation (Average case)**
  - $f(n) = \Theta(g(n))$
  - f(n) is big Theta of f(n)
  - f(n) grows as the same rate as g(n)

- Relations
  - $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$
- **Big Theta is True only when $O(g(n)) = \Omega(g(n))$**

More examples:

$3n^2+4n+5 \neq \Theta(n)$

$3n^2+4n+5 = O(n^2) = \Omega(n^2) = \Theta(n^2)$

$3n^2+4n+5 \neq \Theta(n^3)$

Asymptotic Analysis – Summary

- We have seen notation that we can use to describe the growth of a function.
  - Big Oh
  - Big Omega
  - Big Theta
  - Little Oh (not important)
  - Little Omega (not important)
- ***Instead of dealing with complex functions e.g., we just look at the higher order terms and drop constants.***
- For example.
  - $5n^4 + 2n^3 + 3n^2 + 4 = \Theta(n^4)$.

An Anagram Detection Example

- What is anagram?
  - One string is an anagram of another if the second is simply a rearrangement of the first. For example, 'heart' and 'earth' are anagrams. The strings 'python' and 'typhon' are anagrams as well.
  - The goal is to write a Boolean function that will take two strings and return whether they are anagram
- Is there an algorithm can detect if s1, and s2 strings are Anagrams?
- Answer: Yes, actually there are 4 different solutions!
  1. **Checking Off**
  2. **Sort and Compare**
  3. **Brute Force**
  4. **Count and Compare**

1- Checking off

- We check each letter in s1 and if it occurs in s2
- We check off the letter by replacing it with python "None"
- S1 = [e,a,r,t,h] = 5+4+3+2+1 =
- S2 = [h,t,,r,a,e]
- Steps = 15
- Strings in python are immutable → convert s2 to a list
- We check each character in s1 against the characters in s2 list
- If found → check off "None" → return Boolean
- $T(n) = 1+2+3+\ldots\ldots.+n = n(n+1)/2$
- $T(n) = O(n^2)$

2- Sort and Compare

- We convert each string into a list
- S1 = [e,a,r,t,h] = [a,e,h,r,t]
- S2 = [h,t,,r,a,e] = [a,e,h,r,t]
- We split, sort, and joint letters in s1 and s2
- We check if (s1= s2) → they are anagrams
- We use python built-in sort() method on the the lists

- Sorting complexity dominates iterations
- $T(n) = O(n^2)$ or $O(n\log n)$

3- Brute Force

- Trying all possibilities for both s1 and s2
- 3! = 3 * 2 *1
- n! = n * (n-1) * (n-2) *…….. *1
- We generate all possible strings from s1, which
- There are n possible characters for 1st position, (n-1) possible characters for 2nd position .. And so on. Total No. of candidate strings = n(n-1)(n-2)….3*2*1 = **n!**
- When “n” gets large → n! grows faster than 2^n
- Example: n = 20 → 20! → 2,432,902,008,176,640,000 strings
  - We need 77,146,816,596 years
- Complexity T(n) = O(n!)

4- Count and Compare

- We count the number of times each character occurs.
- We create 2 lists of 26 counters for all the 26 alphabets for both strings.
- Each time we see a character → we increment the counter by +1 at that position.
- The two lists will be identical if anagram.
- S1 = [a,b,c,d,….z] counter = [1,0,0,01.]
- S2 = [a,b,c,d,….z] counter = [1,0,0,01.]
- Complexity = 2 iterations + comparison of 26 characters
- $T(n) = 2n + 26 = O(n)$

Comparing Execution Time

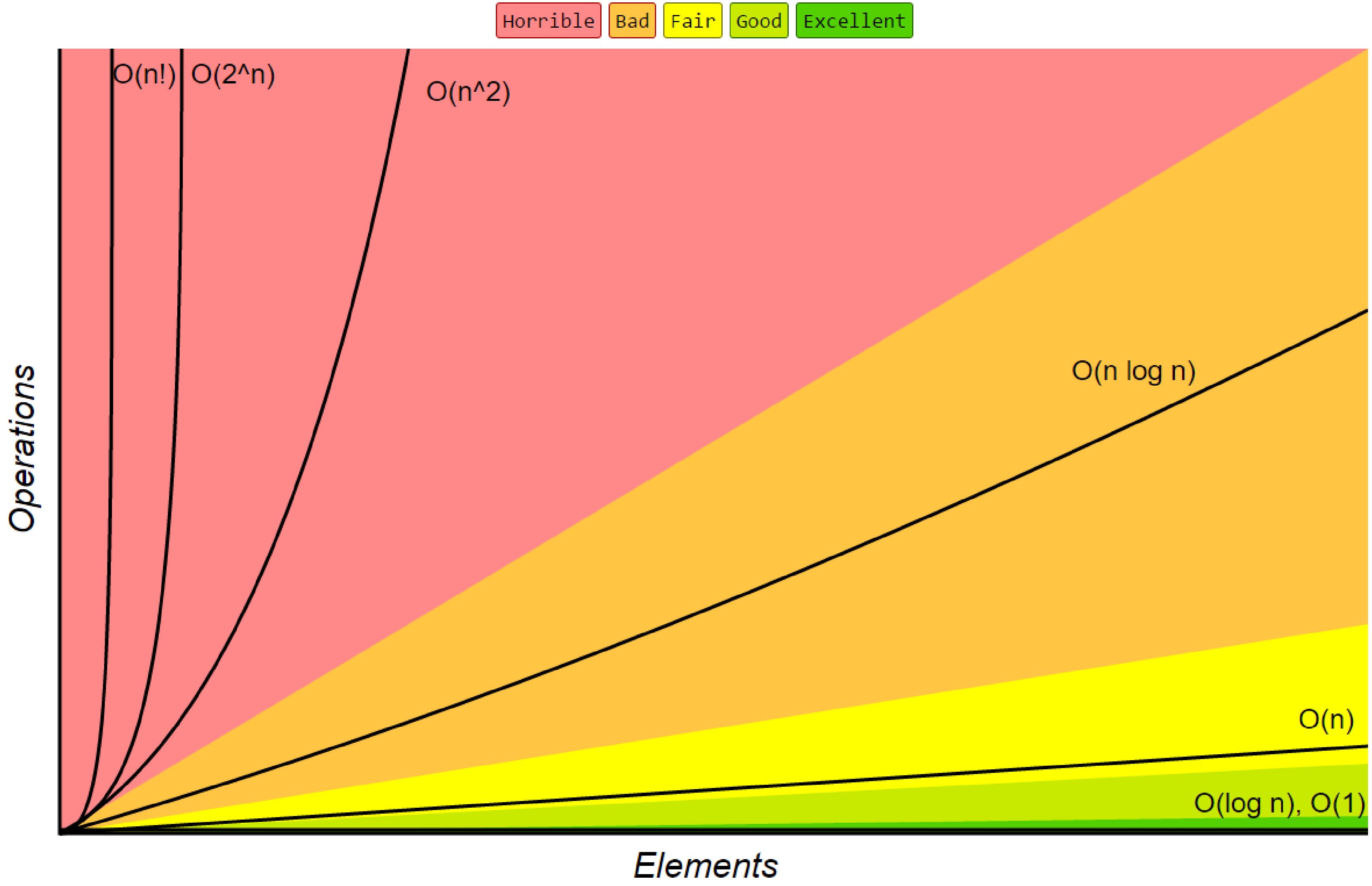
- Even though solution 4 was able to run the algorithm in linear time → it uses additional space for the two lists counts.
- We need to make a decision between the time and space

Asymptotic Dominance

- The functions grow faster from top row to bottom row i.e.,
  - $n! >> 2^n >> n^3 >> n^2 >> n \log n >> n >> \log n >> 1$
  - Here we use ">>" to mean "dominates"

Name	Time
Constant	1
Logarithmic	$\log n$
Linear	n
Log-linear (or linearithmic)	$n \log n$
Quadratic	n^2
Cubic	n^3
Polynomial	n^p
Exponential	b^n
Factorial	$n!$
Incomputable	∞

Asymptotic Dominance



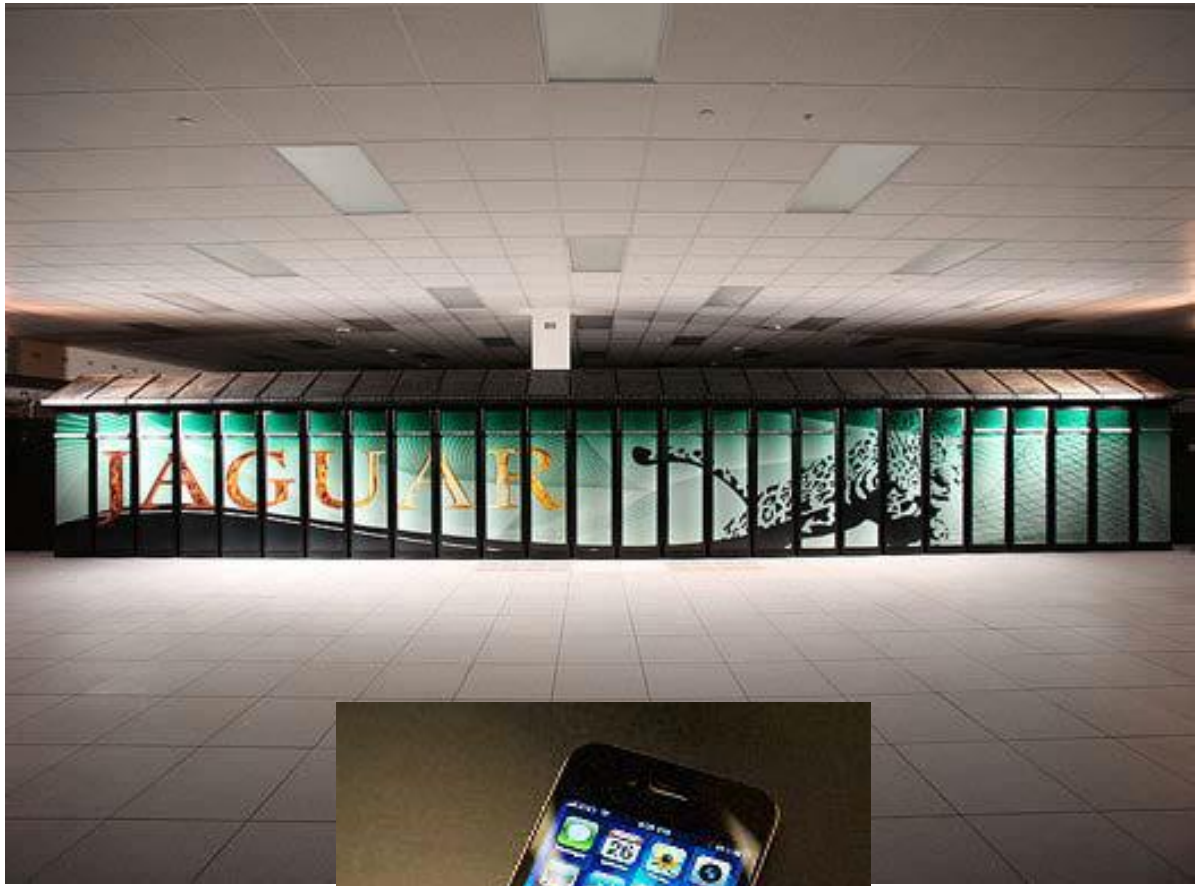
Asymptotic Dominance

- On a computer executing 1 instruction per ns, running time for varying input size.

n $f(n)$	$\lg n$	n	$n \lg n$	n^2	2^n	$n!$
10	0.003 μs	0.01 μs	0.033 μs	0.1 μs	1 μs	3.63 ms
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	1 ms	77.1 years
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	1 sec	8.4×10^{15} yrs
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	18.3 min	
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	13 days	
100	0.007 μs	0.1 μs	0.644 μs	10 μs	4×10^{13} yrs	
1,000	0.010 μs	1.00 μs	9.966 μs	1 ms		
10,000	0.013 μs	10 μs	130 μs	100 ms		
100,000	0.017 μs	0.10 ms	1.67 ms	10 sec		
1,000,000	0.020 μs	1 ms	19.93 ms	16.7 min		
10,000,000	0.023 μs	0.01 sec	0.23 sec	1.16 days		
100,000,000	0.027 μs	0.10 sec	2.66 sec	115.7 days		
1,000,000,000	0.030 μs	1 sec	29.90 sec	31.7 years		

Asymptotic Dominance

- The fastest supercomputer in the world: Cray XT5 Jaguar system at National Center for Computational Sciences. Processes more than 2×10^{17} instructions per second
- Your Apple iPhone, processes around 4×10^{8} instructions per second.



Asymptotic Dominance

- If we have two algorithms A1, A2 that solve the same problem P
  - A1 is $O(n^2)$
  - A2 is $O(n \log n)$
- We will run A1 on the super computer, and we will run A2 on the iPhone.
- The super computer is 4.4 million times faster than the iPhone.

Size of Input	Supercomputer running an $o(n^2)$ algorithm	Phone running an o(nlogn) algorithm
One million	0.0005 seconds	0.05 seconds
One thousand millions	8.3 minutes	1.24 minutes
One million millions	15.8 years	1.15 day

- A2 on iPhone is +5000 times faster than A1 on the super computer.

Asymptotic Dominance

- **Exponential algorithms (b^n):** are hopeless for anything beyond small input.
- **Quadratic algorithms (n^2):** are hopeless beyond about a million.
- **Log-linear algorithms (n log n):** are fine for up to one billion.
- **Logarithmic algorithms (log n):** grow remarkably slowly.
  - When an algorithm repeatedly half something, it can potentially have a logarithmic growth (log n)
  - Example: binary search

Asymptotic Analysis of Code

- We want to apply our knowledge of asymptotic analysis and reason about the time growth of some programs.
- Rules
  - Sequential statements.
    - T(s1; s2) = T(s1) + T(s2)
  - Selection statement.
    - T(if c then s1 else s2 end if) = T(c) + max(T(s1), T(s2))
  - Loop statement that iterates for a maximum number n.
    - T(while c do b end while) = n*(T(c)+T(b))+T(c)

Asymptotic Analysis of Code

- **Example 1:** given the following algorithm:

```
For i := 1 to n do
        s := s + 1
end for
```

- **Q: find the time complexity (write the big oh notation)**
- We go through the loop at most n times, $f(n) = 4n = O(n)$.
  - $g(n) = n$, and $n_0 >= 1$

Asymptotic Analysis of Code

- **Example 2**

```
For i in range (1,n): O(n)
     for j in range (1,n): O(n)
          s = s + 1    O(1)
     print (s)        O(1)
 T(n) = O(n) [O(n)] = O(n^2)
```

- Outermost loop executes n times
- Innermost loop executes n times for each iteration of the outermost loop
- This is n*n

- $f(n) = 3n * 4n = 12n^2 = O(n^2)$

Asymptotic Analysis of Code

- **Example 3**

```
For  i in range (1,n) O(n)
     for j in range (1,i) O(n)
          s = s + 1
     print (s)
end for
```

  - Outer loop executes n times
  - Inner loop executes i times for i^{th} iteration of the outermost
  - $1 + 2 + 3\ldots + n = \frac{n*(n+1)}{2}$
- $f(n) = \frac{1}{2}n^2 + \frac{1}{2}n = O(n^2)$

Performance of Common Python Data Structure

- We will study the performance of many algorithms used in ***Lists*** and ***Dictionaries***; the most two common Python data structure.
- Lists has many algorithms such as:
  - Assigning to index position.
  - Joining 2 lists
    - This operation can be done in many different ways in Python; each with different complexity.
  - Append an item
  - Remove an item
  - And so on

Lists

- Assigning to an index position (mylist[0] = 22) → O(1)
- To grow a list: two ways are there:
  - Using concatenation operator (`mylist = mylist + [4, 5]`) → O(k)
  - Using Append method `(mylist.append([4, 5])` → O(1)

Lists

- There are 4 methods to generate a list:
  - Concatenate
  - Append
  - Comprehension
  - List range

Lists

```python
def test1():
    l = []

    for i in range(1000):
        l = l + [i]

def test2():
    l = []
    for i in range(1000):
        l.append(i)

def test3():
    l = [i for i in range(1000)]

def test4():
    l = list(range(1000))
```

```python
from timeit import Timer

t1 = Timer("test1()", "from __main__ import test1")
print("concat", t1.timeit(number=1000), "ms")

t2 = Timer("test2()", "from __main__ import test2")
print("append", t2.timeit(number=1000), "ms")
t3 = Timer("test3()", "from __main__ import test3")

print("comprehension", t3.timeit(number=1000), "ms")

t4 = Timer("test4()", "from __main__ import test4")
print("list range", t4.timeit(number=1000), "ms")
```

```
concat 6.54352807999 milliseconds
append 0.306292057037 milliseconds
comprehension 0.147661924362 milliseconds
list range 0.0655000209808 milliseconds
```

Lists

- Notice that pop() and pop(i) are different.
- Also append and insert are different.

Operation	**Big-O Efficiency**
indexx[]	$O(1)$
index assignment	$O(1)$
append	$O(1)$
pop()	$O(1)$
pop(i)	$O(n)$
insert(i,item)	$O(n)$
del operator	$O(n)$
iteration	$O(n)$
contains (in)	$O(n)$
get slice [x:y]	$O(k)$
del slice	$O(n)$
set slice	$O(n + k)$
reverse	$O(n)$
concatenate	$O(k)$
sort	$O(n \log n)$
multiply	$O(nk)$

Table 2.2: Big-O Efficiency of Python List Operators

Dictionaries

- Collection of data values mapped with keys → pairs
- We can access items in dictionaries by the key rather than position such in lists.
- The complexity of dictionaries
- [(Ameer,1), (Faisal,3) ,(Ziad,8) ,(Feras,4)]

Dictionaries

- Notice that delete item in a dictionary is much faster than lists.
- Also 'contains' in dictionary is much faster than the one in list.

Operation	**Big-O Efficiency**
copy	$O(n)$
get item	$O(1)$
set item	$O(1)$
delete item	$O(1)$
contains (in)	$O(1)$
iteration	$O(n)$

Table 2.3: Big-O Efficiency of Python Dictionary Operations

Dictionary Vs List

Creating a list and a dictionary with random numbers