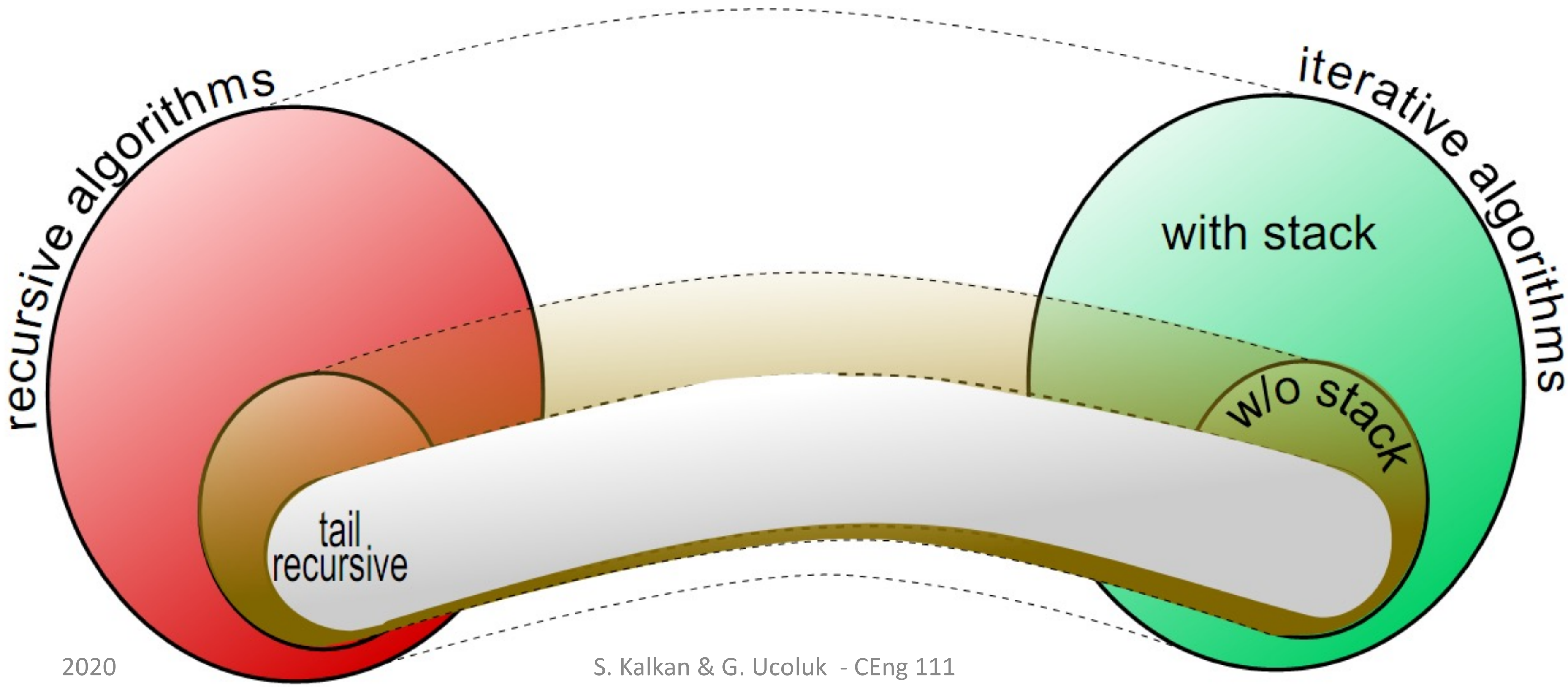
Ceng 111 – Fall 2021
Week 12b

Credit: Some slides are from the "Invitation to Computer Science" book by G. M. Schneider, J. L. Gersting and some from the "Digital Design" book by M. M. Mano and M. D. Ciletti.

Recursion vs. Iteration

Previously on CENG111!

- Any recursive algorithm can be transformed into an iterative algorithm.
- The reverse is also true.



Previously on CENG111!

Recursion vs. Iteration

- Which one is better?
- Better in what way?
  - Resource-wise:
    1. Iteration without stacks
    2. Iteration with stacks
    3. Recursion
  - Implementation-wise:
    1. Recursion
    2. Iteration without stack
    3. Iteration with stack

Big-O Notation; O()

Previously on CENG111!

f(n) is O(g(n)) if and only if there exists a real constant c>0, and a positive integer n_0, such that $|f(n)| \leq c|g(n)|$ for all $n \geq n_0$

- Example: for $f(n) = 2n^2$, $g(n) = n^2$.
  - f(n) is O(g(n)) = $O(n^2)$
  - But, it is also $O(n^3)$ and $O(n^4)$
  - We *prefer* the *smallest*.
  - For example:

$$f(n) = 9\log n + 5(\log n)^3 + 3n^2 + 2n^3 \in O(n^3).$$

Today

- Recursion vs. iteration
- Complexity

Administrative Notes

- THE3:
  - Deadline: 16 January.
- Final:
  - 5 Feb December, Saturday, 13:30

Other notations for computational complexity: $\Omega()$ Notation

f(n) is $\Omega(g(n))$ if and only if there exists a real constant $c > 0$, and positive integer n_0, such that

$$c|g(n)| \leq |f(n)| \text{ for all } n \geq n_0$$

- Lower boundary for f(n).
- $2n = \Omega(n)$
- $n^2 = \Omega(n^2)$

Other notations for computational complexity: $\Theta()$ notation

- $f(n) \in \Theta(g(n))$ if and only if there exists positive real constants c_1 and c_2, and a positive integer n_0, such that

$$c_1|g(n)| \leq |f(n)| \leq c_2|g(n)|$$

- Lower and upper boundary for f(n).
- $2n = \Theta(n)$
- $n^2 = \Theta(n^2)$

Example: We want to show that $1/2n^2 + 3n = \Theta(n^2)$.

Solution: $f(n) = 1/2n^2 + 3n$, $g(n) = n^2$

To show desired result, we need determine positive constants c_1, c_2, and n_0 such that

$0 \leq c_1 . n^2 \leq 1/2n^2 + 3n \leq c_2 . n^2$ for all $n \geq n_0$.

Dividing by n^2, we get $0 \leq c_1 \leq 1/2 + 3/n \leq c_2$

$c_1 \leq 1/2 + 3/n$ holds for any value of $n_0 \geq 1$ by choosing

$c_1 \leq 1/2$

$1/2 + 3/n \leq c_2$ holds for any value of $n_0 \geq 1$ by choosing

$c_2 \geq 7/2$

Thus, by choosing $c_1 = 1/2$ and $c_2 = 7/2$ and $n_0 = 1$, we can verify $1/2n^2 + 3n = \Theta(n^2)$.

Certainly other choices for the constants exist.

Relationships among O, Ω, and Θ - Notations

- f(n) is $O(g(n))$ iff g(n) is $\Omega(f(n))$
- f(n) is $\Theta(g(n))$ iff f(n) is $O(g(n))$ and f(n) is $\Omega(g(n))$
- $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$

Properties

1. We can ignore low-order terms in an algorithm's growth-rate function.
   - If an algorithm is $O(n^3+4n^2+3n)$, it is also $O(n^3)$.
   - We only use the higher-order term as algorithm's growth-rate function.

2. We can ignore a multiplicative constant in the higher-order term of an algorithm's growth-rate function.
   - If an algorithm is $O(5n^3)$, it is also $O(n^3)$.

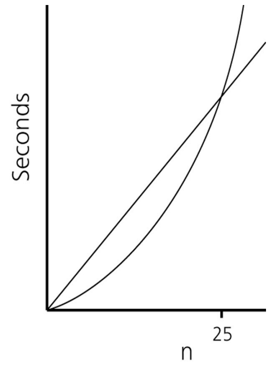
3. $O(f(n)) + O(g(n)) = O(f(n)+g(n))$
   - We can combine growth-rate functions.
   - If an algorithm is $O(n^3) + O(4n^2)$, it is also $O(n^3 + 4n^2)$ ➔ So, it is $O(n^3)$.
   - Similar rules hold for multiplication.

Properties

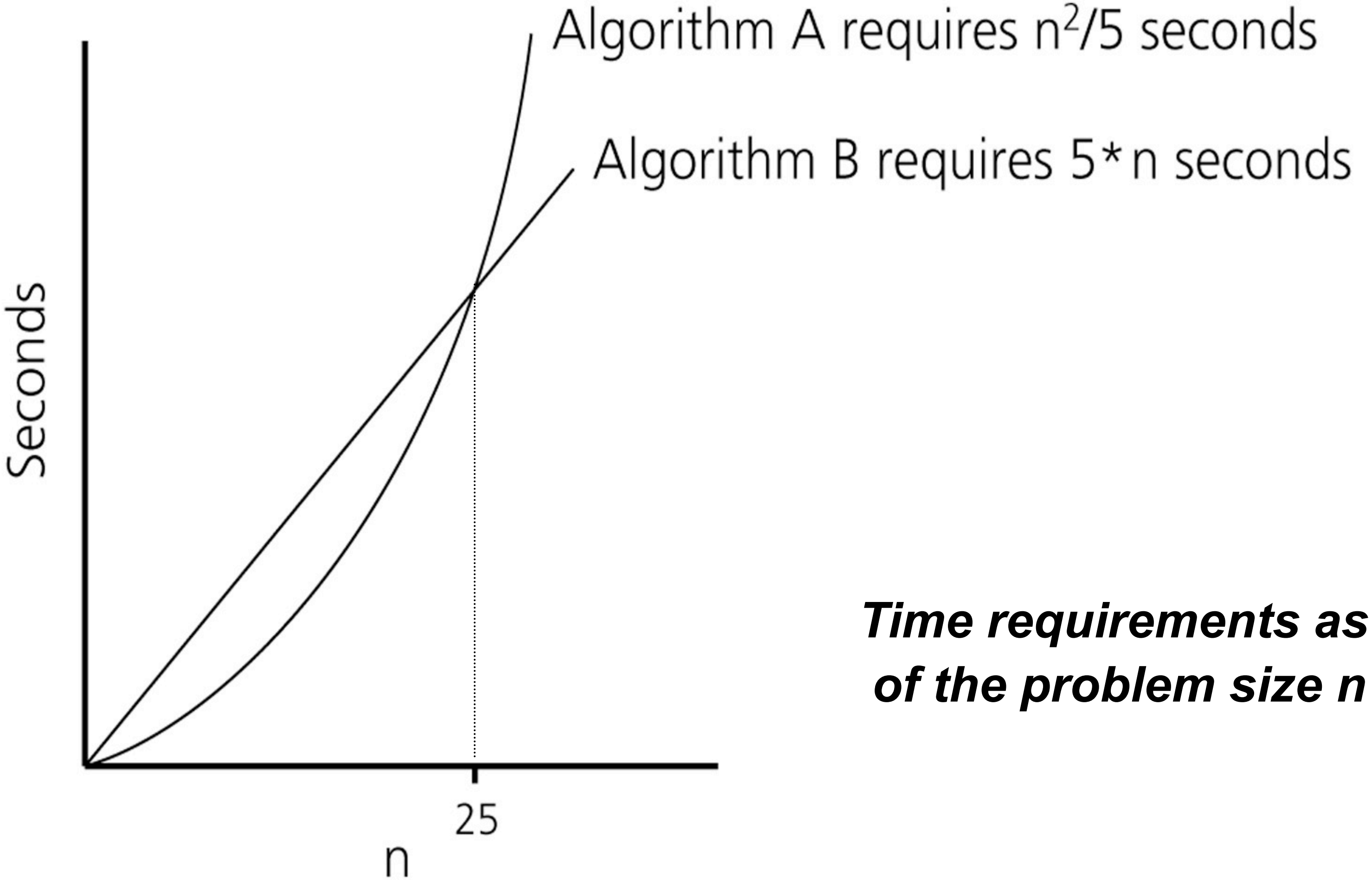
- A = B implies that B = A, right?
- But, $f(n) = O(g(n))$ does not imply $O(g(n)) = f(n)$.
- We prefer to see the '=' operator here as a membership operation:
  - $f(n) = O(g(n))$ implies that $f(n) \in O(g(n))$.
  - That means $O(g(n))$ is a set.

Problems with growth rate analysis

- An algorithm with a smaller growth rate will not run faster than one with a higher growth rate for all n, but only for all 'large enough' n!
- Algorithms with identical growth rates may have strikingly different running times because of the constants in the running time functions.
  - The value of n where two growth rates are the same is called the *break-even point*.



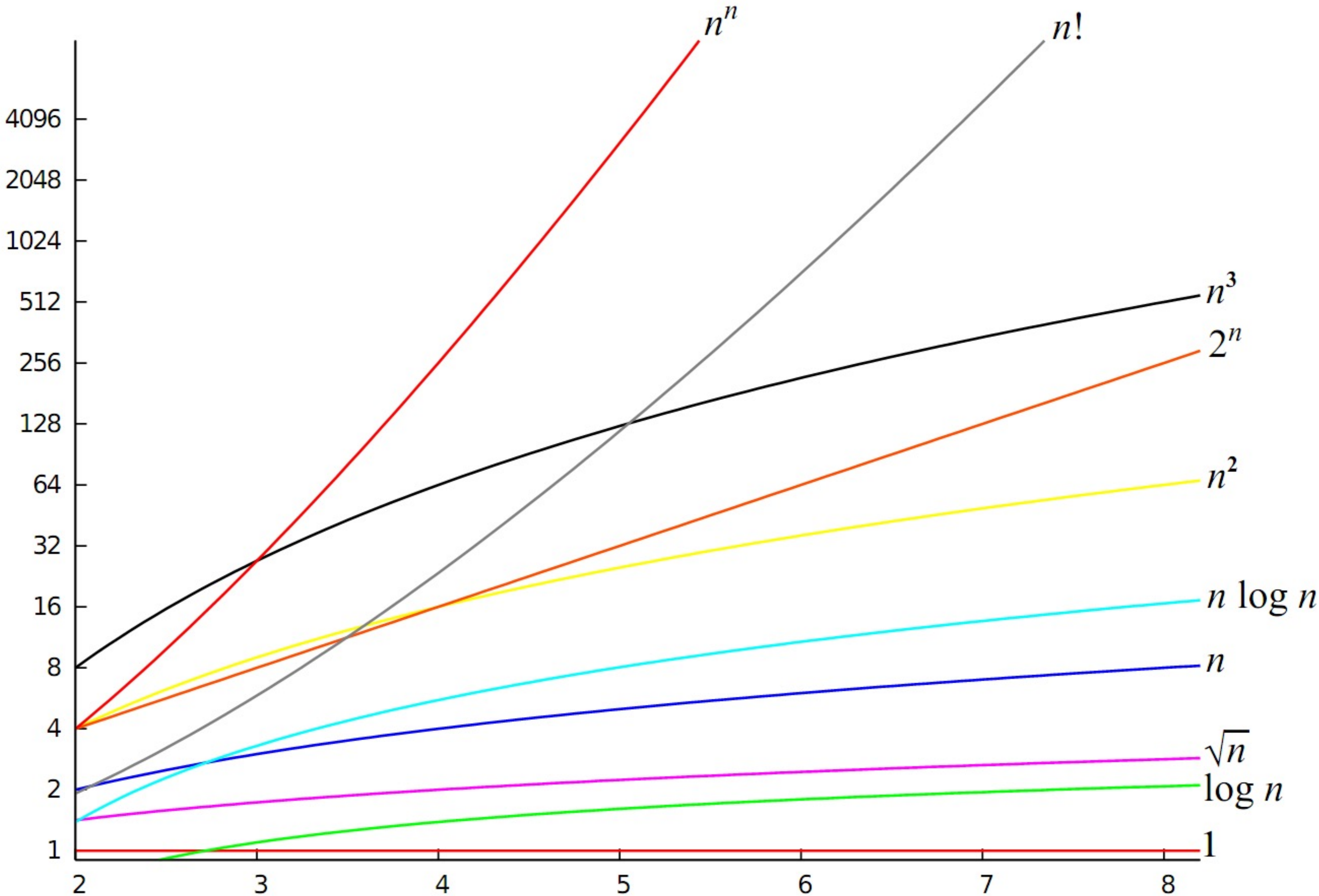
Problems with growth rate analysis



Time requirements as a function of the problem size n

Notation	Name	Example
$O(1)$	constant	Determining if a number is even or odd; using a constant-size lookup table or hash table
$O(\log n)$	logarithmic	Finding an item in a sorted array with a binary search or a balanced search tree as well as all operations in a Binomial heap.
$O(n^c),\ 0 < c < 1$	fractional power	Searching in a kd-tree
$O(n)$	linear	Finding an item in an unsorted list or a malformed tree (worst case) or in an unsorted array; Adding two n-bit integers by ripple carry.
$O(n \log n) = O(\log n!)$	linearithmic, loglinear, or quasilinear	Performing a Fast Fourier transform; heapsort, quicksort (best and average case), or merge sort
$O(n^2)$	quadratic	Multiplying two *n*-digit numbers by a simple algorithm; bubble sort (worst case or naive implementation), shell sort, quicksort (worst case), selection sort or insertion sort
$O(n^c),\ c > 1$	polynomial or algebraic	Tree-adjoining grammar parsing; maximum matching for bipartite graphs
$L_n[\alpha, c],\ 0 < \alpha < 1 = e^{(c+o(1))(\ln n)^\alpha (\ln \ln n)^{1-\alpha}}$	L-notation or sub-exponential	Factoring a number using the quadratic sieve or number field sieve
$O(c^n),\ c > 1$	exponential	Finding the (exact) solution to the traveling salesman problem using dynamic programming; determining if two logical statements are equivalent using brute-force search
$O(n!)$	factorial	Solving the traveling salesman problem via brute-force search; finding the determinant with expansion by minors.

From Wikipedia



n^n
$n!$
n^3
2^n
n^2
$n \log n$
n
$\sqrt{n}$
$\log n$
1
4096
2048
1024
512
256
128
64
32
16
8
4
2
1
2
3
4
5
6
7
8

Function	10	100	1,000	10,000	100,000	1,000,000
1	1	1	1	1	1	1
$\log_2 n$	3	6	9	13	16	19
n	10	10^2	10^3	10^4	10^5	10^6
$n * \log_2 n$	30	664	9,965	10^5	10^6	10^7
n^2	10^2	10^4	10^6	10^8	10^{10}	10^{12}
n^3	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	10^3	10^{30}	10^{301}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

Importance of Developing Efficient Algorithms

Sequential search vs Binary search

Array size	**No. of comparisons by seq. search**	**No. of comparisons by bin. search**
128	128	8
1,048,576	1,048,576	21
~4.10^9	~4.10^9	33

Execution times for algorithms with the given time complexities:

n	**f(n)=n**	**nlgn**	**n^2**	**2^n**
20	0.02 μs	0.086 μs	0.4 μs	1 ms
10^6	1μs	19.93 ms	16.7 min	31.7 years
10^9	1s	29.9s	31.7 years	!!! centuries

Anaysis of Algorihms, A.Yazici

Other Notations

o-Notation: f(n) is o(g(n)), "little-oh of g of n" is the following set:

$o(g(n)) = \{f(n)$: for all positive real constant $c > 0$, there exists a constant $n_0 \geq 0$ such that $0 \leq |f(n)| < c|g(n)|$ for all $n \geq n_0\}$

- We use o-notation to denote an upper bound that is not asymptotically tight, whereas O-notation may be asymptotically tight. Intuitively, in the o-notation, the function f(n) becomes insignificant relative to g(n) as n approaches infinity, that is,

  $$\lim_{n \to \infty} f(n)/g(n) = 0$$

  Ex: $n^2/2 \in o(n^3)$, since $\lim_{n \to \infty} (n^2/2)/n^3 = \lim_{n \to \infty} 1/2n = 0$

  Ex: $2n = o(n^2)$, but $2n^2 \neq o(n^2)$

Proposition: $f(n) \in o(g(n)) \Rightarrow O(f(n)) \subset O(g(n))$

Other Notations

ω - Notation: f(n) is $\omega(g(n))$, "little-omega of g of n", is the following set:

$\omega(g(n))$ = {f(n) : for all positive real constant c > 0, there exists a constant $n_0 \geq 0$ such that $0 \leq c|g(n)| < |f(n)|$ for all $n \geq n_0$}

ω-notation denotes a lower bound that is not asymptotically tight. The relation $f(n) = \omega(g(n))$ implies that,

$$\lim_{n \to \infty} f(n)/g(n) = \infty, \text{ if the limit exists.}$$

That is, f(n) becomes arbitrarily large relative to g(n) as n approaches infinity.

Ex: $n^2/2 = \omega(n)$, since $\lim_{n \to \infty} (n^2/2)/n = \infty$,

but $n^2/2 \neq \omega(n^2)$

Other notations

~ - Notation: Given the function g(n), we define ~g(n) to be the set of all functions f(n) having the property that

$$\lim_{n \to \infty} f(n)/g(n) = 1,$$

If $f(n) \in \sim g(n)$, then we say that f(n) is *strongly* asymptotic to g(n) and denote this by writing $f(n) \sim g(n)$.

Ex: $n^2 = \sim(n^2)$, since $\lim_{n \to \infty} n^2/n^2 = 1$,

Property: $f(n) \sim g(n) \Rightarrow f(n) \in \Theta(g(n))$

Family of Bachmann–Landau notations

METU Computer Engineering

Notation	Naming	Meaning
$f(n)$ is $O(g(n))$	• Big Omicron • Big O • Big Oh	f is bounded above by g (up to constant factor, as it was with Θ) asymptotically
$f(n)$ is $\Omega(g(n))$	• Big Omega	f is bounded below by g (up to constant factor, as it was with Θ) asymptotically
$f(n)$ is $\Theta(g(n))$	• Big Theta	f is bounded both above and below by g (up to constant factors) asymptotically
$f(n)$ is $o(g(n))$	• Small Omicron • Small O • Small Oh	f is dominated by g asymptotically
$f(n)$ is $\omega(g(n))$	• Small Omega	f dominates g asymptotically
$f(n) \sim \quad (g(n))$	• on the order of • twiddles	f is equal to g asymptotically

Complexity Analysis Examples


```python
def is_member(Item, List):
    for x in List:
        if Item == x:
            return True
    return False
```

Complexity Analysis Examples


```
def binary_search(item, List):
    # List: Sorted in ascending order
    length = len(List)
    middle = len(List)/2

    if item == List[middle]:
        return True
    if length == 1:
        return False
    if item < List[middle]:
        return binary_search(item, List[:middle])
    else:
        return binary_search(item, List[middle+1:])
```

Complexity Analysis Examples

METU Computer Engineering

```
def csort(A):
        # Assume that the numbers are in the range 1,...,k
        k = max(A)
        C = [0] * k

        # Count the numbers in A
        for x in A:
                C[x-1] += 1

        # Accumulate the counts in C
        i = 1
        while i < k:
                C[i] += C[i-1] i += 1

        # Place the numbers into correct locations
        B = [0] * len(A)
        for x in A:
                B[C[x-1]-1] = x C[x-1] -= 1

        return B
```

Complexity Analysis Examples

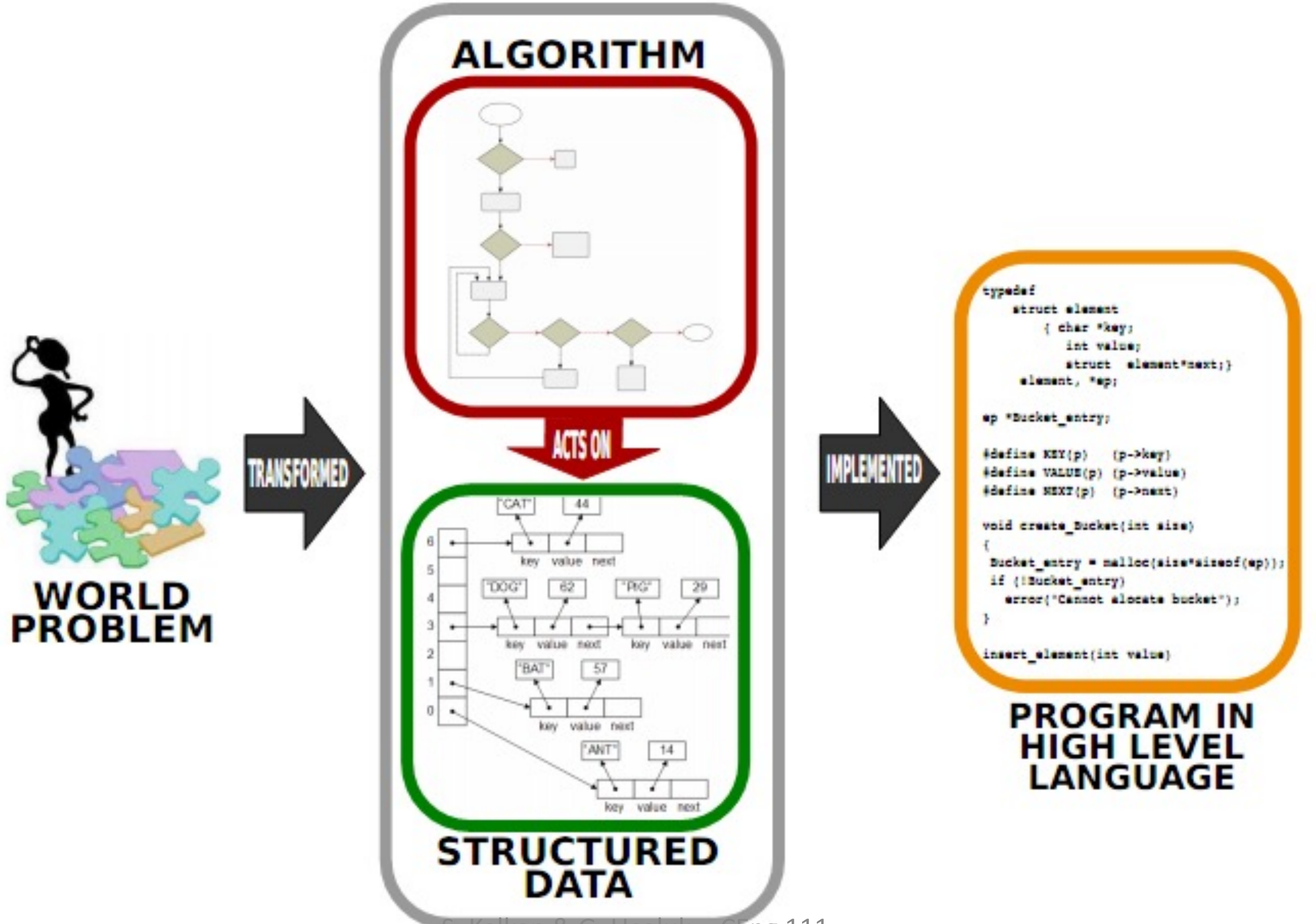
r Engineering

```
def f(List):
    length = len(List)
    changed = True
    while changed:
        changed = False
        i = 0
        while i < length-1:
            if List[i] > List[i+1]:
                (List[i], List[i+1]) = (List[i+1], List[i])
                changed = True
            i += 1
    return List
```

Exercises on Complexity

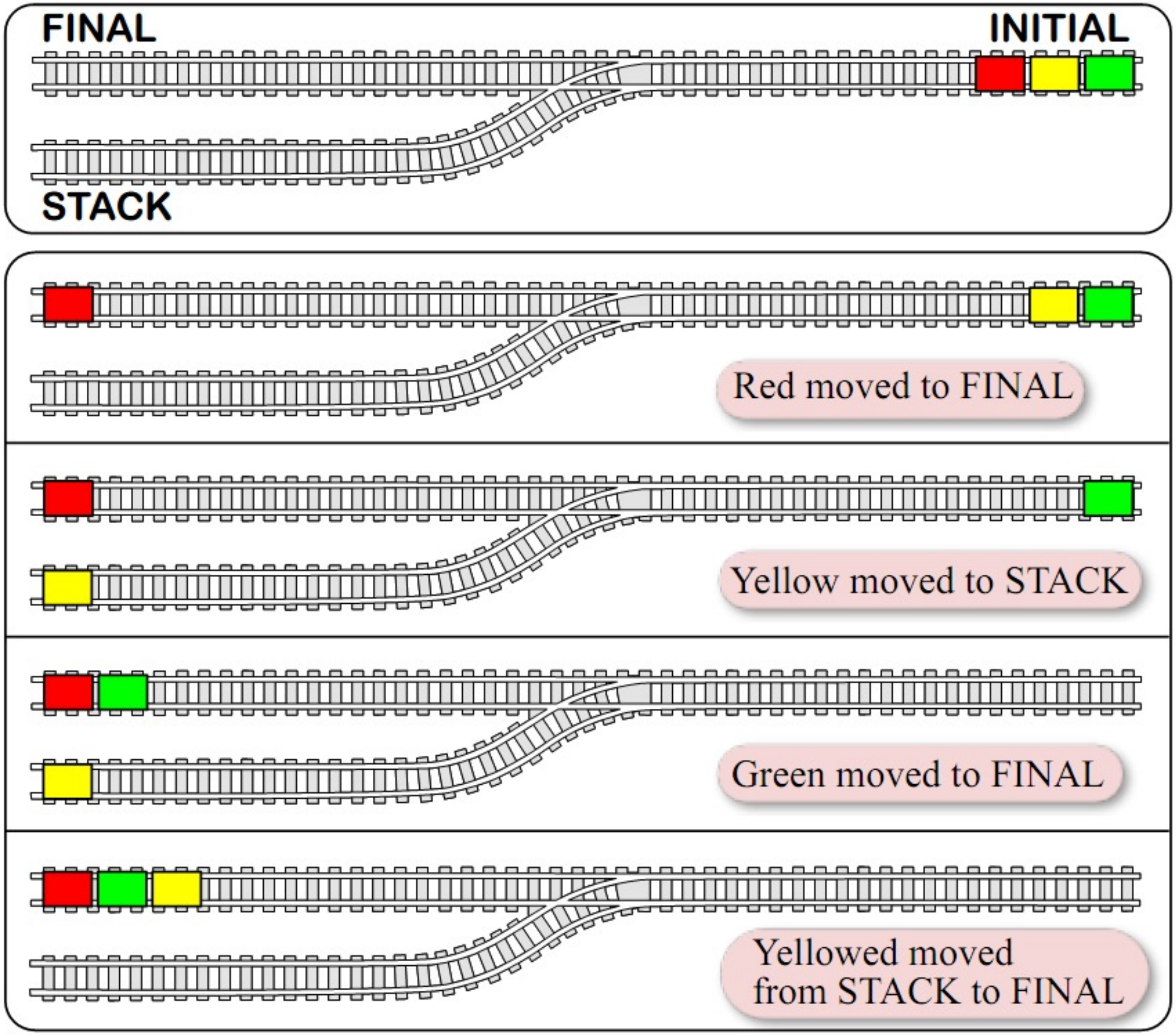
1. What is the complexity of the following?
   - Finding the minimum or the maximum in a list, which is (a) sorted or (b) unsorted.
   - Finding the average of numbers in a list.
2. Assume that we have a sorted list L.
   - What is the complexity of sorting L after inserting a new number?
3. What is the complexity of checking whether a list is sorted?

Design of a solution



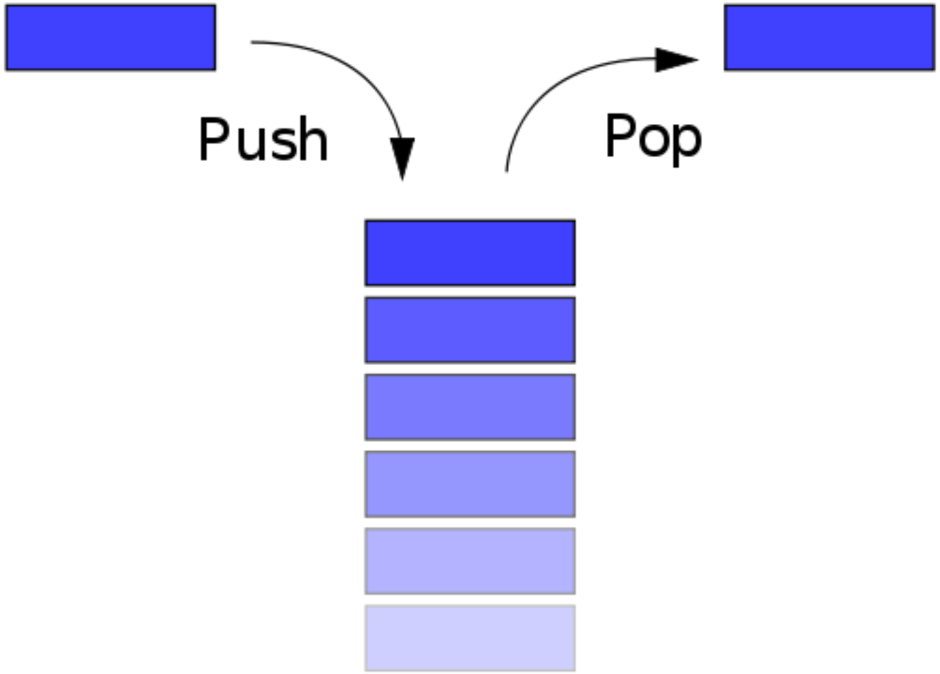
ABSTRACT DATA TYPES

Remember Stacks?



Stacks

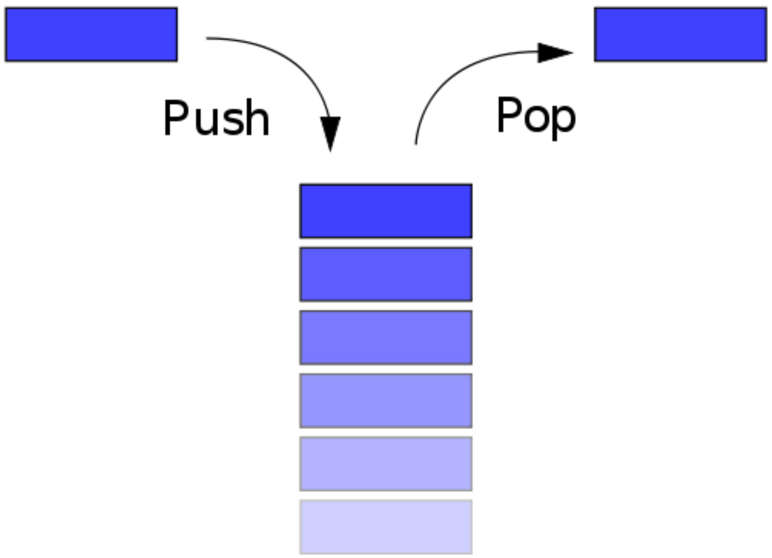
- LIFO:
  - Last In First Out
- We have seen it before (in the Shunting-Yard algorithm)
- Main operations:
  - Push
  - Pop



METU Computer Engineering

Stacks (cont'd)

- Operations:
  1. Push
  2. Pop
  3. Top/Peek
     - Get the top element without removing it
  4. Is-Empty
     - Checks whether the stack is empty
  5. Length
     - # of elements



METU Computer Engineering

Stacks: Formal definition

$$push(item, stack) \Rightarrow item \odot stack$$

- $new() \rightarrow \emptyset$
- $popoff(\xi \odot S) \rightarrow S$
- $top(\xi \odot S) \rightarrow \xi$
- $isempty(\emptyset) \rightarrow$ TRUE
- $isempty(\xi \odot S) \rightarrow$ FALSE

Stacks in Python

Stack Operation	**Corresponding Python Op.**
Pop	L.pop()
Push	L.append(item)
Top/Peek	L[-1]
Is-Empty	L == []
Length	len(L)

Implementing Stacks in Python

```
def CreateStack():
    """Creates an empty stack"""
    return []

def Push(item, Stack):
    """Add item to the top of Stack"""
    Stack.append(item)

def Pop(Stack):
    """Remove and return the item at the top of the Stack"""
    return Stack.pop()

def Top(Stack):
    """Return the value of the item at the top of the
       Stack without removing it"""
    return Stack[-1]

def IsEmpty(Stack):
    """Check whether the Stack is empty"""
    return Stack == []
```