EEL 4837
Programming for Electrical Engineers II

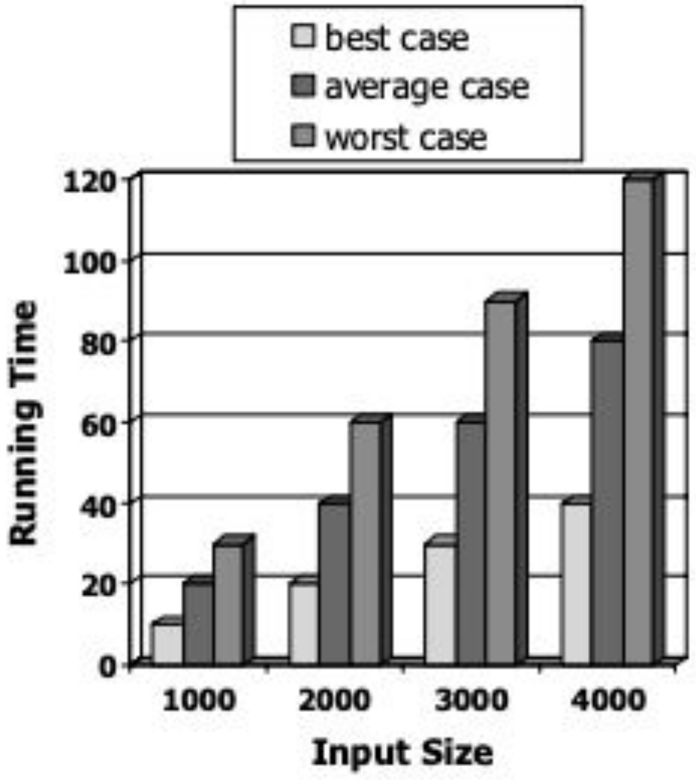
Ivan Ruchkin
Assistant Professor
Department of Electrical and Computer Engineering
University of Florida at Gainesville
iruchkin@ece.ufl.edu
http://ivan.ece.ufl.edu

Time complexity and Big-Oh notation

Readings:
- Weiss chapter 2
- Horowitz 1.3

Program running time

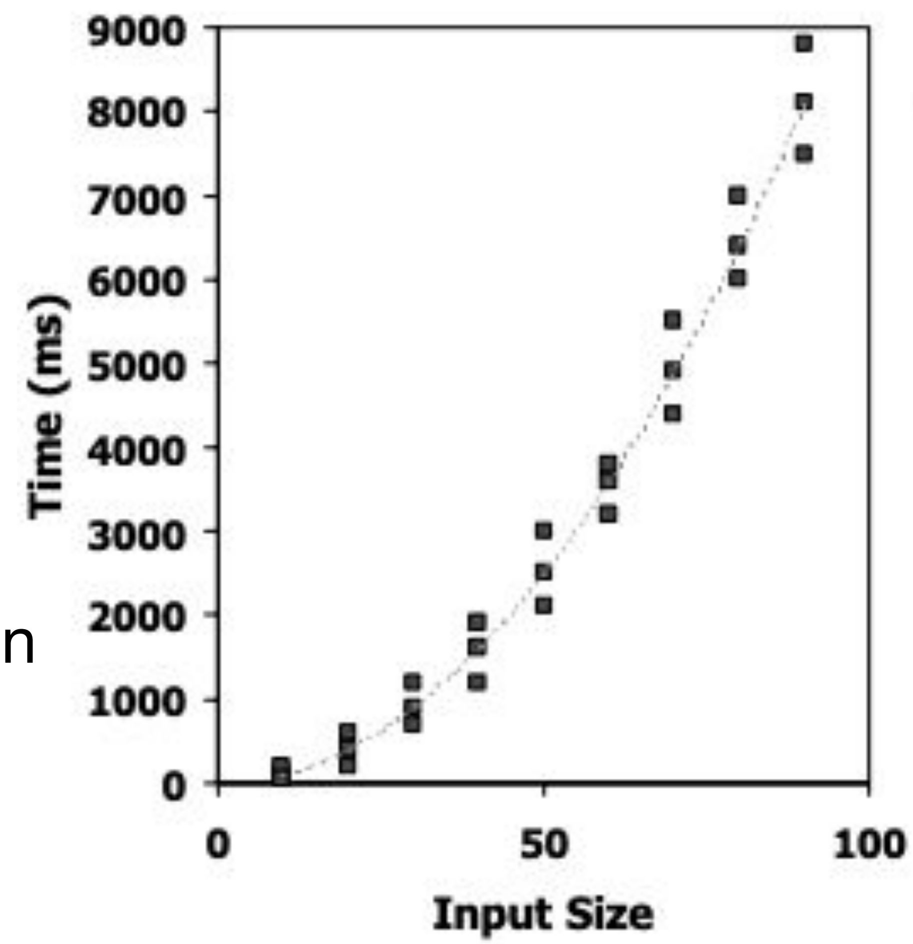
- Most algorithms transform input objects into output objects.
- The running time of an algorithm typically grows with the input size.



- Generally difficult to measure average case
- We typically focus on worst-case analysis (particularly critical for applications like automotive, hardware design, games, etc.)

Measuring program running time

- Write a program.
- Run the program with inputs of varying size.
- "Accurately" measure the running time (e.g., using Unix commands to get wall clock time, user time, system time, kernel time, etc.)
- Plot the results



- May not be indicative of the behavior of the program on other inputs
  - Relies too much on the hardware specifics
- In order to compare two algorithms we have to use same hardware/software environments

Pseudocode

- High-level description of algorithm
- More structured than English prose
- Less detailed than program code
- Preferred notation for describing algorithms
- Hides programming syntax

Example: find max element of an array

Algorithm ***arrayMax***(A, n)
  Input array A of n integers
  Output maximum element of A

  $currentMax \leftarrow A[0]$
  for $i \leftarrow 1$ **to** $n - 1$ **do**
    if $A[i] > currentMax$ **then**
      $currentMax \leftarrow A[i]$
  return $currentMax$

Primitive operations

- Basic computations performed by algorithm
- Identifiable in pseudocode
- Largely independent of programming language
- Assumed to take constant time to compute

- Evaluating an expression
- Assigning a value to a variable
- Indexing into an array
- Calling a method
- Returning from a method

Primitive Operations

By inspecting the pseudocode, we determine the maximum number of primitive operations executed by an algorithm, as a function of input size

Algorithm *arrayMax*(A, n)	# operations
currentMax $\leftarrow A[0]$	2
for $i \leftarrow 1$ **to** $n - 1$ **do**	$2n$
if $A[i] >$ *currentMax* **then**	$2(n - 1)$
currentMax $\leftarrow A[i]$	$2(n - 1)$
{ increment counter i }	$2(n - 1)$
return *currentMax*	1
Total	$8n - 3$

a = Time taken by the fastest primitive operation

b = Time taken by the slowest primitive operation

Let $T(n)$ be worst-case time of ***arrayMax***. Then

$$a\,(8n - 3) \leq T(n) \leq b(8n - 3)$$

“worst-case” in terms of **operation counts**, not **time**

Linear growth of T(n) is an intrinsic property of algorithm arrayMax.
Changing hardware or software will not change this property.

Big-Oh Notation

f(n) = O(g(n)) means there are positive constants c and n_0, such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$. The values of c and n_0 must be fixed for the function f and must not depend on n.

Example: $2n + 10$ is $O(n)$

- $2n + 10 \leq cn$
- $(c - 2)\, n \geq 10$
- $n \geq 10/(c - 2)$
- Pick $c = 3$ and $n_0 = 10$

Example: the function n^2 is not $O(n)$

- $n^2 \leq cn$
- $n \leq c$
- The above inequality cannot be satisfied since c must be a constant

Big-Oh Examples

$f(n) = O(g(n))$ means there are positive constants c and n_0, such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0$. The values of c and n_0 must be fixed for the function f and must not depend on n.

- 7n-2

  7n-2 is O(n)
  need $c > 0$ and $n_0 \ge 1$ such that $7n-2 \le c \cdot n$ for $n \ge n_0$
  this is true for $c = 7$ and $n_0 = 1$

- $3n^3 + 20n^2 + 5$

  $3n^3 + 20n^2 + 5$ is $O(n^3)$
  need $c > 0$ and $n_0 \ge 1$ such that $3n^3 + 20n^2 + 5 \le c \cdot n^3$ for $n \ge n_0$
  this is true for $c = 4$ and $n_0 = 21$

- 3 log n + 5

  3 log n + 5 is O(log n)
  need $c > 0$ and $n_0 \ge 1$ such that $3 \log n + 5 \le c \cdot \log n$ for $n \ge n_0$
  this is true for $c = 8$ and $n_0 = 2$

Some Rules

- If is $f(n)$ a polynomial of degree d, then $f(n)$ is $O(n^d)$, i.e.,
  1. Drop lower-order terms
  2. Drop constant factors
- Use the smallest possible class of functions
  - Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"
- Use the simplest expression of the class
  - Say "$3n + 5$ is $O(n)$" instead of "$3n + 5$ is $O(3n)$"

Big-Oh and Growth Rate

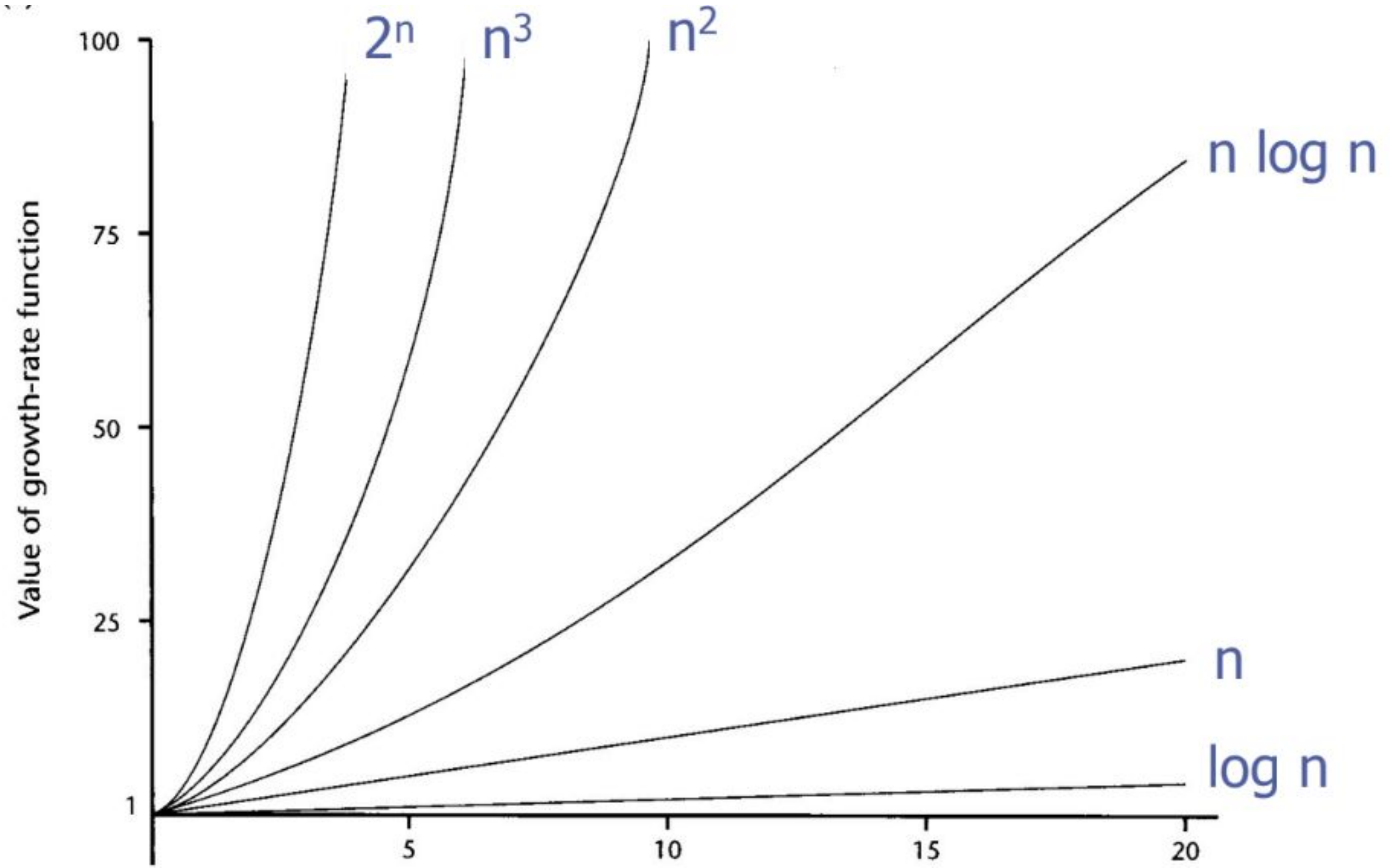
- Big-Oh gives an upper bound on the growth rate of a function
- The statement ``f(n) is O(g(n))" means that the growth rate of f(n) is no more than the growth rate of g(n)
- We can use Big-Oh to rank functions according to their growth rates

Math you should review

- Seven functions that often appear in algorithm analysis:
  - Constant ≈ 1
  - Logarithmic ≈ $\log n$
  - Linear ≈ n
  - N-Log-N ≈ $n \log n$
  - Quadratic ≈ n^2
  - Cubic ≈ n^3
  - Exponential ≈ 2^n

- Summations
- Logarithms and Exponents
  - **properties of logarithms:**
    $\log_b(xy) = \log_b x + \log_b y$
    $\log_b (x/y) = \log_b x - \log_b y$
    $\log_b x^a = a\log_b x$
    $\log_b a = \log_x a / \log_x b$
  - **properties of exponentials:**
    $a^{(b+c)} = a^b a^c$
    $a^{bc} = (a^b)^c$
    $a^b / a^c = a^{(b-c)}$
    $b = a^{\log_a b}$
    $b^c = a^{c*\log_a b}$

Growth rates



Big-Oh and Asymptotic Analysis

- The asymptotic analysis of an algorithm determines the running time in big-Oh notation
- To perform the asymptotic analysis
  - We find the worst-case number of primitive operations executed as a function of the input size
  - We express this function with big-Oh notation
- Example:
  - We determine that algorithm ***arrayMax*** executes at most $8n - 2$ primitive operations
  - We say that algorithm ***arrayMax*** "runs in $O(n)$ time"
- Since constant factors and lower-order terms are eventually dropped anyhow, we can disregard them when counting primitive operations

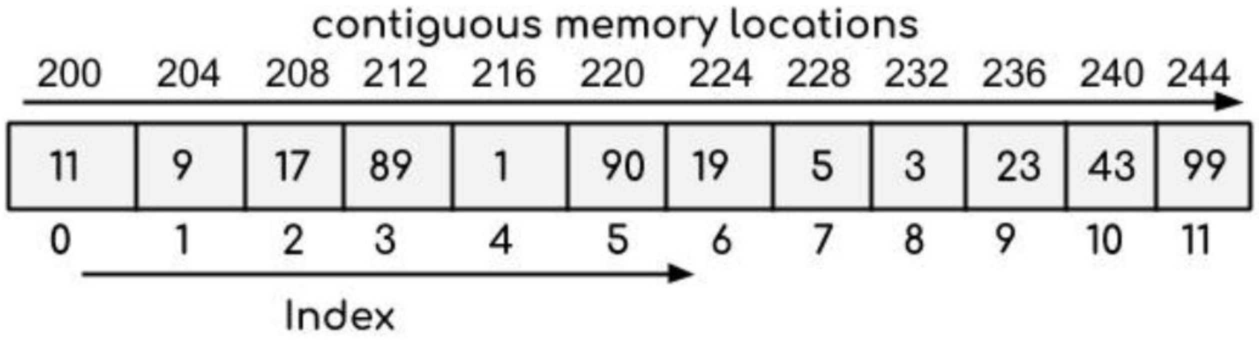
Essentially, this analysis seeks the **smallest, most minimal** function g(n) such that the T(n) = O(g(n))

Algorithm Example: Linear Search

```
int lSearch(int arr[], int n, int x)
{
    int i;
    for (i = 0; i < n; i++)
        if (arr[i] == x)
            return i;
    return -1;
}

int main()
{
    int arr[] = { 3, 4, 1, 7, 5 };
    int n = 5;
    int x = 4;
    int index = lSearch(arr, n, x);
    if (index == -1)
        cout << "Element is not present in the array” << endl;
    else
        cout << "Element found at position " << index << endl;

    return 0;
}
```



Is the number 26 in the array?

Time complexity: O(n)

Searching a Sorted Array

2	5	10	12	15	20	25	31	40
0	1	2	3	4	5	6	7	8

Is the number 26 in the array?

```
int binarySearch(int arr[], int left, int right, int x) {
  while (left <= right) {
     int mid = left + (right - left) / 2;
     if (arr[mid] == x)  return mid;
     else if (arr[mid] < x) left = mid + 1;
     else right = mid - 1;
  }
  return -1;
}
```

- **Number of array indices considered is halved in each iteration**
  - After k iterations, the number of indices considered will be $n/2^k$
- Algorithm stops when indices run out: $n/2^k \leq 1$
- So we choose the *smallest* k such that $n/2^k = 1$
  - $n = 2^k$
  - $k = O(\log n)$

Is binary search a more efficient algorithm than linear search?

Relatives of Big-Oh

- **big-Omega**
  - f(n) is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c \cdot g(n)$ for $n \geq n_0$
- **big-Theta**
  - f(n) is $\Theta(g(n))$ if there are constants $c' > 0$ and $c'' > 0$ and an integer constant $n_0 \geq 1$ such that $c' \cdot g(n) \leq f(n) \leq c'' \cdot g(n)$ for $n \geq n_0$

Example Uses of Ω and Θ

- **$5n^2$ is $\Omega(n^2)$**

  $f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c \cdot g(n)$ for $n \geq n_0$

  let $c = 5$ and $n_0 = 1$

- **$5n^2$ is $\Omega(n)$**

  $f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c \cdot g(n)$ for $n \geq n_0$

  let $c = 1$ and $n_0 = 1$

- **$5n^2$ is $\Theta(n^2)$**

  $f(n)$ is $\Theta(g(n))$ if it is $\Omega(n^2)$ and $O(n^2)$. We have already seen the former, for the latter recall that $f(n)$ is $O(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \leq c \cdot g(n)$ for $n \geq n_0$

  Let $c = 5$ and $n_0 = 1$

Space complexity

- The **amount of extra space** the program needs
  - Can be in abstract “memory units” or bytes
- Same ideas as with the time complexity:
  - Typically focus on the worst-case scenario
  - Count the required memory
  - Use the Big-O notation to express the complexity class
- Example:
  - Function `dedup(int[] arr, int n)` removes duplicates
  - It allocates another array of size n to store the results
  - Its space complexity: O(n)

Abstract data type: Stack

Readings:

- Weiss 3.6
- Horowitz 2.1

Why So Many Data Structures?

Ideal data structure:

fast, elegant, memory efficient

Generates tensions:

- time *vs.* space
- performance *vs.* elegance
- generality *vs.* simplicity
- one operation's performance *vs.* another's

A **dictionary** is implementable with:

- array
- list
- binary search tree
- AVL tree
- Splay tree
- Red-Black tree
- hash table

How Should We Learn Data Structures?

- Present a data structure
- Motivate with some applications
- Repeat until browned entirely through
  - develop a way to implement the data structure
  - analyze its properties
    - efficiency
    - correctness
    - limitations
    - ease of programming
- Contrast data structure's strengths and weaknesses
  - understand when to use each one

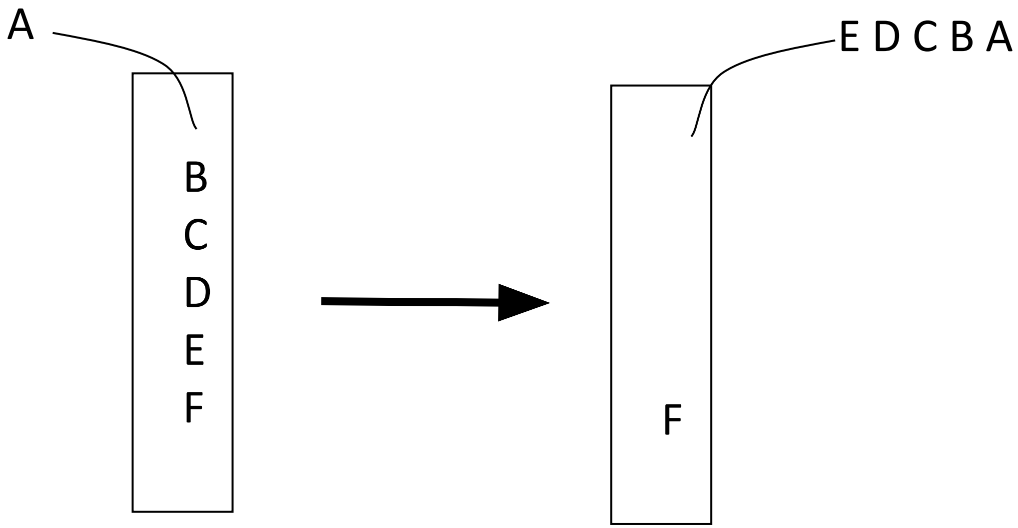
Abstract data types

- **Idea:** define a data structure by specifying what functions it supports
- Separates the interface from the implementation
  - Implementation details are hidden
  - The “client” code can be used with any implementation

An **abstract data type** is a mathematical representation of a data structure with a *set of values* and a *set of operations* supported by these values

Stack

- **Stack operations**
  - create
  - destroy
  - push
  - pop
  - is_empty
  - Is_full (???)



- *Stack property* – **LIFO (Last In First Out):** if x is on the stack before y is pushed, then x will be popped after y is popped

Implementing a Stack with arrays

- A simple way of implementing the Stack ADT uses an array
- We add elements from left to right
- A variable keeps track of the index of the top element

Stack Implementation

```
#define SIZE 5
using namespace std;

class STACK
{
    private:
        int num[SIZE];
        int top;
    public:
        STACK();
        int push(int);
        int pop();
        int isEmpty();
        int isFull();
};
```

```
int STACK::isEmpty(){
    if(top==-1)
        return 1;
    else
        return 0;
}
```

```
int STACK::isFull(){
    if(top==(SIZE-1))
        return 1;
    else
        return 0;
}
```

```
int STACK::push(int n){
    //check stack is full or not
    if(isFull()){
        return 0;
    }
    ++top;
    num[top]=n;
    return n;
}

int STACK::pop(){
    //to store and print which number
    //is deleted
    int temp;
    //check for empty
    if(isEmpty())
        return 0;
    temp=num[top];
    --top;
    return temp;

}
```

Performance and Limitations

- Performance
  - Let n be the number of elements in the stack
  - The space used is $O(n)$
  - Each operation runs in time $O(1)$
- Limitations
  - The maximum size of the stack must be defined a priori and cannot be changed
  - Trying to push a new element into a full stack causes an implementation-specific exception

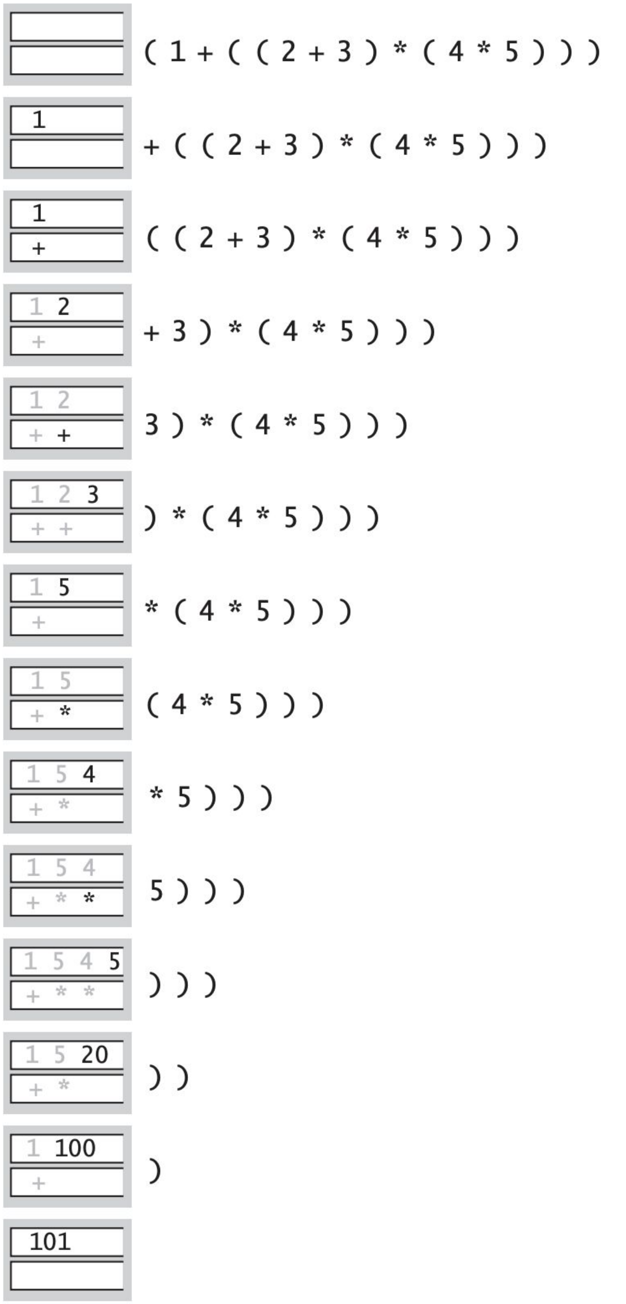
Evaluating (infix) Expressions

(1 + ((2 + 3) * (4 * 5)))

Two-stack algorithm. [E. W. Dijkstra]

- Value: push onto the value stack.
- Operator: push onto the operator stack.
- Left parenthesis: ignore.
- Right parenthesis: pop operator and two values; push the result of applying that operator to those values onto the operand stack.

Exercise: Implement this algorithm in C++



Postfix Expressions

- Postfix notation is another way of writing arithmetic expressions
- In postfix notation, the operator is written after the two operands

  infix: 2+5 *postfix*: 2 5 +

  infix (3+ ((4 + 5) * 2)) *postfix: 3 4 5 + 2 * +*
- Expressions are evaluated from left to right
  - Pushing operands into a stack
  - Pop and execute operations
- Precedence rules and parentheses are never needed!

Exercises:

1. Convert a (fully parenthesized) infix expression to postfix
2. Evaluate a postfix expression