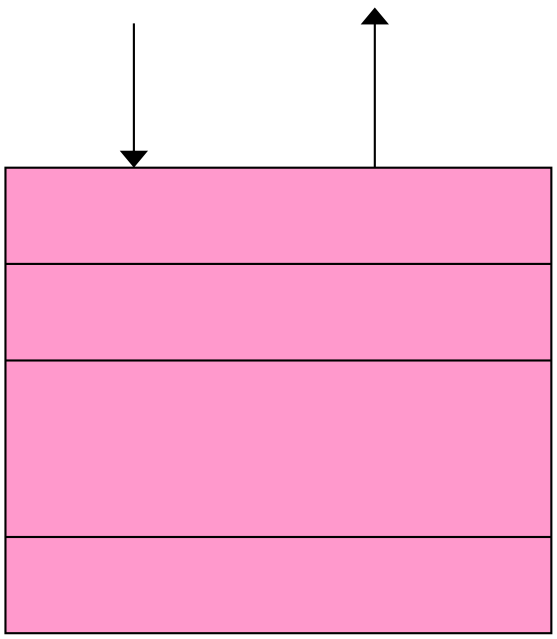
CS3334 Data Structures
Lec-3 Program Complexities

Review: Objective of Lec 2

- Stack Abstract Data Type
- Sequential Allocation
- Linked Allocation
- Applications

Review: Stack

- Stack is a list with the restriction that insertions and deletions (usually all the accesses) can only be performed at one end of the list
- Also known as: Last-in-first-out (LIFO) list



Review: ADT of Stack

Value:

A sequence of items that belong to some data type ITEM_TYPE

Operations for a stack *s*:

1. Boolean IsEmpty()
   Postcondition: If the stack is empty, return true, otherwise return false
2. Boolean IsFull()
   Postcondition: If the stack is full, return true, otherwise return false
3. ITEM_TYPE Pop() /*take away the top one and return its value*/
   Precondition: s is not empty
   Postcondition: The top item in s is removed from the sequence and returned
4. ITEM_TYPE top() /*return the top item's value*/
   Precondition: s is not empty
   Postcondition: The value of the top item in s is returned
5. Void Push(ITEM_TYPE e) /*add one item on top of the stack*/
   Precondition: s is <u>not full</u>
   Postcondition: e is added to the sequence as the top one

Review: Array Implementation of Stack

```
// MyStack.h
#include “stdlib.h”
{
	public class MyStack
	{
		public:
				MyStack( int );
				bool IsEmpty();
				bool IsFull();
				void push(int );
				int pop();
				int top();
		private:
				int* data;
				int top;
				int MAXSize;
	};
}
```

```
// MyStack.cpp

#include “MyStack.h”
MyStack::MyStack(int size)
{
		data=new int[size];
		top=-1;
		MAXSize=size;
}
bool MyStack::IsEmpty()
{
		return (top==-1);
}
bool MyStack::IsFull()
{
		return (top==MAXSize-1);
}
```

Review: Array Implementation of Stack: push

```
. . .
private:
      int* data;
      int top;
      int MAXSize;

void MyStack::push(int x)
{
      if (!IsFull() )
      {
            top=top+1;
            data[top] = x;
      }
      else
                  ....
}
```

To "push" an item onto the stack

- Check whether not yet full.
- Increase the top indicator (slot number) of the stack.
- Copy the item to the top position immediately.

Slot #0: filled
Slot #1: filled
Slot #2: filled
Slot #3: to be filled
Slot #4: not yet filled

...

Slot #99: not yet filled

Top of stack: slot #2 => 3

Review: Array Implementation of Stack: pop

```
. . .
private:
    int* data;
     int top;
     int MAXSize;


int MyStack::pop( )
{   int rtn_value;
    if (!IsEmpty())
    {
       rtn_value=data[top];
       top=top-1;
       return rtn_value;
    }
    else
    …
}
```

To "pop" an item from the stack (to take away the top one and return its value)

- Check whether it is empty.
- Save the value of item at the top position (to return it later)
- Decrease the top indicator (slot #)
- Return the saved value.
- *No need to clear any slot.*

Slot #0: filled
Slot #1: filled
Slot #2: filled
Slot #3: to be popped
Slot #4: not yet filled

…

Slot #99: not yet filled

Top of stack: slot #3 => 2

Review: Array Implementation of Stack: top

```
. . .
private:
      int* data;
        int top;
        int MAXSize;

int MyStack::top( )
{
      if (!IsEmpty())
      {
               return (data[top]);
      }
      else
      . . .
}
```

To return the value of an item from the stack (the top item)

- Check whether it is empty.
- Return the value of the item at the top position.

Slot #0: filled
Slot #1: filled
Slot #2: filled
Slot #3: to be returned
Slot #4: not yet filled
…
Slot #99: not yet filled

Top of stack: slot #3 (no change)

Review: Stacks: Use Dynamic Array

- How to choose the size of array data[]?
  - As we insert more and more, eventually the array will be full

- Solution: Use a dynamic array
  - Maintain capacity of data[]
  - Double capacity when size=capacity (i.e. full)
  - Half capacity when size $\leq$ capacity/4

- Question: What if we change capacity/4 to capacity/2 ?
  - E.g., initial cap is 4; I, I, I, I, I (expand; cap=8, size=5), D (shrink; cap=4, size=4), I (expand; cap=8, size=5), D (shrink; cap=4, size=4), I (expand), D (shrink), ….

Review: Stacks: Another implementation

```
class Stack
{
      public:
        Stack(int initCap=100);
        Stack(const Stack& rhs);
        ~Stack();

        void push(Item x);
        void pop(Item& x);

      private:
        void realloc(int newCap);
        Item* data;
        int size;
        int cap;
};
```

```
// An internal func. to support resizing of array
void Stack::realloc(int newCap) {
      if (newCap < size) return;
      //oldarray "point to" data
      Item *oldarray = data;

      //create new space for data with size newCap
      data = new Item[newCap];
      for (int i=0; i<size; i++)
            data[i] = oldarray[i];
      cap = newCap;
      delete [] oldarray;
}

void Stack::push(Item x) {
      if (size==cap)  realloc(2*cap);
      array[size++]=x;
}
```

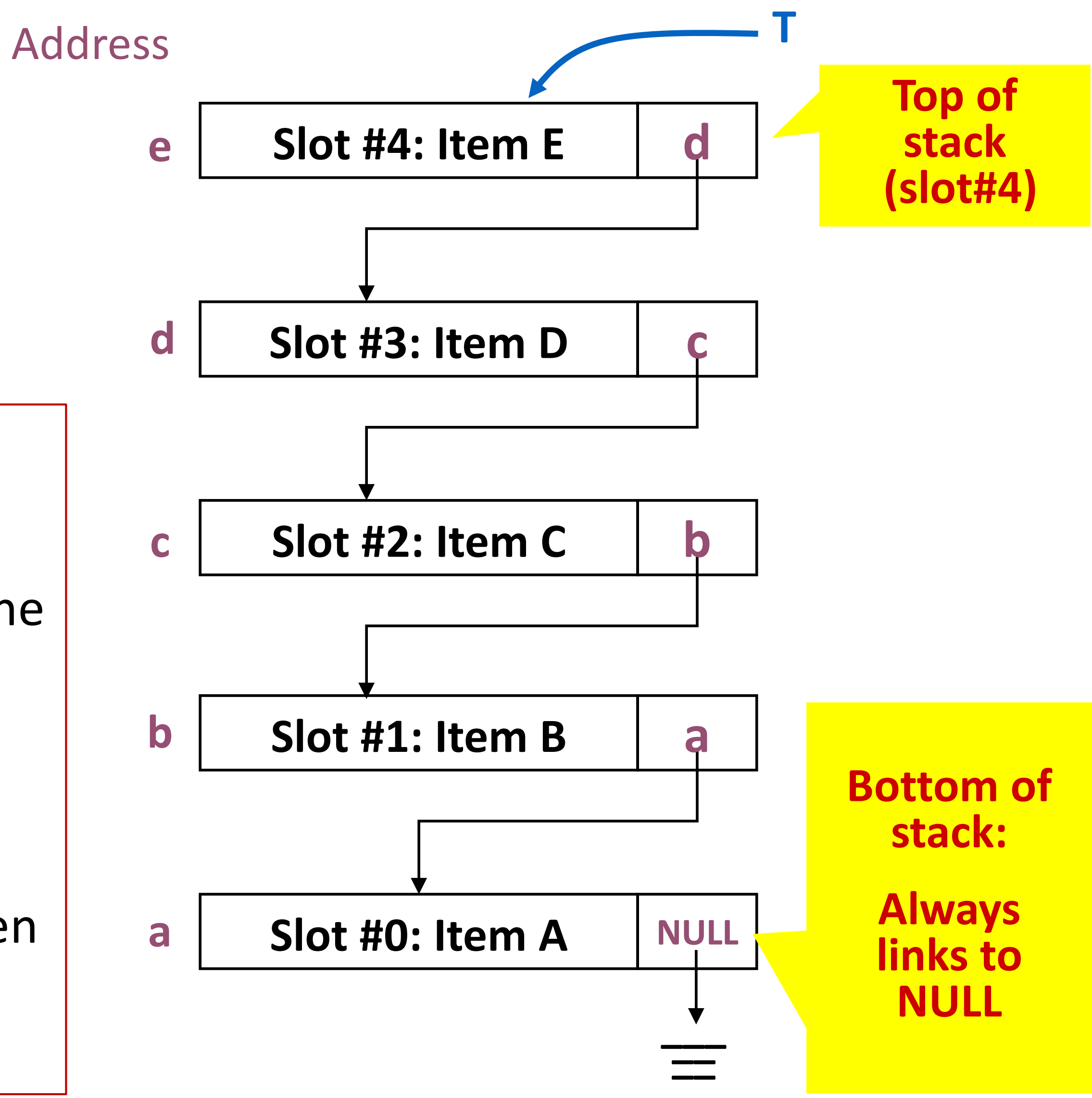
Review: Stacks: Another implementation

```
void Stack::pop(Item& x)
{
    // assume EmptyStack is a special value
    if (size==0)
        x=EmptyStack;
    else
    {
        x=array[--size];
        if (size <= cap/4)
            realloc(cap/2);
    }
}
```

Review: Linked Implementation of Stack

Stack can also be implemented with **linked list**.

- Typically, a pointer points to the top of the stack. (T)
- When the stack is empty, this pointer will be NULL.
- Each slot is allocated only when it is needed to store an item.



Review: Linked Implementation of Stack

```
// MyStack.h


#include “stdlib.h”
#include “ListNode.h”
{
    class MyStack
    {
    public:
        MyStack( );
        Pop();
        IsEmpty();
        Push(int );
        …
    private:
        ListNode *Top;
    };
}
```

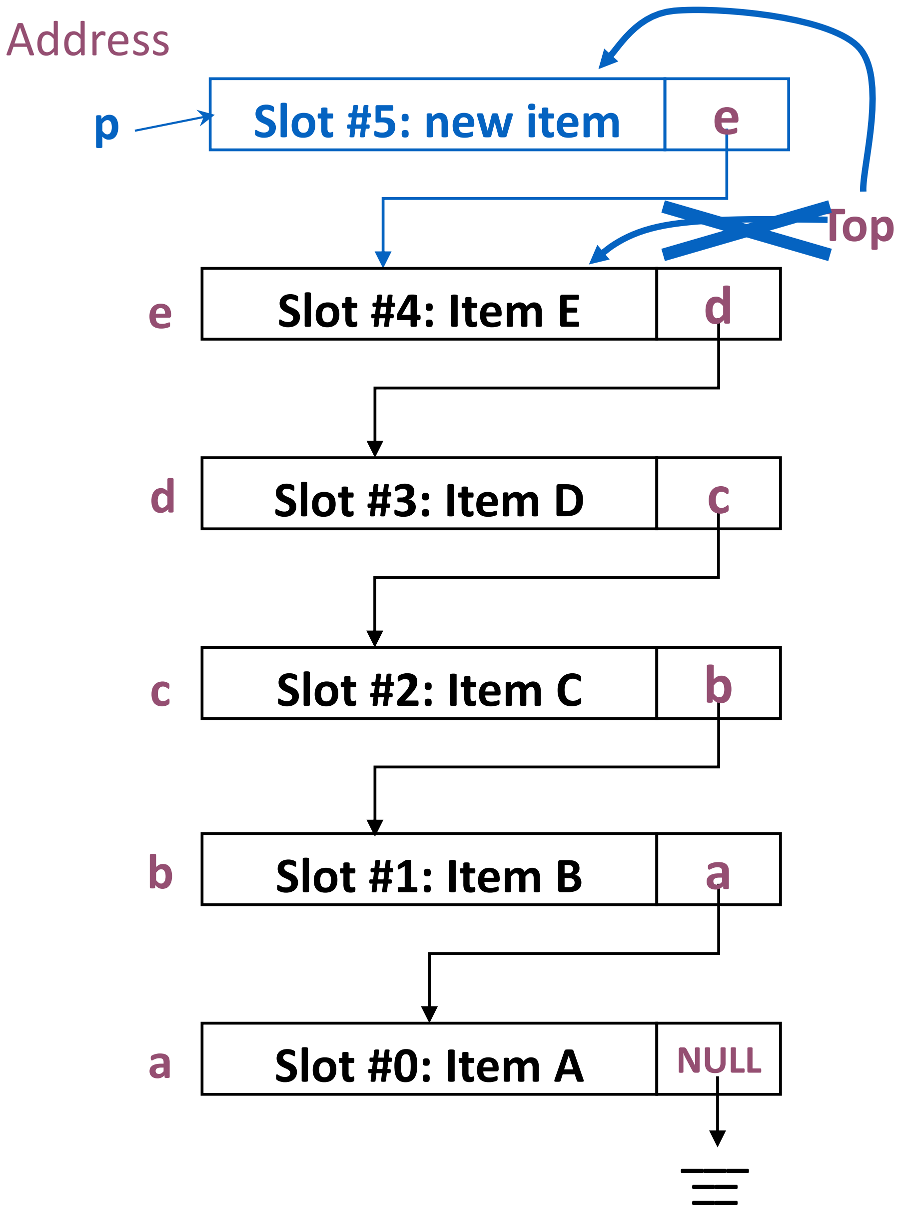
```
// ListNode.h

#include “stdlib.h”
{
    class ListNode
    {
    public:
        ListNode( int );
        ListNode( int, ListNode *);
        ListNode *get_Next()
        {
            return next;
        }
        …
    private:
        int data;
        ListNode *next;
    };
}
```

Review: Linked Implementation of Stack: push

Push: To insert new information onto the top of the stack

- Allocate memory for an auxiliary pointer **p**
- Put new item into p->data
- p->next = T
- T=p

```
void MyStack::Push (int new_item)
{
    ListNode* p;
    p=new ListNode(new_item, Top);
    // p->data = new_item;
    // p->next = Top;
    Top = p;
}
```

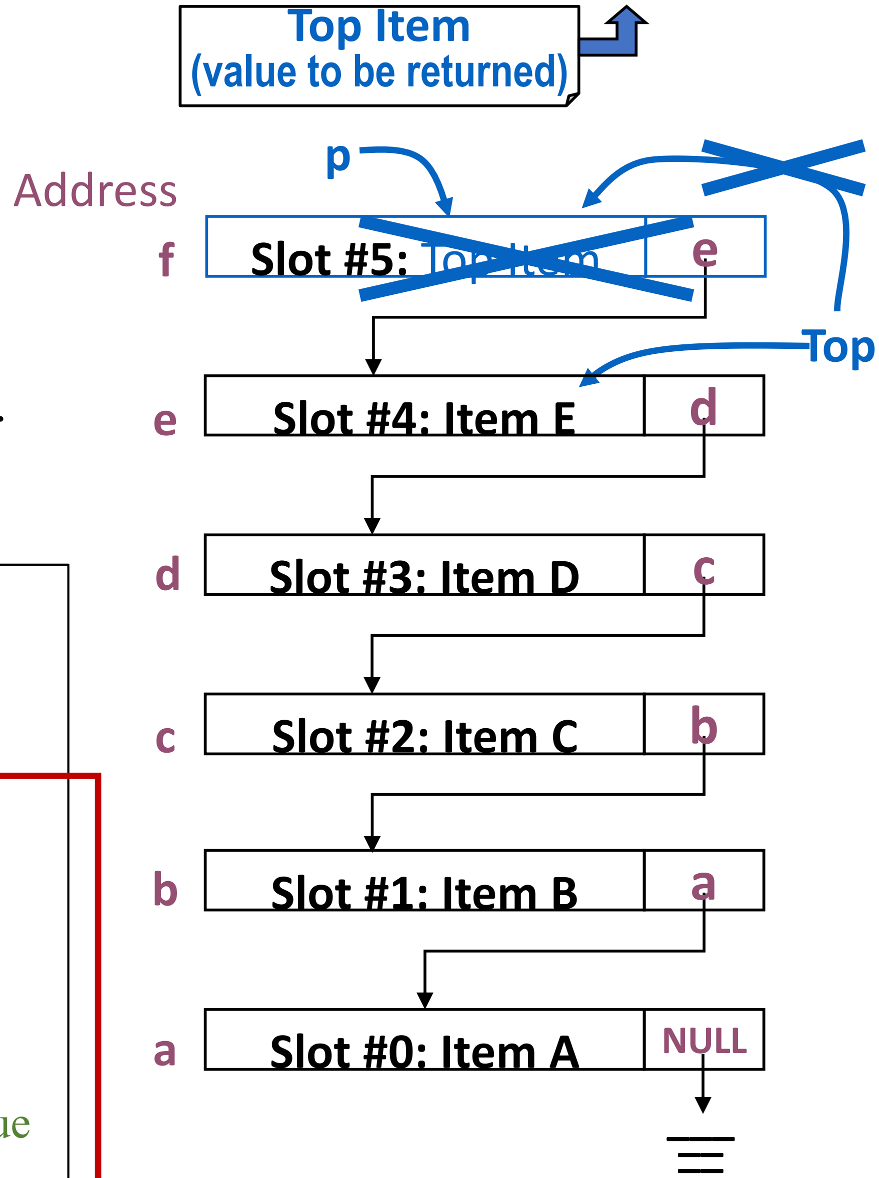


Review: Linked Implementation of Stack: pop

Pop: To take away (and delete) the top item and return its value.

- Check whether the stack is empty.
- Store the value of the item so that we can return it later.
- Update the T pointer to point to the next item.
- Return the value of the top item.

```
int MyStack::Pop () {
ListNode* p;    //a pointer to point to original top node
int rtn_value;   //the value of the item to be returned
if (IsEmpty())  //check whether the stack is empty
{ //Exception handling }
rtn_value=Top->data;  //save the value to be returned
Top= Top->next;        //update the T pointer
return (rtn_value);      //return the original top node value
}
```



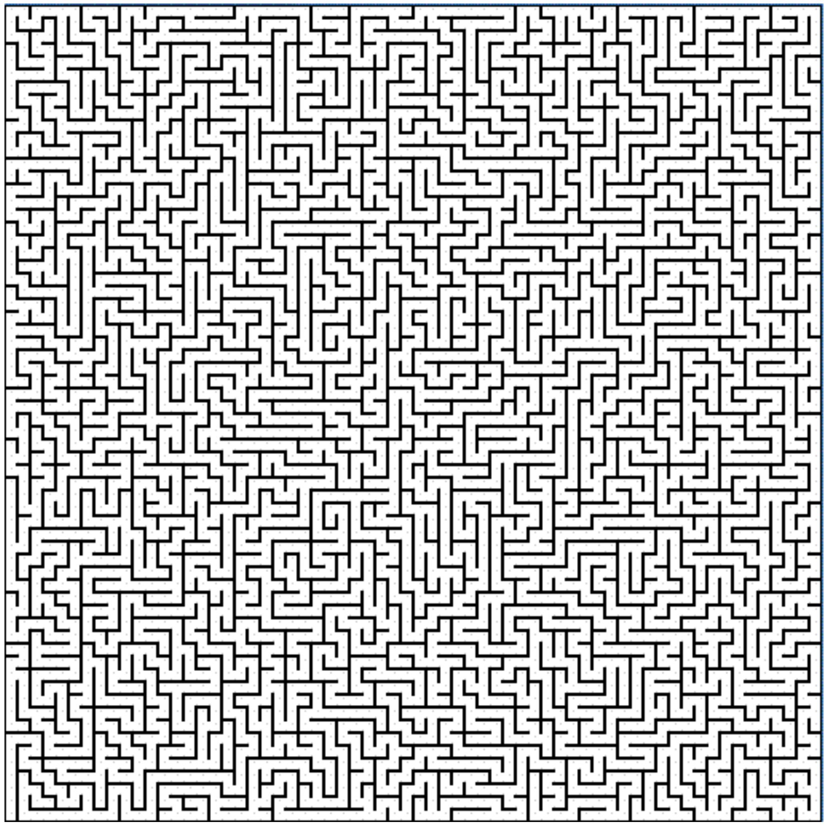
Review: Linked Implementation of Stack: pop

```
int MyStack::Pop () {
        ListNode* p;    //a pointer to point to original top node
        int rtn_value;   //the value of the item to be returned
        if (IsEmpty())  //check whether the stack is empty
        { //Exception handling }
        rtn_value=Top->data;  //save the value to be returned
        ListNode* temp = Top;
        Top= Top->next;        //update the T pointer
        delete temp;
        return (rtn_value);      //return the original top node value
}
```

Review: Application1: Backtracking

Generating a maze

Start (0, 0)



End (width-1, heigh-1)

Using stacks (simplest way)

1. Start from the entrance cell
2. Randomly select an unvisited neighbor cell of the stack top and break the wall, then push the new cell onto the stack
3. If all the neighbors are already visited, then go back by popping cells from the stack
4. Until the exit is reached

Try by yourself on a 4*4 maze!

Review: Application 2: Balancing Symbols

- When writing programs, we use
  - () parentheses [] brackets {} braces
- A lack of one symbol may cause the compiler to emit a hundred lines without identifying the real error
- Using stack to check the balance of symbols
  - [( )] is correct while [( ]) is incorrect

- Read the code until end of file
  - **If** the character is an opening symbol: ([{, **then** push it onto the stack
  - **If** the character is a closing symbol:)] }, **then** pop one (if the stack is not empty) from the stack to see whether it is the correct correspondence
  - Output error in other cases

Review: Application 3 Evaluation of Postfix Expression

- Infix Expression Example: (A+B)*((C-D)*E+F)

  We need to add "(" and ")" in many cases.

- Postfix Expression Example: AB+CD-E*F+*

  Each operator follows the two operands.

  The order of the operators (left to right) determines the actual order of operations in evaluating the expression.

- Prefix expression Example : *+AB+*-CDEF

  Each operator precedes the two operands.

Review: Application 3 Evaluation of Postfix Expression

6 2 + 3 1 - 4 * 7 + *

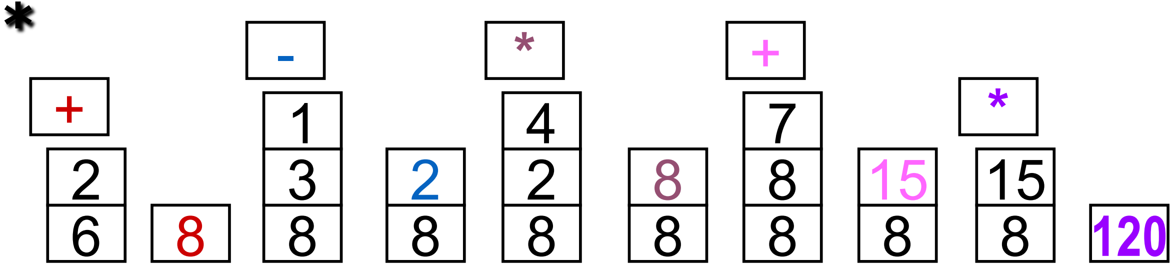
=8 3 1 - 4 * 7 + *

=8 2 4 * 7 + *

=8 8 7 + *

=8 15 *

=120



The method:

- Scan the expression from left to right.
- For each symbol, if it is an operand, we store them for later operation (LIFO) push
- If the symbol is an operator, take out the latest 2 operands stored and compute with the operator. pop pop

  Treat the operation result as a new operand and store it. push
- Finally, we can obtain the result as the only one operand stored. pop

Review: Application 4 Infix expression->postfix expression

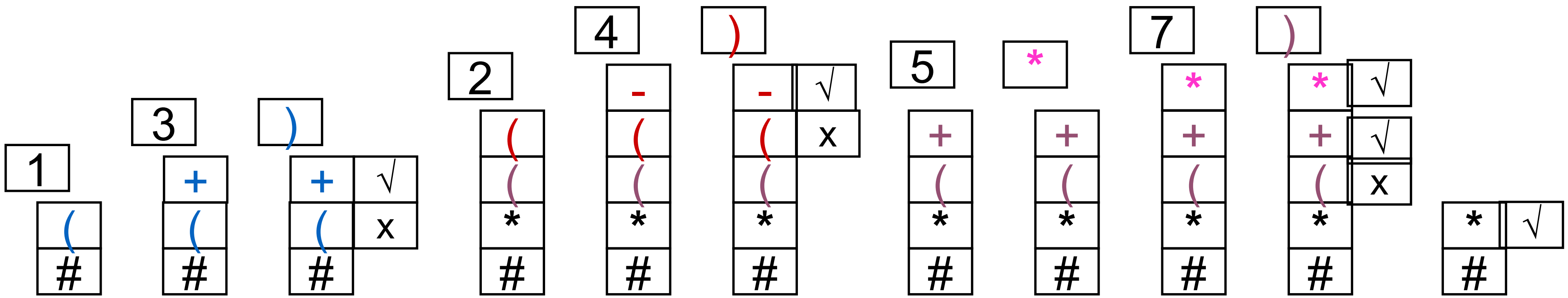
Define the precedence relation of some of the operators:

is the special symbol to denote the bottom of stack.

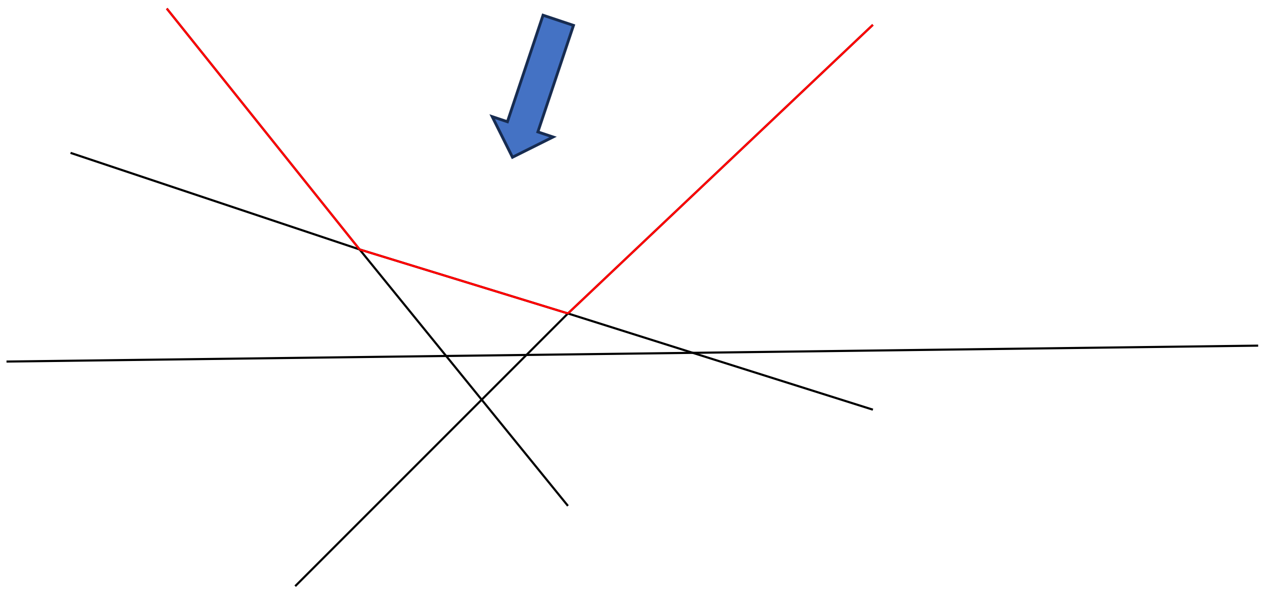
Operators	priority no.
#	0
(	1
+ or -	2
* or /	3

Example:(1+3)*((2-4)+5*7) => 1 3 + 2 4 - 5 7 * + *

1	3	+	2	4	-	5	7	*	+	*

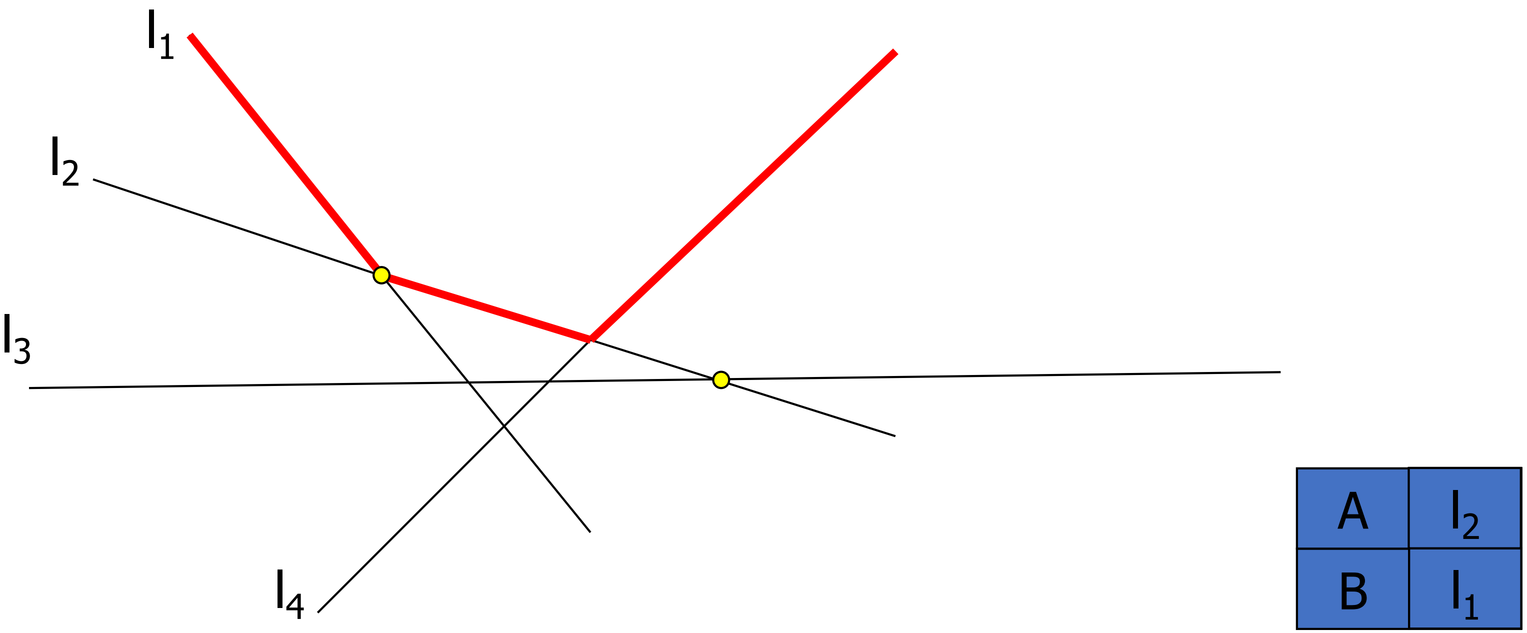


Review: Application 5: Identify the boundary of lines

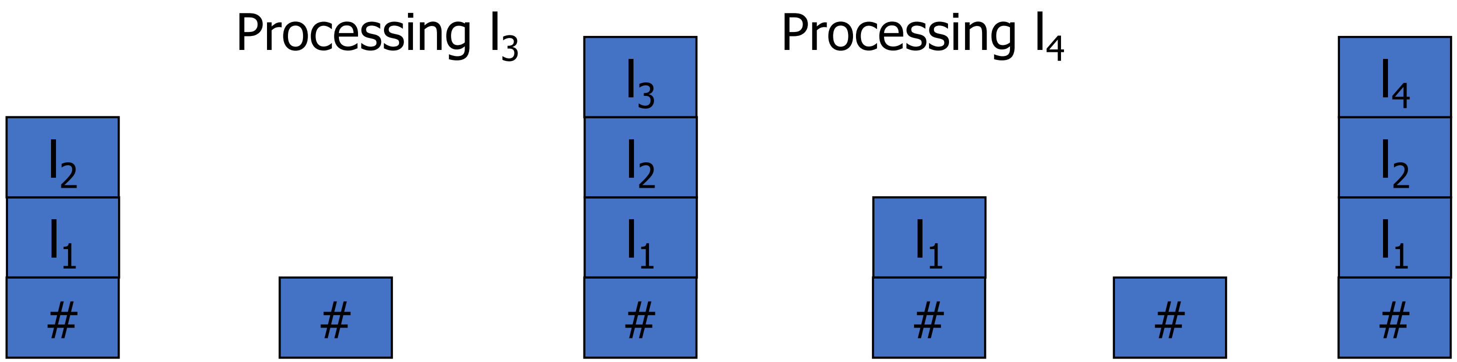


- Given several lines, identify which parts of the lines can been seen if you look from the above

Identify the boundary of lines



- Example:



Exercise 1

Given a balanced expression that can contain opening and closing parenthesis, check if it contains any duplicate parenthesis or not.

Examples:

```
Input:  ((x+y))+z
Output: true

Input:  (x+y)
Output: false
```

```
bool findDuplicateparenthesis(string str)
{
        . . .  // using stack
}
```

Exercise 1

```
bool findDuplicateparenthesis(string str)
{
        stack<char> Stack;
        for (char ch : str) {
                if (ch == ')’) {
                        char top = Stack.pop();
                        int elementsInside = 0;
                        while (top != '(') {
                                elementsInside++;
                                top = Stack. pop();
                        }
                        if(elementsInside < 1)
                                return true;
                }
                else
                        Stack.push(ch);
        }
        return false;
}
```

Exercise 2

Given a non-negative integer num represented as a string, remove *k* digits from the number so that the new number is the smallest possible.

Examples:

```
Input: num = "1432219", k = 3
Output: "1219"

Input: num = "10200", k = 1
Output: "200"
```

```
int removeKdigits(string num, int k)
{
        . . . // using stack
}
```

Exercise 2

```
int removeKdigits(string num, int k) {
  int stringLength = num.size();
  if (stringLength == k)
    return "0";

  stack < char > S;
  int n = k, idx = 0;
  while (idx < stringLength) {
    int currentNumber = num[idx] - '0';
    while(n > 0 && !S.empty() && (S.top()-'0') > currentNumber) {
      n--;
      S.pop();
    }
    S.push(num[idx]);
    idx++;
  }

  while(n>0) {
    S.pop();
    n--;
  }

  string result;
  while (!S.empty())
    result += S.pop();

  reverse(result.begin(), result.end());
  int number = stringToDigit(result);
  return number;
}
```

Objectives

- Algorithms
- Asymptotic Notation
- Asymptotic Performance
- Analyze program complexities

Algorithms

- What is an algorithm?

  A sequence of elementary computational steps that transform the input into the output
- What for?

  A tool for solving well-specified computational problems, e.g., Sorting, Matrix Multiplication
- What do we need to do with an algorithm?
  - Correctness Proof:

    for every input instance, it halts with the correct output
  - Performance Analysis (1 second or 10 years?):

    How does the algorithm behave as the problem size gets large

    both in running time and storage requirement

A Sorting Problem

Input : $\langle a_0, a_1, \ldots, a_{n-1} \rangle$

Output: A permutation (re-ordering) $\langle a'_0, a'_1, \ldots, a'_{n-1} \rangle$ of the input sequence such that $a'_0 \leq a'_1 \leq \ldots \leq a'_{n-1}$

Example:

<22, 51, 34, 44, 67, 11> becomes *<11, 22, 34, 44, 51, 67>*

Insertion Sort

5, 3, 1, 2, 6, 4

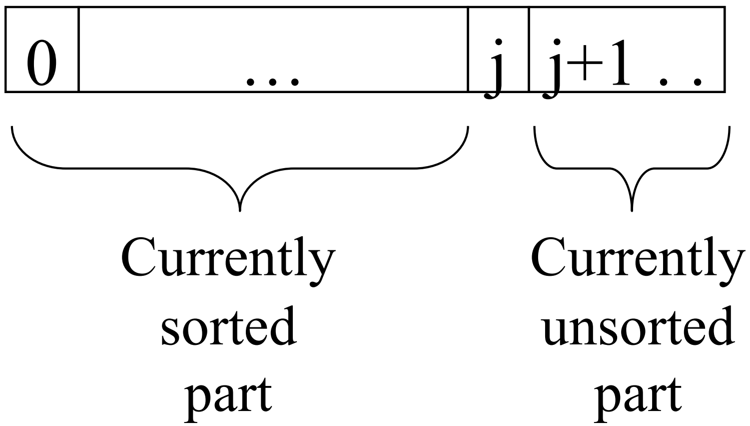
3, 5, 1, 2, 6, 4

1, 3, 5, 2, 6, 4

1, 2, 3, 5, 6, 4

1, 2, 3, 5, 6, 4

1, 2, 3, 4, 5, 6



- To sort *A[0,1,…,n-1]* in place
- Steps:
  - Pick element A[j]
  - Move A[j-1,…,0] to the right until proper position for A[j] is found
- Example 1 3 5 2 6 4

Insertion Sort (cont.)

Insertion-Sort (A)

```
1. for j=1 to n-1
2.      key=A[j]
3.      i=j-1
4.      while i>=0 and A[i]>key
5.              A[i+1]=A[i]
6.              i=i-1
7.      A[i+1]=key
```

	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]
j=1	*5*	3	1	2	6	4
j=2	*3*	*5*	1	2	6	4
j=3	*1*	*3*	*5*	2	6	4
j=4	*1*	*2*	*3*	*5*	6	4
j=5	*1*	*2*	*3*	*5*	*6*	4
j=6	*1*	*2*	*3*	*4*	*5*	*6*

j=3   1 3 5 2 6 4

1 3 5 → 2 6 4

1 3 → 2 5 6 4

1 2 3 5 6 4

Insertion Sort (cont.)

Note that when we are dealing with k^{th} number, the first k-1 numbers are already sorted.
The k^{th} number is inserted in the correct position.

5, 3, 1, 2, 6, 4

<u>3</u>, 5, 1, 2, 6, 4

<u>1</u>, 3, 5, 2, 6, 4

1, <u>2</u>, 3, 5, 6, 4

1, 2, 3, 5, <u>6</u>, 4

1, 2, 3, <u>4</u>, 5, 6

Correctness of Algorithm

- Why can the algorithm correctly sort?
- We only consider algorithms with loops
  - Find a property as loop invariant
- How to show something is loop invariant?
  - Initialization:

    It is true prior to the first iteration of the loop
  - Maintenance:

    If it is true before an iteration, it remains true before the next iteration
  - Termination:

    When the loop terminates, the invariant gives a useful property that helps to show the algorithm is correct

Running time of Insertion Sort

Insertion-Sort(A)

```
1  for j = 1 to n-1
2      key = A[j]
3      i = j-1
4      while i >= 0 and A[i] > key
5          A[i+1] = A[i]
6          i = i - 1
7      A[i+1] = key
```

Cost	**times**
c_1	n
c_2	$n-1$
c_3	$n-1$
c_4	$\Sigma_{j=1..n-1} (t_j+1)$
c_5	$\Sigma_{j=1..n-1} t_j$
c_6	$\Sigma_{j=1..n-1} t_j$
c_7	$n-1$

c_1, c_2, .. = running time for executing line 1, line 2, etc.

t_j = no. of times that line 5,6 are executed, for each j.

The running time $T(n)$

$$= c_1*n+c_2*(n-1)+c_3*(n-1)+c_4*(\Sigma_{j=1..n-1} (t_j+1))+c_5*(\Sigma_{j=1..n-1} t_j)+c_6*(\Sigma_{j=1..n-1} t_j)+c_7*(n-1)$$

Analyzing Insertion Sort

$$T(n) = c_1*n+c_2*(n-1)+c_3*(n-1)+c_4*(\Sigma_{j=1..n-1}(t_j+1))+$$
$$c_5*(\Sigma_{j=1..n-1} t_j)+c_6*(\Sigma_{j=1..n-1} t_j)+c_7*(n-1)$$

Worse case:

Reverse sorted: for example, 6,5,4,3,2,1

→ inner loop body executed for all previous elements.

→ $t_j=j$.

→ $T(n) = c_1*n+c_2*(n-1)+c_3*(n-1)+c_4*(\Sigma_{j=1..n-1}(j+1))+$
$c_5*(\Sigma_{j=1..n-1} j)+c_6*(\Sigma_{j=1..n-1} j)+c_7*(n-1)$

→ $T(n) = An^2+Bn+C$

Note: $\Sigma_{j=1..n-1} j = n(n-1)/2$
$\Sigma_{j=1..n-1} (j+1) = (n+2)(n-1)/2$

Analyzing Insertion Sort

$T(n) = c_1*n+c_2*(n-1)+c_3*(n-1)+c_4*(\Sigma_{j=1..n-1} (t_j+1))$
$+c_5*(\Sigma_{j=1..n-1} t_j)+c_6*(\Sigma_{j=1..n-1} t_j)+c_7*(n-1)$

Worst case

Reverse sorted → inner loop body executed for all previous elements. So, $t_j=j$.

→ T(n) is quadratic: $T(n)=An^2+Bn+C$

Average case

Half elements in A[0..j-1] are less than A[j]. So, $t_j = j/2$

→ T(n) is also quadratic: $T(n)=A'n^2+B'n+C'$

Best case

Already sorted → inner loop body never executed. So, $t_j=0$.

→ T(n) is linear: $T(n)=An+B$

Kinds of Analysis

(Usually) Worst case Analysis:

T(n) = max time on any input of size ***n***

Knowing it gives us a guarantee about the upper bound.

In some cases, worst case occurs fairly often

(Sometimes) Average case Analysis:

T(n) = average time over all inputs of size ***n***

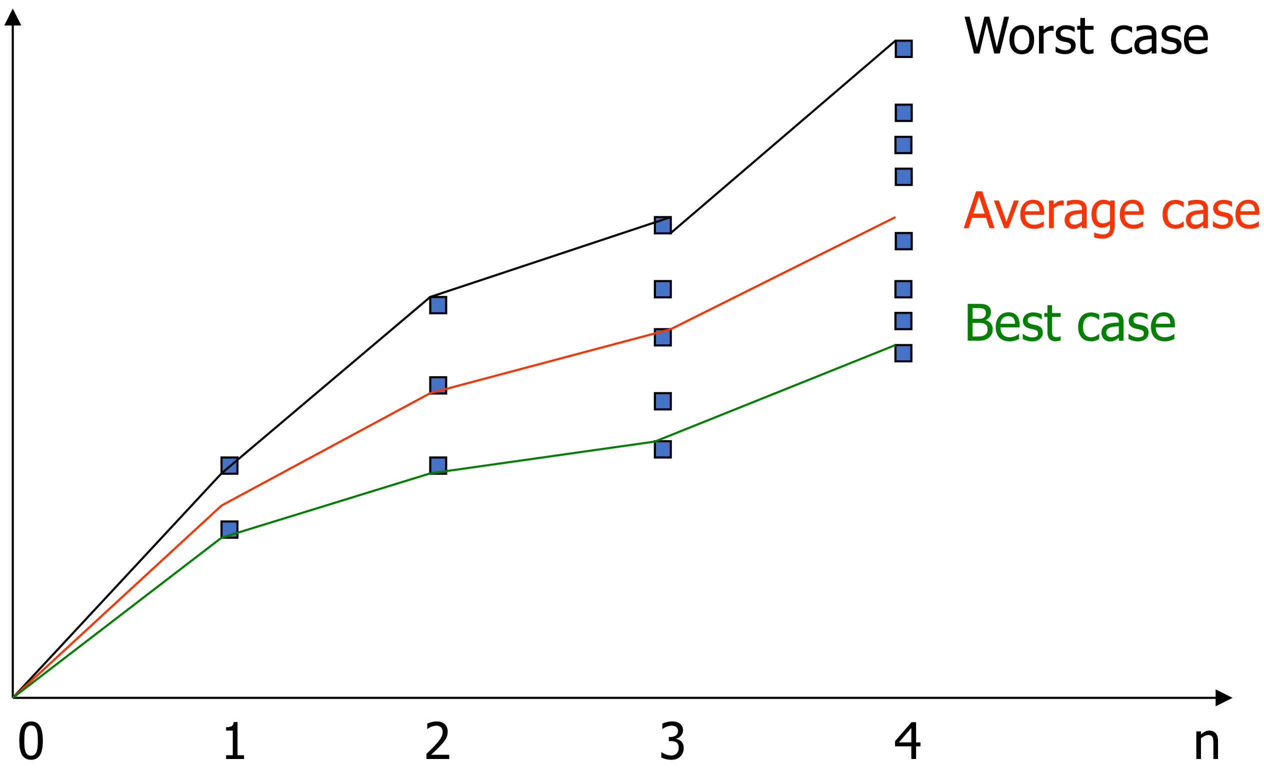
Average case is often as bad as worst case.

(Rarely) Best case Analysis:

Cheat with slow algorithm that works fast on some input.

Good only for showing bad lower bound.

Kinds of Analysis



- Worst Case: maximum value
- Average Case: average value
- Best Case: minimum value

Order of Growth

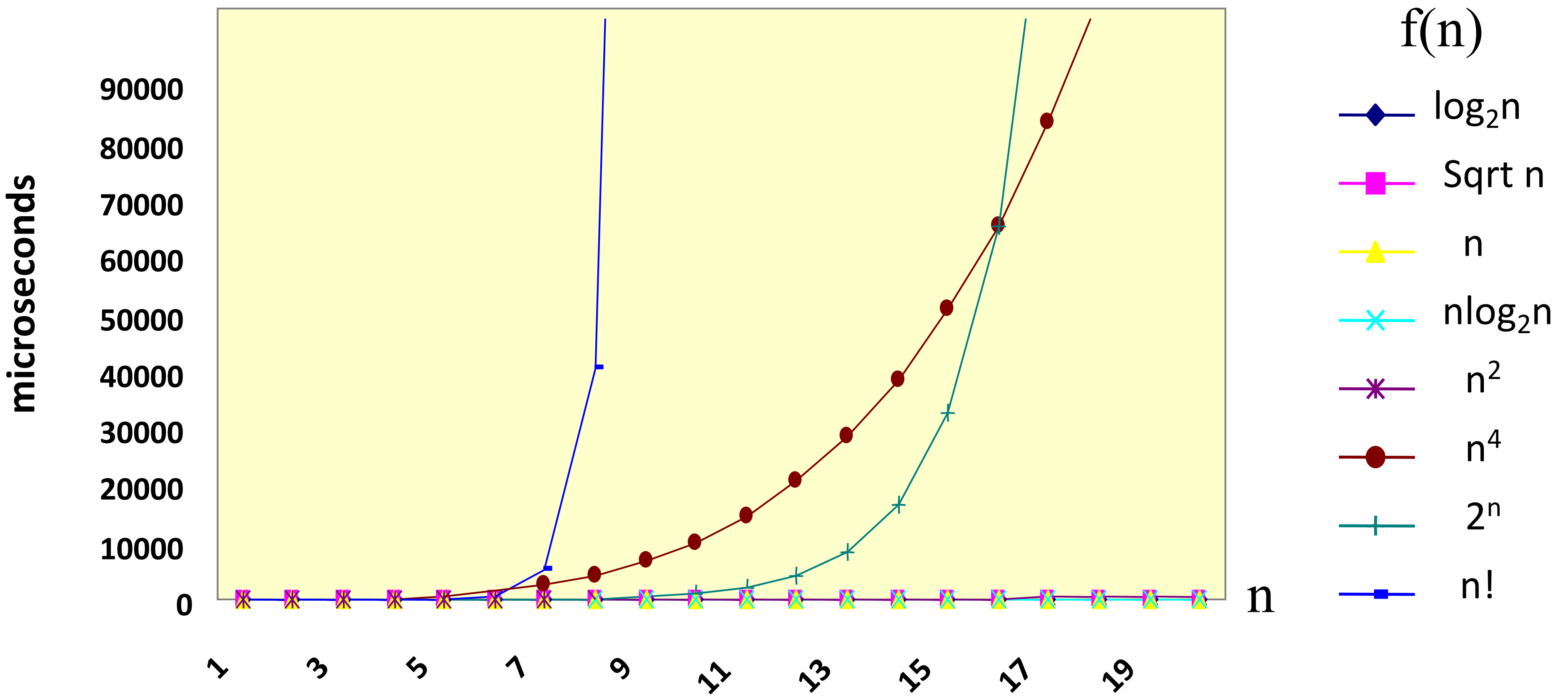
Examples:

Running time of algorithm in microseconds
(in term of data size *n*)

	f(n)	n=20	n=40	n=60
Algorithm A	$Log_2 n$	$4.32 * 10^{-6}$sec	$5.32 * 10^{-6}$sec	$5.91 * 10^{-6}$sec
Algorithm B	Sqrt(n)	$4.47 * 10^{-6}$sec	$6.32 * 10^{-6}$sec	$7.75 * 10^{-6}$sec
Algorithm C	n	$20 * 10^{-6}$sec	$40 * 10^{-6}$sec	$60 * 10^{-6}$sec
Algorithm D	$n \log_2 n$	$86 * 10^{-6}$sec	$213 * 10^{-6}$sec	$354 * 10^{-6}$sec
Algorithm E	n^2	$400 * 10^{-6}$sec	$1600 * 10^{-6}$sec	$3600 * 10^{-6}$sec
Algorithm F	n^4	0.16 sec	2.56 sec	________ sec
Algorithm G	2^n	1.05 sec	12.73 days	_______ years
Algorithm H	n!	77147 years	$2.56 * 10^{34}$ years	$2.64 * 10^{68}$ years

Order of Growth

Assume: an algorithm can solve a problem of size *n* in f(*n*) microseconds (10^{-6} seconds).



Note: for example, for all f(n) in $\Theta(n^4)$,

the shapes of their curves are nearly the same as $f(n)=n^4$.

Asymptotic Notation

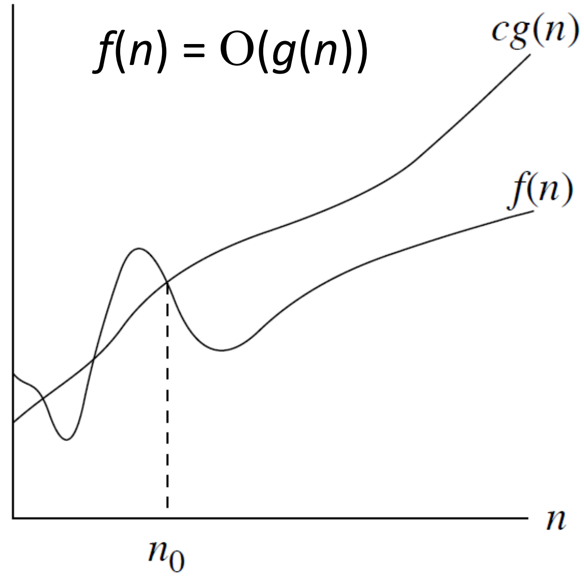
- How can we indicate running times of algorithms?
- Need a notation to express the growth rate of a function
- A way to compare "size" of functions:
  - O-notation ("Big-oh") $\approx \leq$ (upper bound)
  - Ω-notation ("Big-omega") $\approx \geq$ (lower bound)
  - Θ-notation ("theta") $\approx =$ (sandwich)

O -notation (1/2)

- O-notation provides an **asymptotic upper bound** of a function.
- For a given function $g(n)$, we denote $O(g(n))$

(pronounced "big-oh" of g of n) by the set of functions:

$O(g(n))$ = { $f(n)$: there exist **positive** constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$ }



O -notation (2/2)

- We write $f(n) = \mathrm{O}(g(n))$ to
  - Indicate that $f(n)$ is a member of the set $\mathrm{O}(g(n))$
  - Give that $g(n)$ is an upper bound for $f(n)$ to within a constant factor

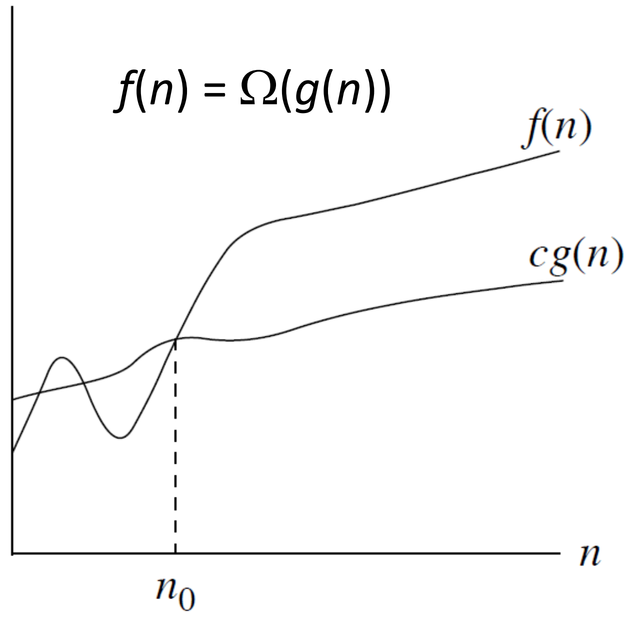
- Example: $2n^2 = \mathrm{O}(n^3)$, with $c = 1$ and $n_0 = 2$
  - When $n = 1$: $2(1)^2 = 2 \le (1)^3 = 1$ ☒
  - When $n = 2$: $2(2)^2 = 8 \le (2)^3 = 8$ ☑
  - When $n = 3$: $2(3)^2 = 18 \le (3)^3 = 27$ ☑

Ω -notation (1/2)

- Ω-notation provides an **asymptotic lower bound** of a function.
- For a given function $g(n)$, we denote $\Omega(g(n))$

(pronounced "big-omega" of g of n) by the set of functions:

$\Omega(g(n))$ = { $f(n)$: there exist positive constants c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0$ }



Ω -notation (2/2)

- We write $f(n) = \Omega(g(n))$ to
  - Indicate that $f(n)$ is a member of the set $\Omega(g(n))$
  - Give that $g(n)$ is a lower bound for $f(n)$ to within a constant factor

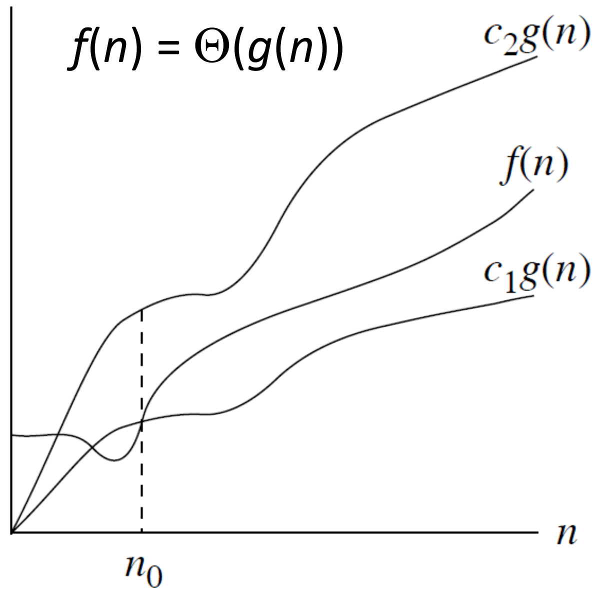
- Example: $n^2 + n = \Omega(n^2)$, with $c = 1$ and n_0=1
  - When $n = 1$: $(1)^2 + 1 = 2 \geq (1)^2 = 1$ ☑
  - When $n = 2$: $(2)^2 + 2 = 6 \geq (2)^2 = 4$ ☑
  - When $n = 3$: $(3)^2 + 3 = 12 \geq (3)^2 = 9$ ☑

Θ -notation (1/2)

- Θ-notation provides an **asymptotically tight bound** of a function.
- For a given function $g(n)$, we denote $\Theta(g(n))$

(pronounced "theta" of g of n) by the set of functions:

$\Theta(g(n))$ = {$f(n)$: there exist positive constants c_1, c_2 and n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0$}



Θ -notation (2/2)

- Theorem

  $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

- Example: $n^2/2 - 2n = \Theta(n^2)$, with $c_1 = 1/4$, $c_2=1/2$, and $n_0 = 8$
  - When n = 7: $1/4[(7)^2] = 12.25 \leq (7)^2/2 - 2(7) = 10.5 \leq 1/2[(7)^2] = 24.5$ ☒
  - When n = 8: $1/4[(8)^2] = 16 \leq (8)^2/2 - 2(8) = 16 \leq$
    $1/2[(8)^2] = 32$ ☑
  - When n = 9: $1/4[(9)^2] = 20.25 \leq (9)^2/2 - 2(9) = 22.5 \leq 1/2[(9)^2] = 40.5$ ☑

O versus o

- Little-o Notation
  - ➢ $f(n) = o(g(n))$: a strict upper bound for a function $f(n)$
  - ➢ $o(g(n)) = \{ f(n)$: there exist positive constants c and n0 such that $0 \leq f(n) < cg(n)$ for all $n \geq n0$ $\}$
  - ➢ o versus O : o means better e.g. $n=o(n^2)$
- Why $100n=O(n)$?
  - ➢ When n is very big like 10000000000000000
  - ➢ 100n: 1000000000000000000
  - ➢ n: 10000000000000000
  - ➢ nearly the same
- Why $n=o(n^2)$?
  - ➢ When n is 10000000000000000
  - ➢ n^2 is 100000000000000000000000000000000
  - ➢ Differ a lot!

Asymptotic Notation

- Relationship between typical functions
  - $\log n = o(n)$
  - $n = o(n \log n)$
  - $n^c = o(2^n)$ where n^c may be n^2, n^4, etc.
  - If $f(n)=n+\log n$, we call *log n* lower order terms

$\log n < \mathrm{sqrt}(n) < n < n\log n < n^2 < n^4 < 2^n < n!$

Asymptotic Notation

- When calculating asymptotic running time
  - Drop low-order terms
  - Ignore leading constants
- Example 1: $T(n) = An^2+Bn+C$
  - An^2
  - $T(n) = O(n^2)$
- Example 2: $T(n) = An\log n+Bn^2+Cn+D$
  - Bn^2
  - $T(n) = O(n^2)$

Exercise 1

Order the following functions by growth rate:

N, N^2, $N \log N$, $N \log\log N$, $N \log (N^2)$, $2/N$, $2^{N/2}$, 37, $N^2 \log N$.

Indicate which functions grow at the same rate (it they are).

$$\boxed{2/N < 37 < N < N\log\log N < N\log N \leq N\log(N^2) < N^2 < N^2\log N < 2^{N/2}}$$

Asymptotic Performance

Very often the algorithm complexity can be observed directly from simple algorithms

Insertion-Sort(A)

```
for j = 1 to n-1
    key = A[j]
    i = j-1
    while i >= 0 and A[i] > key
        A[i+1] = A[i]
        i = i - 1
    A[i+1] = key
```

$O(n^2)$

There are 4 very useful rules for such Big-Oh analysis ...

Asymptotic Performance

General rules for Big-Oh Analysis:

Rule 1. FOR LOOPS

The running time of a *for* loop is at most the running time of the statements inside the *for* loop (including tests) times no. of iterations

```
for (i=0;i<N;i++)
      a++;
```

O(N)

Rule 2. NESTED FOR LOOPS

The total running time of a statement inside a group of nested loops is the running time of the statement multiplied by the product of the sizes of all the loops.

```
for (i=0;i<N;i++)
      for (j=0;j<N;j++)
            k++;
```

$O(N^2)$

Rule 3. CONSECUTIVE STATEMENTS

Count the maximum one.

```
for (i=0;i<N;i++)
      a++;

for (i=0;i<N;i++)
      for (j=0;j<N;j++)
            k++;
```

$O(N^2)$

Rule 4. IF / ELSE

For the fragment:

```
If (condition)
      S1
else
      S2,
```

take the test + the maximum for S1 and S2.

Asymptotic Performance

Example of Big-Oh Analysis:

```
void function1(int n)
{   int i, j;
    int x=0;

    for (i=0;i<n;i++)
            x++;

    for (i=0;i<n;i++)
        for (j=0;j<n;j++)
            x++;
}
```

This function is O(__)

```
void function2(int n)
{   int i;
    int x=0;

    for (i=0;i<n/2;i++)
            x++;
}
```

This function is O(__)

Asymptotic Performance

Example of Big-Oh Analysis:

```
void function3(int n)
{  int i;
   int x=0;

   if (n>10)
      for (i=0;i<n/2;i++)
         x++;
   else
   {  for (i=0;i<n;i++)
         for (j=0;j<n/2;j++)
            x--;
   }
}
```

This function is O(__)

```
void function4(int n)
{  int i;
   int x=0;

   for (i=0;i<10;i++)
      for (j=0;j<n/2;j++)
         x--;
}
```

This function is O(__)

Asymptotic Performance

Example of Big-Oh Analysis:

```
void function5(int n)
{  int i;
   for (i=0;i<n;i++)
      if (IsSignificantData(i))
         SpecialTreatment(i);
}
```

This function is O(____)

Suppose

- IsSignificantData is O(n)
- SpecialTreatment is O(n log n)

Comparison of Good and Bad

- Coin Flipping
  - There are N rows of coins
  - Each row consists of 9 coins
  - They formulate a matrix of size N*9
  - Some coins are heads up and some are tails up
  - We can flip a whole row or a whole column every time
  - Your program need to find a flipping method that can make the number of “heads up” coins maximum

Asymptotic Performance

- Recursion

```
int Power(int base,int pow)
{       if (pow==0)   return 1;
        else   return base*Power(base,pow-1);
}
```

- Example

  $3^2=9$

  Power(3,2)=3*Power(3,1)

  Power(3,1)=3*Power(3,0)

  Power(3,0)=1

  T(n): the number of multiplications needed to compute Power(3,n)

  $T(n)=T(n-1)+1$; $T(0)=0$

  $T(n)=n$

  Function T(n) is O(n)

Asymptotic Performance

- Why recursion?
  - ➢ Can’t we just use iteration (loop)?
- The reason for recursion
  - ➢ Easy to program in some situations
- Disadvantage
  - ➢ More time and space required
- Example:
  - ➢ Tower of Hanoi Problem

Tower of Hanoi

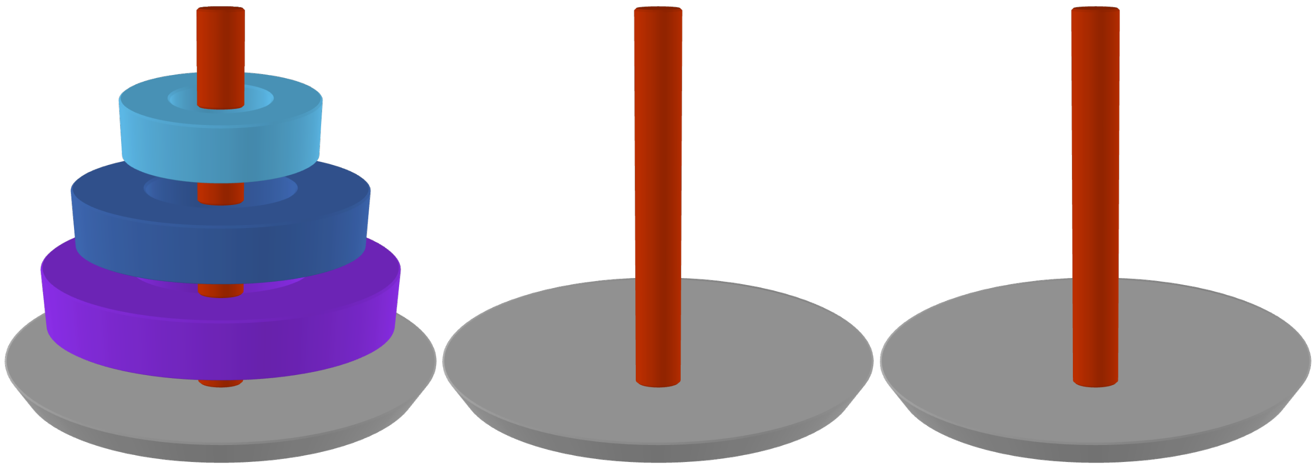
The problem:
Use fewest steps to move all disks from the source rod to the target without violating the rules through the whole process
(given one intermediate rod for buffering)?

Given some rods for stacking disks.

Rules:

(1) The disks must be stacked in order of size.

(2) Each time move 1 disk.



a source rod   an intermediate rod   a target rod

Tower of Hanoi

- Suppose you can manage the n-1 disks
- How do you solve the n disks case?
- A recursive solution:
  - ➢ Step 1: Move the top n-1 disks from source rod to intermediate rod via target rod
  - ➢ Step 2: Move the largest disk from source rod to target rod
  - ➢ Step 3: Move the n-1 disks from intermediate rod to target rod via source rod

Tower of Hanoi

```
void Towers (int n, int Source, int Target, int Interm)
{
  if (n==1)
        cout<<"From"<<Source<<"To"<<Target<<endl;
  else
  {     Towers(n-1, Source, Interm, Target);
        Towers(1, Source, Target, Interm);
        Towers(n-1, Interm, Target, Source);
  }
}
```

How many "cout" are executed?

- $T(n)=2T(n-1)+1$

Recursive Relation

- $T(n)=T(n-1)+A$; $T(1)=1$
  - → $T(n)=O(n)$
- $T(n)=T(n-1)+n$; $T(1)=1$
  - → $T(n)=O(n^2)$
- $T(n)=2T(n/2) + n$; $T(1)=1$
  - → $T(n)=O(n \log n)$, why???

- More general form: $T(n)=aT(n/b)+cn$
  - Master's Theorem (You are not required to know)

Learning Objectives

1. Understand the meaning of O and o and able to use
2. Analyze program complexities for simple programs
3. Able to compare which function grows faster
4. Able to do worst case analysis

D:1; C:1,2; B:1,2,3; A:1,2,3,4

Exercise 1

Give an analysis of the running time (Big-Oh will do).

(1)
```
sum = 0;
for( i = 0; i < n; ++i )
    ++sum;
```

(2)
```
sum = 0;
for( i = 0; i < n; ++i )
    for( j = 0; j < n; ++j )
        ++sum;
```

(3)
```
sum = 0;
for( i = 0; i < n; ++i )
    for( j = 0; j < n * n; ++j )
        ++sum;
```

(4)
```
sum = 0;
for( i = 0; i < n; ++i )
    for( j = 0; j < i; ++j )
        ++sum;
```

(5)
```
sum = 0;
for( i = 0; i < n; ++i )
    for( j = 0; j < i * i; ++j )
        for( k = 0; k < j; ++k )
            ++sum;
```

(6)
```
sum = 0;
for( i = 1; i < n; ++i )
    for( j = 1; j < i * i; ++j )
        if( j % i == 0 )
            for( k = 0; k < j; ++k )
                ++sum;
```

Exercise 2

Give an analysis of the running time (Big-Oh will do).

```
int Search(int arr[], int l, int r, int x) {
    if (r >= l) {
        int mid = l + (r - l) / 2;
        if (arr[mid] == x)
            return mid;

        if (arr[mid] > x)
            return Search(arr, l, mid - 1, x);

        else
                return Search(arr, mid + 1, r, x);
    }
    return -1;
}
```