COL106
Data Structures and Algorithms

Subodh Sharma and Rahul Garg

Stacks and Queues

Images courtesy: www.123rf.com; www.livemint.com

Recap of Complexity Analysis

Definition: A function $f(n)$ is $O(g(n))$ if there exists constants c, n_0 such that $f(n) \leq c\, g(n)$ for all $n \geq n_0$

Definition: A function $f(n)$ is $\Omega(g(n))$ if there exists constants c, n_0 such that $f(n) \geq c\, g(n)$ for all $n \geq n_0$.

Definition: A function $f(n)$ is $\Theta(g(n))$ if $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$

Definition: A function $f(n)$ is $o(g(n))$ if for every constant c, there is a n_0 such that $f(n) \leq c\, g(n)$ for all $n \geq n_0$

Asymptotic Notations: Analogy

Functions Asymptotic Analysis	Numbers
f(n) is O(g(n))	$f \leq g$
f(n) is Ω(g(n))	$f \geq g$
f(n) is Θ(g(n))	$f = g$
f(n) is o(g(n))	$f < g$
f(n) is ω(g(n))	$f > g$

Time Complexity Notations: Little-o

Represents strict < asymptotic inequality

Definition [little-o]: A function f(n) is o(g(n)) if for every constant c, there is a n_0 such that $f(n) \leq c\, g(n)$ for all $n \geq n_0$

Definition [Big-O]: A function f(n) is O(g(n)) if there exists constants c, n_0 such that $f(n) \leq c\, g(n)$ for all $n \geq n_0$

Little o vs. Big O

$f1(n) = n^2$

$f2(n) = n^2 + 10 \log(n)$

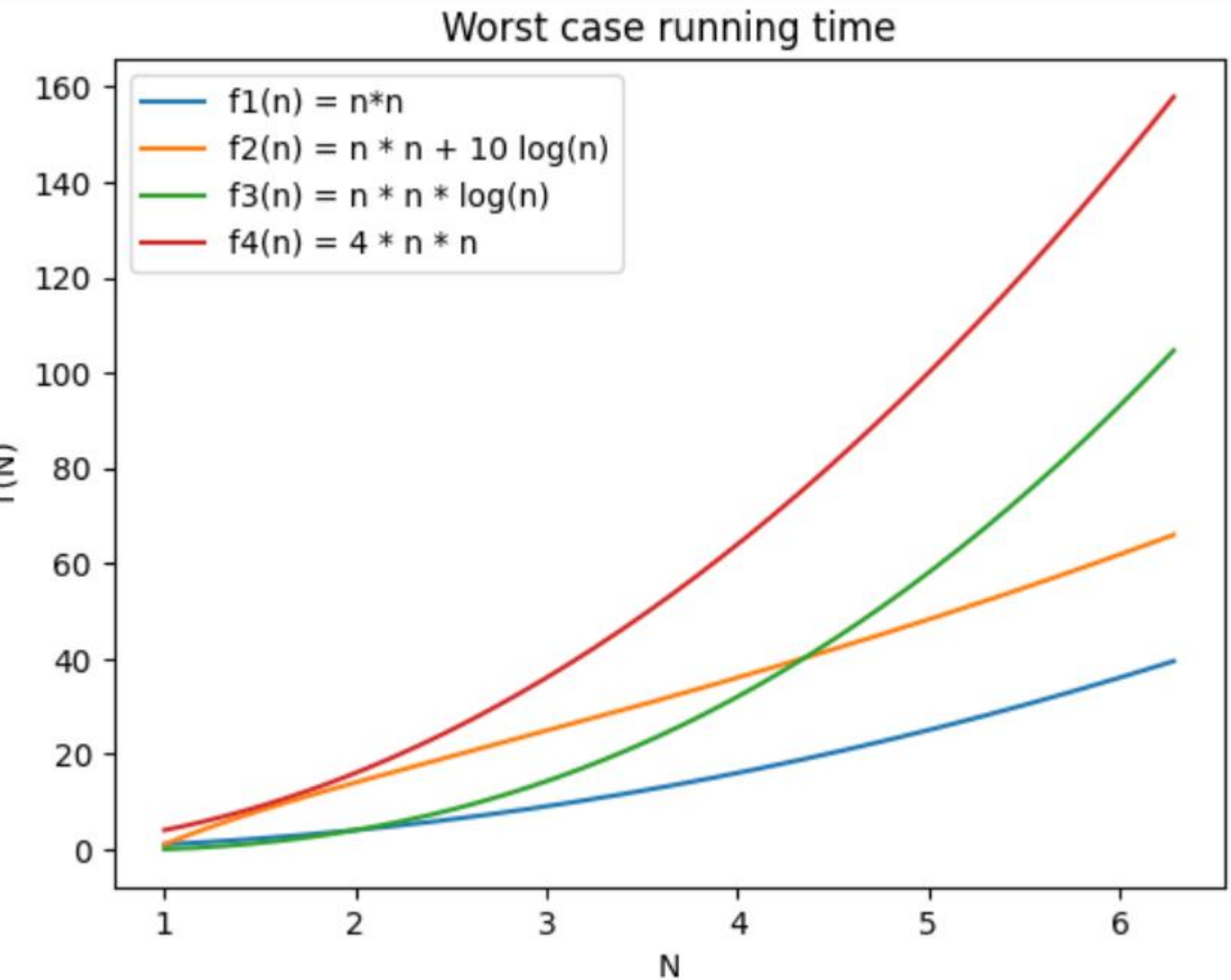
$f3(n) = n^2 \log(n)$

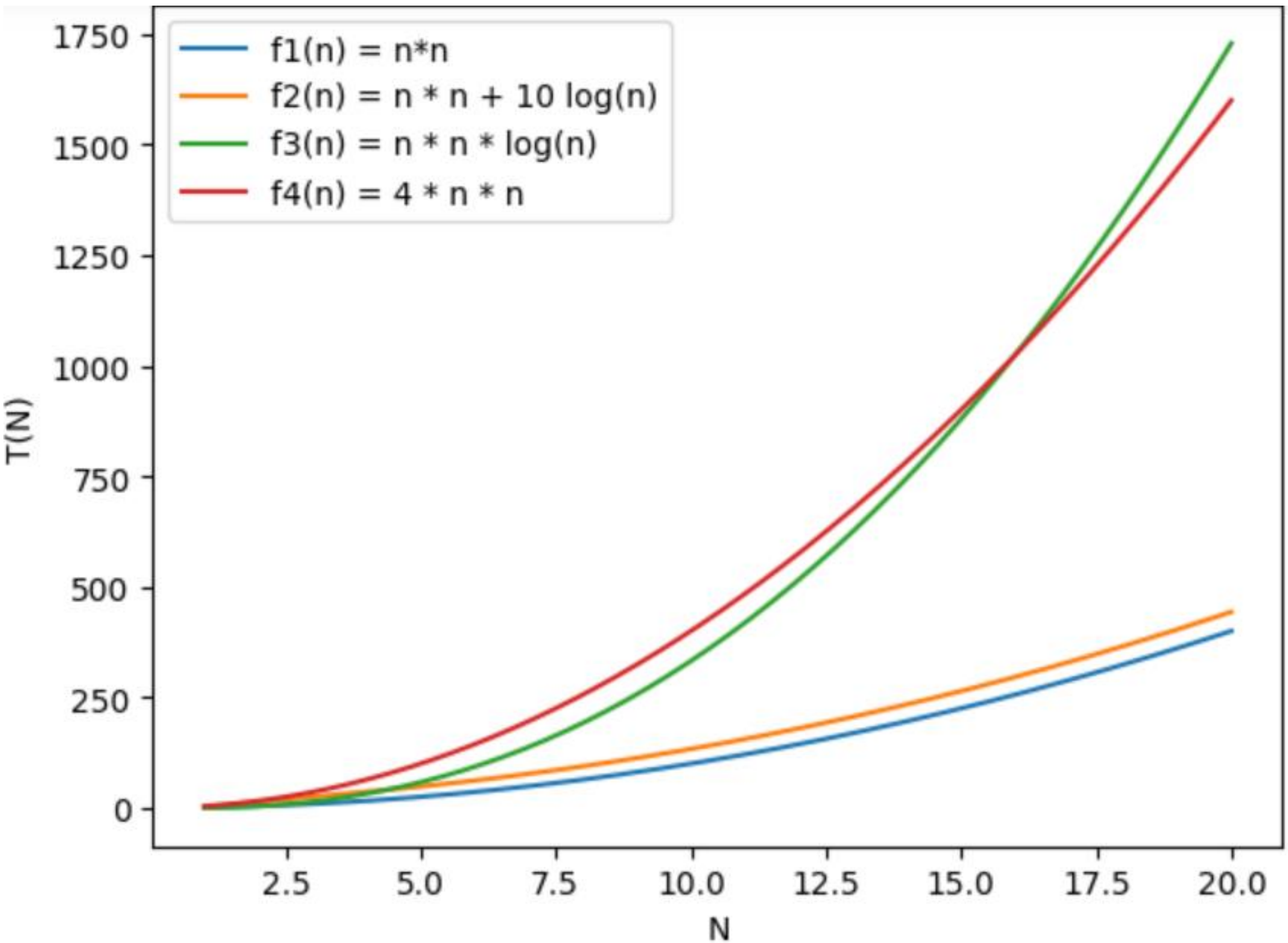
$f4(n) = 4n^2$

Statement	T/F
$f2(n) = O(f1(n))$	T
$f1(n) = O(f2(n))$	T
$f2(n) = o(f1(n))$	F
$f4(n) = O(f3(n))$	T
$f4(n) = o(f3(n)$	T
$f4(n) = o(f1(n)$	F
$f4(n) = \Theta(f2(n))$	T
$f4(n) = \Theta(f3(n))$	F

Definition [little-o]: A function $f(n)$ is $o(g(n))$ if for every constant c, there is a n_0 such that $f(n) \leq c\, g(n)$ for all $n \geq n_0$

Definition [Big-O]: A function $f(n)$ is $O(g(n))$ if there exists constants c, n_0 such that $f(n) \leq c\, g(n)$ for all $n \geq n_0$


Worst case running time
f1(n) = n*n
f2(n) = n * n + 10 log(n)
f3(n) = n * n * log(n)
f4(n) = 4 * n * n
T(N)
N
0
20
40
60
80
100
120
140
160
1
2
3
4
5
6


f1(n) = n*n
f2(n) = n * n + 10 log(n)
f3(n) = n * n * log(n)
f4(n) = 4 * n * n
T(N)
N
0
250
500
750
1000
1250
1500
1750
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

Stacks

- What is a stack?
- Container of objects
  - Put an object (push)
  - Take an object out (pop)
  - Order of objects: Last-In First-Out (LIFO)
- Real life examples of a stack

Image source: rawpixel.com

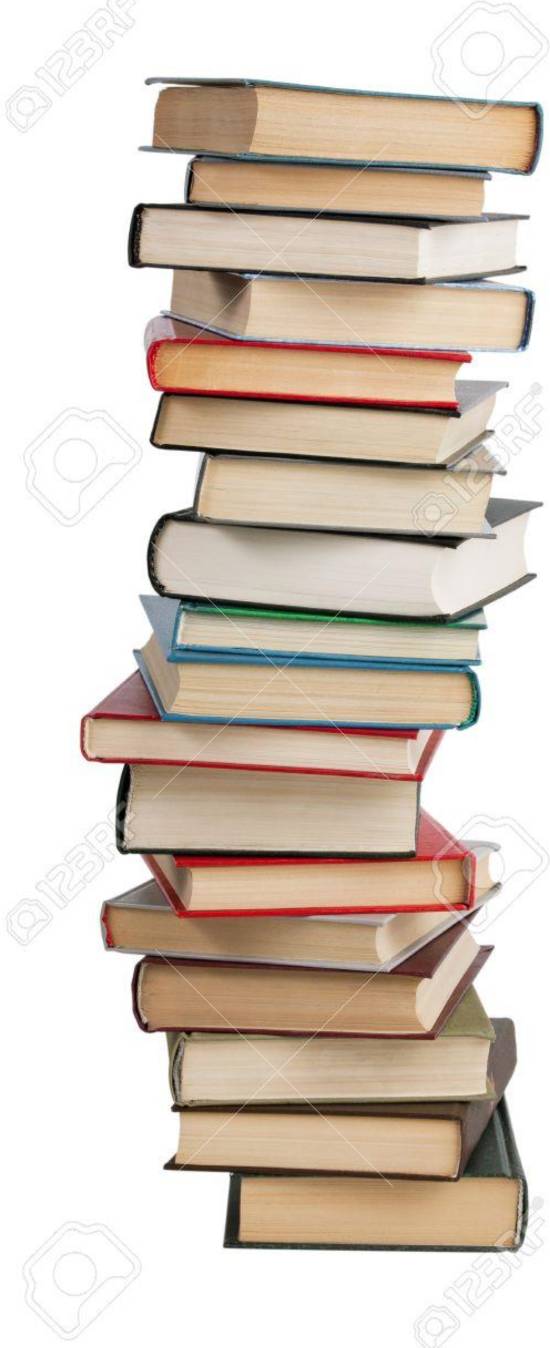


Image source: 123rf.com

Other Examples

Other Examples of Stack

- Email stack
- Web browsing history
- Text editors undo

Operations on Stack

- New
- Push
- Pop
- Top
- IsEmpty
- Size
- Print all from top
- Print all from bottom
- Print one form top
- Print one from bottom

Demo by Volunteers

Available commands:

Push, Pop, Top, IsEmpty

Stack of integers

Operation	Output	Stack Content After the Operation

Queues

- What is a Queue?
- Container of objects
  - Put an object (enqueue)
  - Take an object out (dequeue)
  - Order of objects: First-In First-Out (FIFO)
- Real life examples of a queue

Image credit: Jurgen Ziewe/Getty Images; Taken from: [Cue vs. Queue: How to Choose the Right Word (thoughtco.com)](thoughtco.com)

Female voters standing in a queue for casting their votes during the 3rd Phase of General Elections-2014, in New Delhi on April 10, 2014.
Image source: Wikipedia

Queues

- Other real-life examples

Operations of Queues

- Enqueue
- Dequeue
- Front
- Size
- IsEmpty

Demo by Volunteers

Available commands:
Enqueue, Dequeue, Front, Size, IsEmpty
Queue of integers

Operation	Output	Queue Content After the Operation

Stack Application: Matching Brackets

- Parentheses: ( ); Braces: { }; Square brackets: [ ]
- How do we find out if a string of the above symbols is well formed?
- Definition: A sequence of symbols consisting of parentheses, braces and square brackets is a well-formed-sequence if and only if
  - It is either blank string, or (), or {} or [] OR
  - (well-formed-sequence), or {well-formed-sequence} or [well-formed-sequence] OR
  - A well-formed-sequence followed by another well-formed-sequence

Examples

1: ( ) [([ ] [ ])] [ ]

2: ( )] [{ }

3: ([ ] ({ [ ] ( ) } [( ) ( )]))

4: ([ ] ({ [ ] ( ) } [( ) (}]))

5: ( ) { } [ ] { ( ) [ ] { } } [( ) { } [ ]]

6: ( ) { { }] { } ( ) [ ] { ([ ] { }) }

How to check for Well-Formed-Sequence?

- Definition: A sequence of symbols consisting of parentheses, braces and square brackets is a well-formed-sequence if and only if
  - It is either blank string, or (), or {} or [] OR
  - (well-formed-sequence), or {well-formed-sequence} or [well-formed-sequence] OR
  - A well-formed-sequence followed by another well-formed-sequence
- Algorithm 1: Keep deleting (), {} and [] until you can no longer delete. If you are left with a blank-string then the original sequence was well-formed

Examples

1: ( ) [([ ] [ ])] [ ]

2: ( )] [{ }

3: ([ ] ({ [ ] ( ) } [( ) ( )]))

4: ([ ] ({ [ ] ( ) } [( ) (}]))

5: ( ) { } [ ] { ( ) [ ] { } } [( ) { } [ ]]

6: ( ) { { }] { } ( ) [ ] { ([ ] { }) }

Algorithm 1

```
repeat {
    done = 1
        for (i = 0 to length(s) - 1) {
            if ((S[i] == '(' && S[i+1] == ')') ||
                (S[i] == '{' && S[i+1] == '}') ||
                (S[i] == '[' && S[i+1] == ']')) {

                for (j = i + 2 to length(s) - 1)
                    S[j - 2] = S[j];
                done = 0; i = 0;
                Update length(S);
            }
    } until (done == 1);
if (length(S) == 0) return T else return F;
```

Analysis of Algorithm 1

Delete string of size 2

(((((((((((((((((((((((((((((((()))))))))))))))))))))))))))))))))

N/2

Scan string of size N/2- 1

N/2

Copy string of size N/2- 1

T(N) = N/2 – 1 + N/2 – 1 + C + T(N-2)

T(0) = C

T(N) = O(N^2)

Can we have a faster Algorithm Using Stacks?

How to check for Well-Formed-Sequence?

- Definition: A sequence of symbols consisting of parentheses, braces and square brackets is a well-formed-sequence if and only if
  - It is either blank string, or (), or {} or [] OR
  - (well-formed-sequence), or {well-formed-sequence} or [well-formed-sequence] OR
  - A well-formed-sequence followed by another well-formed-sequence
- Algorithm 2:
  1. Scan the input and push if you encounter (or { or [
  2. Eat if you encounter a well-formed sequence
  3. If you encounter) or } or] check for a matching (or { or [from top of the stack. Pop if matching symbol found. Else report that the sequence is not well-formed
  4. If entire input string scanned and stack is empty then input was well-formed-sequence

Algorithm 2

Input: String S; Output T if S is well-formed F otherwise

initialize stack;

```
for (i = 0 to length(S) - 1) {
    if (S[i] is '(' or '{' or '[')
        stack.push(S[i]);
    else if (not stack.isEmpty() &&
              ((stack.top == '(' && S[i] == ')') ||
               (stack.top == '{' && S[i] == '}') ||
               (stack.top == '[' && S[i] == ']') ))
        stack.pop();
    else return F;
}
if (stack.isEmpty())
    return T;
else
    return F;
```

Algorithm 2 ([ ] ({ [ ] ( ) } [( ) ( )]))

0123456789ABCDEFGH

Value of i	Output	Stack Content at the end of the loop

Analysis of Algorithm 2

(((((((((((((((((((((((((((((((()))))))))))))))))))))))))))))))))))

N/2
Push string of size N/2

N/2
Pop string of size N/2

T(N) = N/2 + N/2 + C
T(N) = O(N)

Analysis if Algorithm 2

- Is this analysis correct?

Analysis of Algorithm 2

- T(N) is the worst-case time taken by the algorithm for any input of size N
- Not for just one specific input
- We need to find an upper-bound on T(N)

Algorithm 2: Analysis

Count number of pop, push, top operations and iterations of the loop (for, if1, if2, return)

```
for (i = 0 to length(S) - 1) {
    if (S[i] is '(' or '{' or '[')
        stack.push(S[i]);
    else if (not stack.isEmpty() &&
             ((stack.top == '(' && S[i] == ')') ||
              (stack.top == '{' && S[i] == '}') ||
              (stack.top == '[' && S[i] == ']') ))
        stack.pop();
    else return F;
}
if (stack.isEmpty())
    return T;
else
    return F;
```

Stack Implementation using Array

Array Based Implementation: Stack

```
class Stack {
private:
  int stackCapacity;
  int *S;
  int t;
  // Invariant: S[t] is the top of the stack
  // Stack contents are from S[0] to S[t]
```

Array Based Implementation: Stack

```
class Stack {
  private:
      int stackCapacity;
      int *S;
      int t;
  public:
      // Invariant: S[t] is the top of the stack
      // Stack contents are from S[0] to S[t]

      Stack(int sz) { S = new int[sz]; t = -1; stackCapacity = sz;}
      ~Stack() { delete S; }
      void push(int e) { S[++t] = e; }
      int pop() {return S[t--]; }
      nt top() {return S[t]; }
      bool isEmpty() { return (t == -1); }
      int size() { return t + 1; }
};
```

Let us Debug

```
class Stack {
  private:
      int stackCapacity;
      int *S;
      int t;
  public:
      // Invariant: S[t] is the top of the stack
      // Stack contents are from S[0] to S[t]

      Stack(int sz) { S = new int[sz]; t = -1; stackCapacity = sz;}
      ~Stack() { delete S; }
      void push(int e) { S[++t] = e; }
      int pop() {return S[t--]; }
      nt top() {return S[t]; }
      bool isEmpty() { return (t == -1); }
      int size() { return t + 1; }
};
```

Error Free push() and pop()

```
#define STACK_FULL_EXCEPTION -1
#define STACK_EMPTY_EXCEPTION -2
void Stack::push(int e) {
      if (t >= stackCapacity - 1) throw STACK_FULL_EXCEPTION;
      else S[++t] = e;
}
int Stack::pop() {
        if (t == -1) throw STACK_EMPTY_EXCEPTION;
        else return S[t--];
}
int Stack::top() {
        if (t == -1) throw STACK_EMPTY_EXCEPTION;
        else return S[t];
```

A Growable Stack: Additive Increase

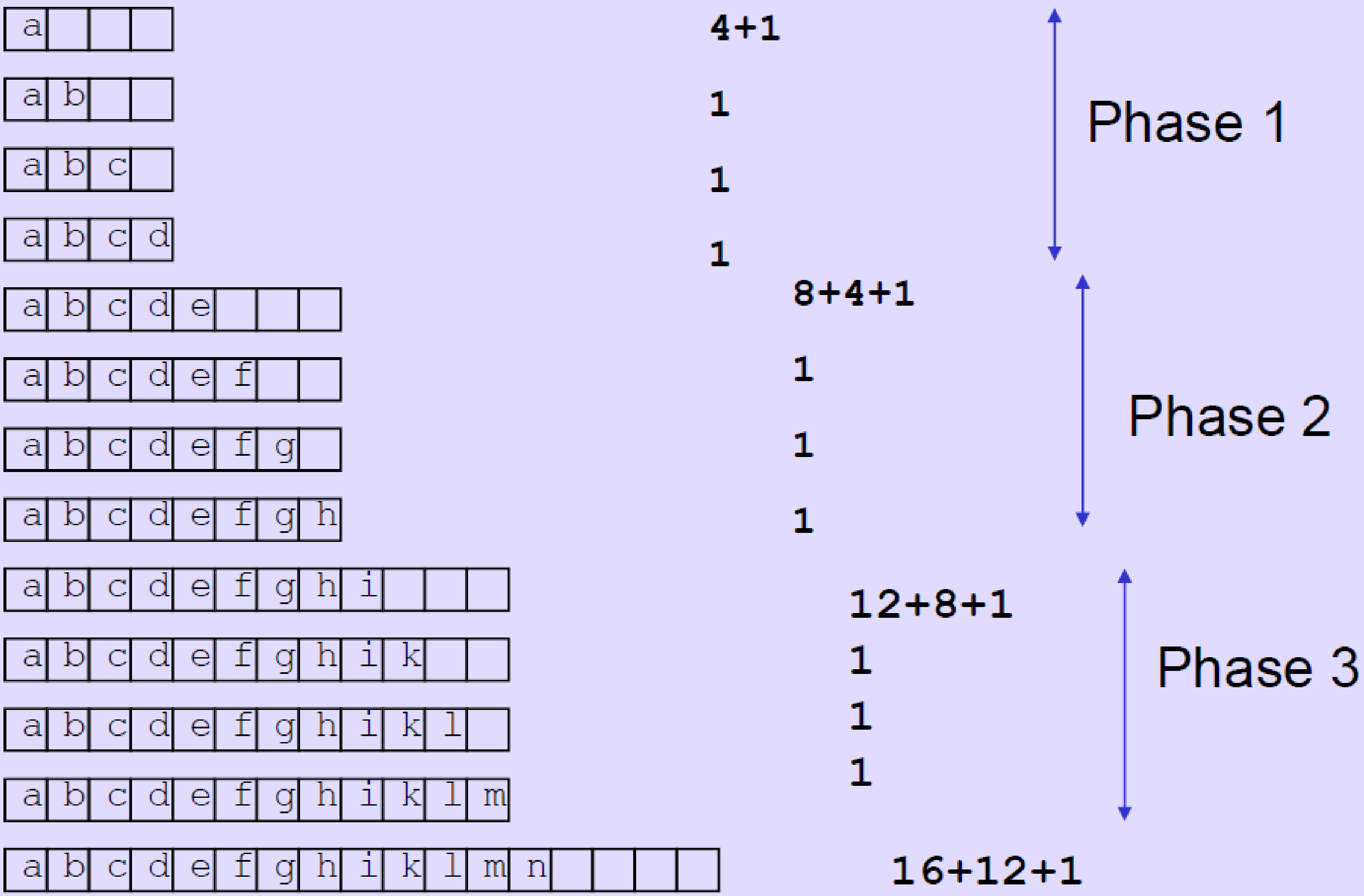
```
// Growable stack: Additive Increase
#define C 4
void Stack::push(int e) {
        if (t >= stackCapacity - 1) { // Reallocate space and copy
      int *temp = new int[stackCapacity + C] // TBD: Check for error
      for (i = 0; i <= t; i++)
              temp[i] = S[i];
      delete S;
      S = temp;
      stackCapacity += C;
        }
        S[++t] = e;
}
```

A Growable Stack: Multiplicative Increase

```
#define C 2
void Stack::push(int e) {
  if (t >= stackCapacity - 1) { // Reallocate space and copy
    int *temp = new int[stackCapacity * C]; // TBD: Check for error
    for (i = 0; i <= t; i++)
      temp[i] = S[i];
    delete S;
    S = temp;
    stackCapacity *= C;
  }
  S[++t] = e;
}
```

Analysis: Additive Increase (C=4)

start with an array of size 0. cost of a special push is 2N + 5



Slide adapted from Naveen Garg, IIT Delhi

Analysis: Additive Increase (C=4)

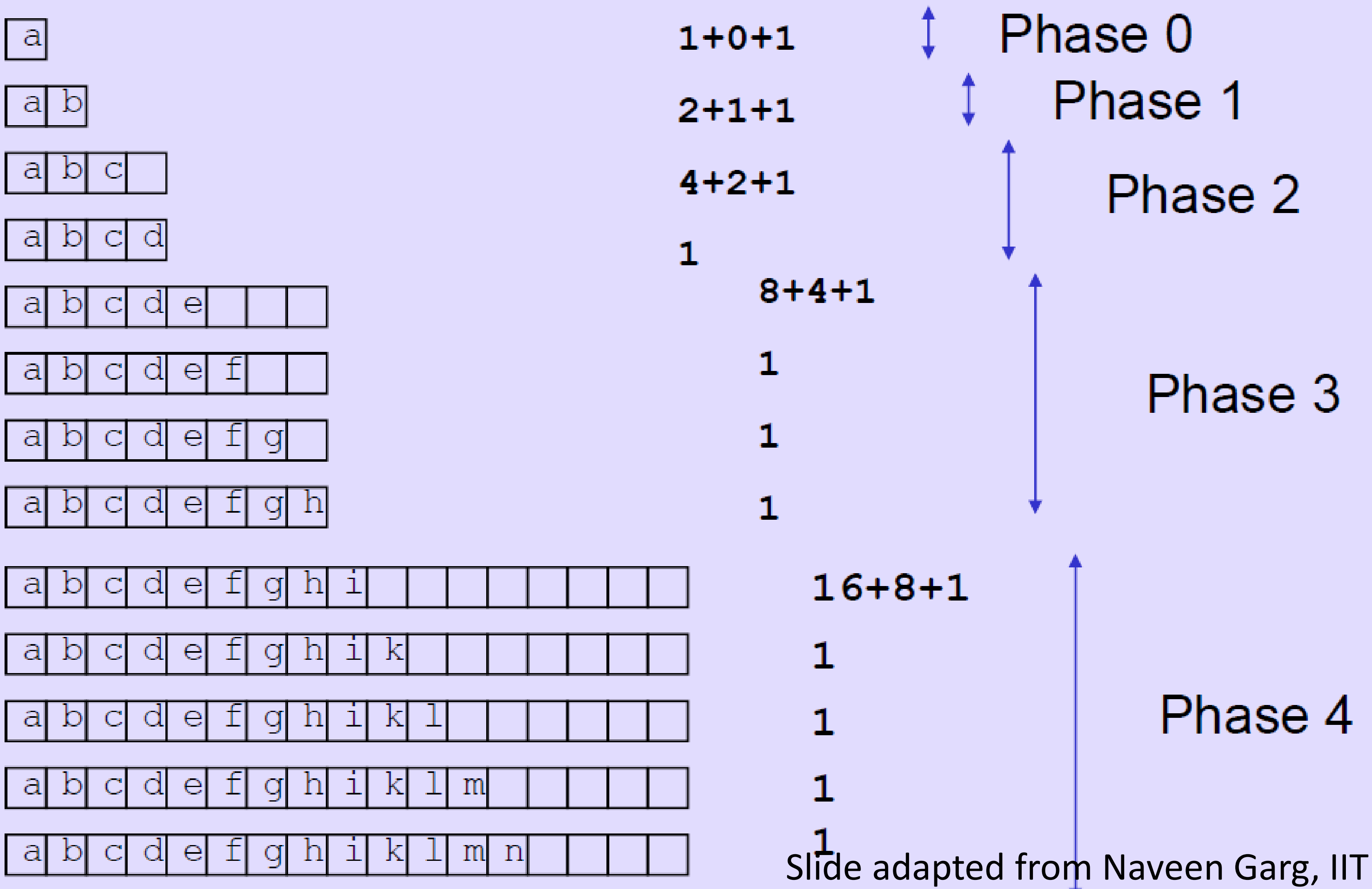
- In phase i the array has size $c \times i$
- Total cost of phase i is
  - $c \times i$ is the cost of creating the array
  - $c \times (i-1)$ is the cost of copying elements into new array
  - c is the cost of the c pushes.
- Hence, cost of phase i is 2ci
- In each phase we do c pushes. Hence for n pushes we need n/c phases. Total cost of these n/c phases is

$$2c\,(1 + 2 + 3 + \ldots + n/c) \approx O(n^2)$$

Slide adapted from Naveen Garg, IIT Delhi

Analysis: Multiplicative Increase

start with an array of size 0. cost of a special push is 3N + 1



Slide adapted from Naveen Garg, IIT Delhi

Analysis: Multiplicative Increase

- In phase i the array has size 2^i
- Total cost of phase i is
  - 2^i is the cost of creating the array
  - 2^{i-1} is the cost of copying elements into new array
  - 2^{i-1} is the cost of the 2^{i-1} pushes done in this phase
- Hence, cost of phase i is 2^{i+1}
- If we do n pushes, we will have log n phases.
- Total cost of n pushes
- $= 2 + 4 + 8 + ... + 2^{\log n + 1} = 4n - 1$

Slide adapted from Naveen Garg, IIT Delhi

Further Improvement

- Current algorithm only grows the stack
- But if the program no longer has many elements in the stack, then the memory can potentially be freed up for other programs
- Stack shrinking strategies
  - Additive decrease
  - Multiplicative decrease
- Problem with fully adaptive stacks
  - The problem of fluctuations
- Building hysteresis in the algorithm
  - Analysis of the two strategies with hysteresis

Fully Adaptive Stack: Problem and Solution Outline

Fully Adaptive Stack

```
#define C 2
int Stack::pop() {
  int result = S[t];
  if (t <= stackCapacity / C) { // Reallocate space and copy
    int *temp = new int[stackCapacity / C]; // TBD: Check error
    for (i = 0; i <= t; i++)
      temp[i] = S[i];
    delete S;
    S = temp;
    stackCapacity /= C;
  }
  t--;
  return result;
}
```

Introducing Hysteresis

Fully Adaptive Stack with Hysteresis

```
// Fully adaptive stack with hysteriesis
#define C 2
int Stack::pop() {
  int result = S[t];
  if (t <= stackCapacity / C / C) { // Reallocate space and copy
    int *temp = new int[stackCapacity / C] // TBD: Check for error
      for (i = 0; i <= t; i++)
        temp[i] = S[i];
    delete S;
    S = temp;
    stackCapacity /= C;
  }
  t--; return result;
}
```

Thank You