CS 106B, Lecture 5
Stacks and Big O

reading:

Programming Abstractions in C++, Chapter 4-5

Plan for Today

- Analyzing algorithms using **Big O** analysis
  - Understand what makes an algorithm "good" and how to compare algorithms
- Another type of collection: the **Stack**

Plan for Today

- Analyzing algorithms using **Big O** analysis
  - Understand what makes an algorithm "good" and how to compare algorithms
- Another type of collection: the **Stack**

Big O Intuition

- Lots of different ways to solve a problem
- Measure algorithmic **efficiency**
  - Resources used (time, memory, etc.)
  - We will focus on time
- Idea: algorithms are better if they take less time
- Problem: amount of time a program takes is variable
  - Depends on what computer you're using, what other programs are running, if your laptop is plugged in, etc...

Big O

- Idea: assume each statement of code takes some unit of time
  - for the purposes of this class, that unit doesn't matter
- We can count the number of units of time and get the runtime
- Sometimes, the number of statements depends on the input – we'll say the input size is N

Big O

```
statement1;                             // runtime = 1


for (int i = 1; i <= N; i++) {          // runtime = N^2
    for (int j = 1; j <= N; j++) {      // runtime = N
        statement2;
    }
}


for (int i = 1; i <= N; i++) {          // runtime = 3N
    statement3;
    statement4;
    statement5;
}

// total = N^2 + 3N + 1
```

Big O

- The actual constant doesn't matter – so we get rid of the constants: $N^2 + 3N + 1 \rightarrow N^2 + N + 1$
- Only the biggest power of N matters: $N^2 + N + 1 \rightarrow N^2$
  - The biggest term grows so much faster than the other terms that the runtime of that term "dominates"
- We would then say the code snippet has **$O(N^2)$ runtime**

Finding Big O

- Work from the innermost indented code out
- Realize that some code statements are more costly than others
  - It takes $O(N^2)$ time to call a function with runtime $O(N^2)$, even though calling that function is only one line of code
- Nested code multiplies
- Code at the same indentation level adds

What is the Big O?

```
int sum = 0;
for (int i = 1; i < 100000; i++) {
    for (int k = 1; k <= N; k++) {
        sum++;
    }
}
Vector<int> v;
for (int x = 1; x <= N; x += 2) {
    v.insert(0, x);
}
cout << v << endl;
```

Complexity Classes

- complexity class: A category of algorithmic efficiency based on the algorithm's relationship to the input size "N".

Class	Big-Oh	If you double N, ...
constant	$O(1)$	unchanged
logarithmic	$O(\log_2 N)$	increases slightly
linear	$O(N)$	doubles
log-linear	$O(N \log_2 N)$	slightly more than doubles
quadratic	$O(N^2)$	quadruples
quad-linear	$O(N^2 \log_2 N)$	slightly more than quadruple
cubic	$O(N^3)$	multiplies by 8
...	...	...
exponential	$O(2^N)$	multiplies drastically
factorial	$O(N!)$	multiplies drastically

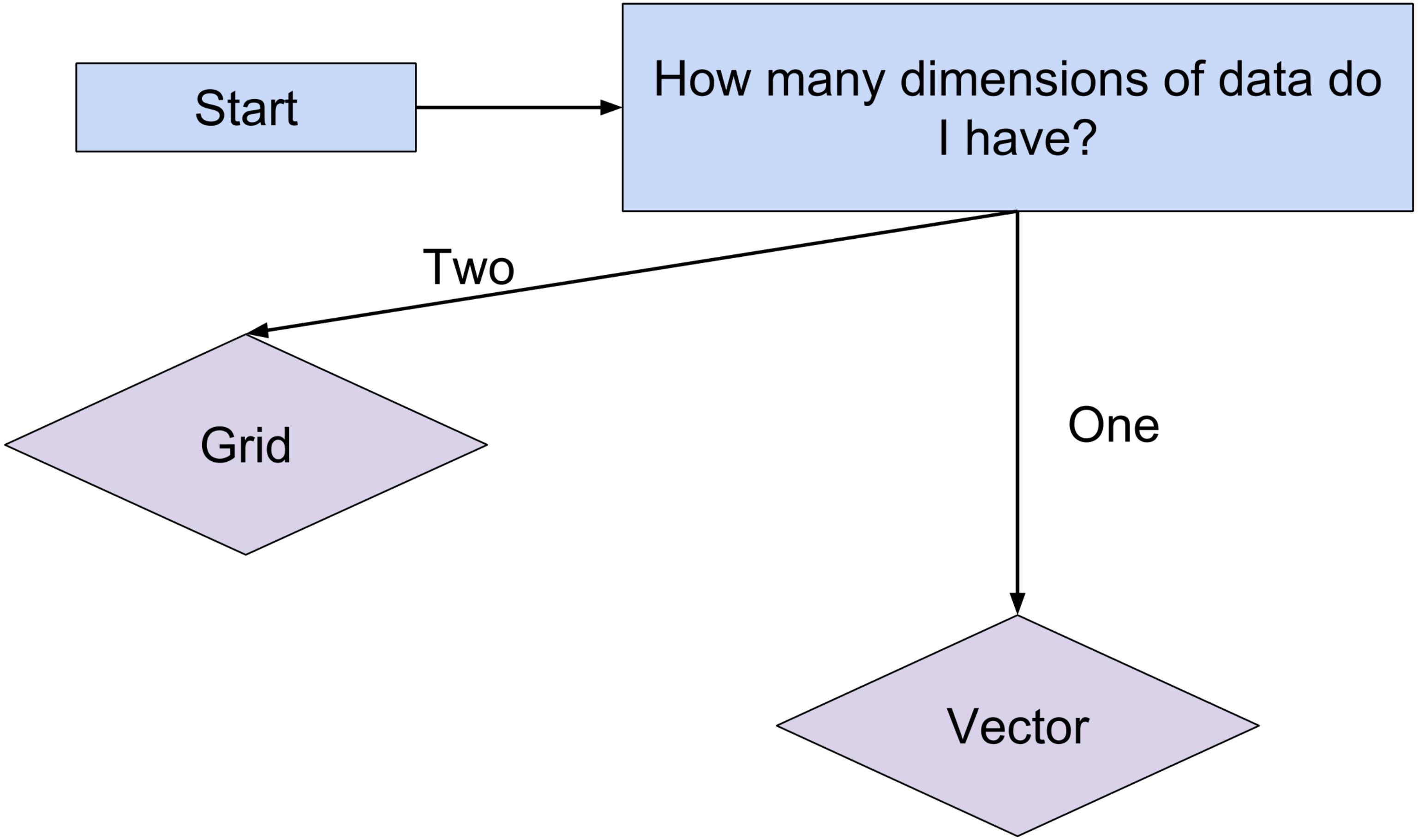
Announcements

- Style Guide
  - Function prototypes
- Only use what we have learned in class so far
- No late days charged for Assn 0
- Use the output comparison tool for Assn 1!

Plan for Today

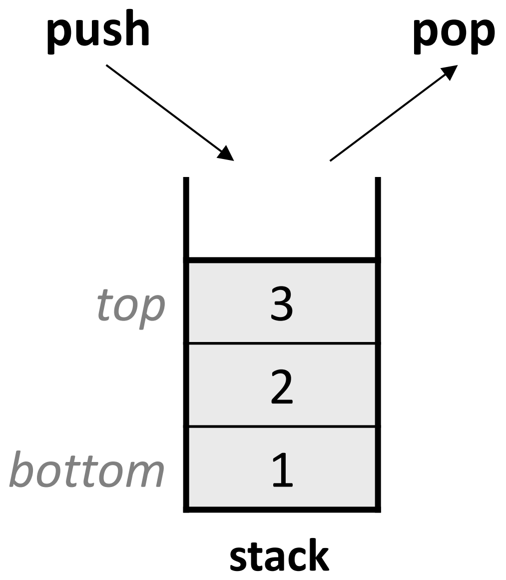
- Analyzing algorithms using **Big O** analysis
  - Understand what makes an algorithm "good" and how to compare algorithms
- Another type of collection: the **Stack**

ADTs – the Story so Far



A new ADT: the Stack

- A specialized data structure that only allows a user to add, access, and remove the **top** element
  - **"Last In, First Out" - LIFO**
  - Super fast (O(1)) for these operations
    - Built directly into the hardware
- Main operations:
  - **`push(value)`**: add an element to the top of the stack
  - **`pop()`**: remove and return the top element in the stack
  - **`peek()`**: return (but do not remove) the top element in the stack



Stack examples

- Real life
  - Pancakes
  - Clothes
  - Plates in the dining hall
- In computer science
  - Function calls
  - Keeping track of edits
  - Pages visited on a website to go back to

Stack Syntax

```
#include "stack.h"

Stack<int> nums;
nums.push(1);
nums.push(3);
nums.push(5);
cout << nums.peek() << endl; // 5
cout << nums << endl; // {1, 3, 5}
nums.pop(); // nums = {1, 3}
```

`s.isEmpty()`	O(1)	returns `true` if stack has no elements
`s.peek()`	O(1)	returns **top** value without removing it; throws an error if stack is empty
`s.pop()`	O(1)	removes **top** value and returns it; throws an error if stack is empty
`s.push(value);`	O(1)	places given value on **top** of stack
`s.size()`	O(1)	returns number of elements in stack

Stack limitations/idioms

- You cannot access a stack's elements by index.

```
Stack<int> s;
...
for (int i = 0; i < s.size(); i++) {
    do something with s[i];          // does not compile
}
```

- Instead, you pull elements out of the stack one at a time.
- **common pattern: Pop each element until the stack is empty.**

```
// process (and empty!) an entire stack
while (!s.isEmpty()) {
    do something with s.pop();
}
```

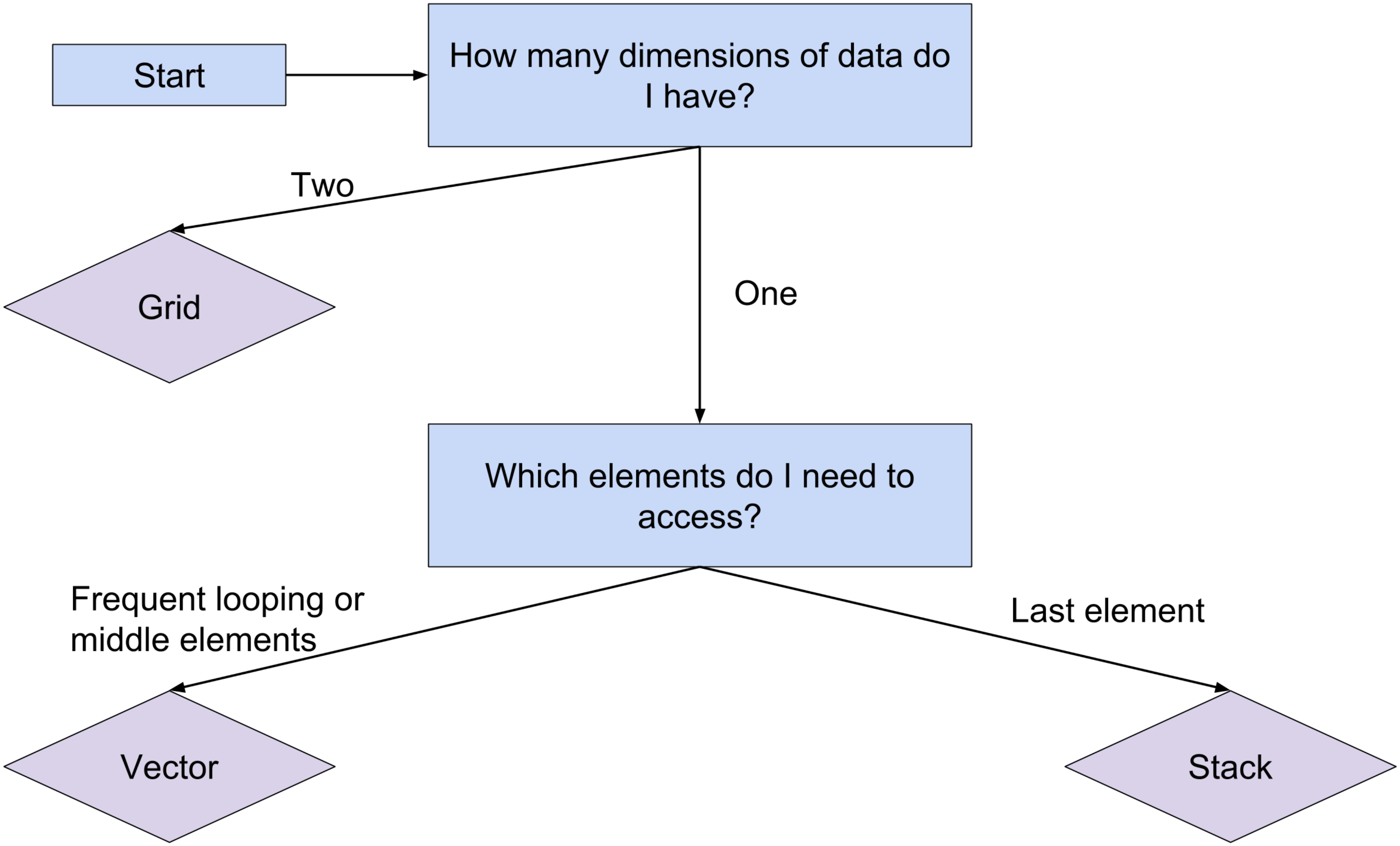
Sentence Reversal

- Goal: print the words of a sentence in reverse order
  - "Hello my name is Inigo Montoya" -> "Montoya Inigo is name my Hello"
  - "Inconceivable" -> "Inconceivable"
- Assume characters are only letters and spaces
- How could we use a `Stack`?

Sentence Reversal Solution

```
void printSentenceReverse(const string &sentence) {
    Stack<string> wordStack;
    string word = "";
    for (char c : sentence) {
        if (c == SPACE) {
            wordStack.push(word);
            word = ""; // reset
        } else {
            word += c;
        }
    }
    if (word != "") {
        wordStack.push(word);
    }
    cout << " New sentence: ";
    while (!wordStack.isEmpty()) {
        word = wordStack.pop();
        cout << word << SPACE;
    }
    cout << endl;
}
```

ADTs – the Story so Far



Look Ahead

- Assignment 1 (Game of Life) is due Wednesday, July 3, at 5PM. You can work in a pair.
- No class on July 4th
  - There is no section on July 4th either. This means section attendance for this week is optional. We will record a section on Wednesday, right after class in the same room.
  - We recommend if you have a section on Wednesday to still attend, and if you have a section on Thursday to watch the taped section online or stay after lecture on Wednesday.