Lecture 3 Handout: Running Time Analysis and Big-O practice

Last time, we derived the running time of the recursive Fibonacci: ______

- **Is this estimate too pessimistic?**
- **How well does it represent practice?**

Empirical Approach: Use data to model running time (lab01)

n	Time (ms)
40	788.09
41	1270.18
42	2070.68
43	3391.74
44	6411.54
45	9589.44
50	100329.11

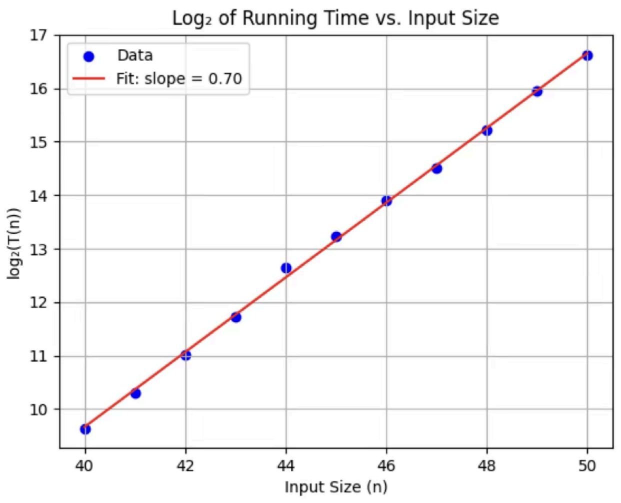
Ratios between consecutive n

n = 40 to 41: 1270.18 / 788.09 = 1.6118
n = 41 to 42: 2070.68 / 1270.18 = 1.6302
n = 42 to 43: 3391.74 / 2070.68 = 1.6378
n = 43 to 44: 6411.54 / 3391.74 = 1.8908
n = 44 to 45: 9589.44 / 6411.54 = 1.4959

Average ratio = 1.653

Observation: Time grows fast — roughly 1.6x per n.

Hypothesis: Exponential growth, like $T(n) = a * b^n$?



- How can we confirm exponential growth?
- Calculate:

$\log_2(788.09) \approx 9.62$ (n=40)
$\log_2(100329.11) \approx 16.61$ (n=50)

Slope = $(16.61 - 9.62) / (50 - 40)$
≈ 0.7

$b \approx 2^{0.7} \approx 1.62$ $a \approx 2^{-18.39}$

Why use empirical analysis?

Why use Big-O?

BigO Practice (nested loops)
Analyze the running time of buildPattern (Big-O)

```
string buildPattern(int n) {
    string result = "";
    for (int i = 0; i < n; i++) result += "x";

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < i; j++) {
              result += "y";
        }
    }
    return result;
}
```

Abstract Data Types and Operator Overloading

(15 mins) Coding Demo: arranging a music playlist using `std:: list`

(6 mins) Activity 1: `CustomList vs. std::list`

In this activity, you'll work with a simple CustomList class and compare it to the C++ Standard Library's std::list. Use the code below to guide your answers.

```cpp
class CustomList { //first try
public:
   Node* head;
   CustomList() : head(nullptr) {}
   void add(string val) {
     Node* newNode = new Node{val, nullptr};
     if (!head) {
           head = newNode;
     } else {
           Node* temp = head;
           while (temp->next) {
               temp = temp->next;
           }
           temp->next = newNode;
     }
   }
   // Note: No destructor provided
};
```

```cpp
struct Node {
   string value;
   Node* next;
};
void createPlaylist() {
  // Your code here









}
```

(4 mins) Coding Task: Complete the createPlaylist function to:

- Create a CustomList playlist.
- Add the songs "Bad," "Beat It," and "Thriller" (in that order)
- Print all songs in the playlist by traversing the list (e.g., using cout).*Hint*: You'll need to loop through the nodes starting from head.

(2 mins) Discussion Task: Imagine a friend wants to use CustomList for their music app. List two reasons why std::list might be a better choice, considering:

- *Ease of use* (e.g., built-in features, syntax).
- *Efficiency* (e.g., performance of operations).
- *Safety* (e.g., avoiding data corruption).

Abstract Data Type (ADT): A data structure defined by its operations—what it does, not how it's built.

(5 mins) Activity 2: Spot the upgrades to CustomList
Below is an improved CustomList resembling std::list.
Analyze and enhance it in two steps:

1. Annotate (3 mins): Add brief comments to each line, explaining its purpose or why it's there. Compare to the old CustomList (from Activity 1) and identify upgrades (e.g., cleaner interface, better efficiency, improved safety).

2. Extend (2 mins): Add one new method to the public section—write its declaration and a short note on its purpose. Jot down any questions about the code.

```
class CustomList { //Second try
public:
   CustomList() : head(nullptr), tail(nullptr) {}
   CustomList(std::initializer_list<string> init);
   ~CustomList();
   void push_back(const string& val);
   void push_front(const string& val);
   void pop_back();
   void pop_front();
   void clear();

   bool empty() const;

private:
   struct Node {
       string value;
       Node* next;
   };
   Node* head;
   Node* tail;

};
```

(10 mins) Operator overloading live demo

```
list<string> playlist1 = {"Bad", "Beat It"};
list<string> playlist2 = {"Heal the World"};
cout << "One playlist: ";
cout << playlist1; // No chaining
cout << "Both playlists: ";
cout << playlist1 << playlist2; // Chaining!
```

Fill in the Blanks (Main Points)

1. What does `cout << playlist1` do without overloading?
2. Write the function call for the line: `cout << playlist1`;
3. Parameter types: cout is of type __________,
   playlist is of type ____________.
4. Write the stub of the overloaded operator<< with a void return type.
5. Why does cout << playlist1 << playlist2 break with void return type?

Try this later—write the parameterized constructor with defaults and build operator+ for complex numbers!

```
class Complex {
public:
// Write the Parameterized constructor with defaults

   double getReal() const { return real; }
   double getImag() const { return imag; }
   void print() const { cout << real << " + " << imag
                              << "j" << endl; }

private:
   double real;
   double imag;
};
// Add definition for operator+ here.
// Hint: << was a function, + is too!




int main() {
  Complex x(3.0, 4.0);  // Example: 3 + 4j
 // Test constructor (3 ways), test operator+, try z = x + y



  return 0;
}
```