**Fibonachi Sequence using Memoization**

**Memoization** is an optimization technique used in dynamic programming and recursive algorithms to avoid redundant computations **by storing the results of expensive function calls and returning the cached result when the same inputs occur again.** This technique is particularly useful for recursive functions that exhibit overlapping subproblems or repeated calculations.

//reverses a string

string stringreverser(string String){

    if (String.empty()){

        return "";

    }

    cout << String.back();

    String.pop\_back();

    return stringreverser(String);}

//reverses an integer

int sum(int integer){

    if(integer < 10){

        cout << integer << endl;

        return integer;

    }

    int lastdigit = integer % 10;

    cout << lastdigit;

    int newinteger = (integer - lastdigit)/10;

    sum(newinteger);

    return 0;}

#include <iostream>

#include <unordered\_map>

using namespace std;

unordered\_map<int, int> memo; // Memoization storage

int fibonacci(int n) {

if (n <= 1) {return n;}

if (memo.find(n) != memo.end()) {

return memo[n]; }

memo[n] = fibonacci(n - 1) + fibonacci(n - 2);

return memo[n];}

class Truckloads{

    public:

    int numTrucks(int numCrates, int loadSize);

};

int Truckloads::numTrucks(int numCrates, int loadSize){

    if (numCrates <= loadSize){

        return 1; //Only one truck is needed as all the crates fit

    } else {

        int pile1 = numCrates/2;

        int pile2 = numCrates - pile1;

        return numTrucks(pile1, loadSize) + numTrucks(pile2, loadSize);}}

Move\* Human::makeMove(){

    string move;

    Decision themove;

    cin >> move;

    moveName = themove.getMove(move);

    return moveName;}

int fibonacci(int n) {//tells nth term

    if (n <= 1) {

        return n;

    }

    return fibonacci(n - 1) + fibonacci(n - 2);

}

int fibonacciSum(int n) { //sum of terms

    if (n <= 0) {

        return 0;

    }

    return fibonacci(n) + fibonacciSum(n - 1);}

A screenshot of a web page

Description automatically generated

int sumOfSquare(int n) { //return sum of squares

    if (n <= 1) {

        return 1;

    } else {

        int currentSquare = n \* n;

        int previousSum = sumOfSquare(n - 1);

        return currentSquare + previousSum;}}

int numberprinter(string characters){ //prints the first number in an integer if(characters.length() == 0){

        return 0;

    }

    if(isdigit(characters.front())){

        cout << characters.front() << endl;

        return characters.front();}

    char character = characters[1];

    if (isdigit(character)){

        cout << character;

        return 0;}

    characters.erase(0,1);

    numberprinter(characters);

    return 0;}

int factorial(int number){ //prints the factorial of a number

    if (number <= 1){

        return number;}

    int new\_number = number \* factorial(number-1);

    return new\_number;}

int power(int number,int exponent){ //return the power of a number

    if (exponent == 0){

        return 1;}

    if (exponent <= 1){

        return number;}

    int new\_number = number \*    power(number, exponent - 1);

    return new\_number;}

int summer(int number,int start){ //sums 1 + 1\*2 + 2\*3 + ... + (n-1)\*n

    if (number < 1){

        cout << number << endl;

        return number;    }

    int new\_number = start\*(start+1);

    return new\_number + summer(number-1, start+1);

}

//checks if a string is a pallindrome

string palindrome\_checker(string characters){

   if (characters.empty() || characters.front() != characters.back()){

    cout << "not palindrome";

    return "";}

   if (characters.length() == 1){

    return characters; }

   if (characters.begin() == characters.end()){

    characters.erase(0,1);

    characters.pop\_back();

    palindrome\_checker(characters);}

    return "is a palindrome";}

**The Master Theorem Cheat Sheet**

#### 1. Recurrence Form

T(n) = a \* T(n/b) + f(n)

- a: Number of recursive subproblems.

- b: Factor by which the input size is reduced.

- f(n): Non-recursive part, which includes the work done outside of recursive calls.

#### 2. Calculate n^log\_b(a)

n^log\_b(a)

#### 3. Compare f(n) to n^log\_b(a)

- If f(n) is O(n^log\_b(a-ε)) for some ε > 0, then T(n) = Θ(n^log\_b(a)).

- If f(n) is Θ(n^log\_b(a)), then T(n) = Θ(n^log\_b(a) \* log(n)).

- If f(n) is Ω(n^log\_b(a+ε)) for some ε > 0, and if a\*f(n/b) <= k\*f(n) for some k < 1 and sufficiently large n, then T(n) = Θ(f(n)).

#### 4. Determine the Complexity

- Compare f(n) to n^log\_b(a) and use the appropriate case to find the time complexity:

- Case 1: T(n) = Θ(n^log\_b(a))

- Case 2: T(n) = Θ(n^log\_b(a) \* log(n))

- Case 3: T(n) = Θ(f(n))

### Examples:

#### Example 1: T(n) = 2\*T(n/2) + n

- a = 2, b = 2, f(n) = n

- n^log\_b(a) = n^1 = n

- f(n) is Θ(n), which matches n^log\_b(a).

- Therefore, the complexity is T(n) = Θ(n \* log(n)).

#### Example 2: T(n) = 3\*T(n/4) + n^2

- a = 3, b = 4, f(n) = n^2

- n^log\_b(a) = n^(log\_4(3))

- f(n) is Θ(n^2), which is greater than n^(log\_4(3)).

- Therefore, the complexity is T(n) = Θ(n^2).

#### Example 3: T(n) = 2\*T(n/2) + n^2 \* log(n)

- a = 2, b = 2, f(n) = n^2 \* log(n)

- n^log\_b(a) = n^1 = n

- f(n) is Ω(n^1), and a\*f(n/b) = 2\*(n/2)^2\*log(n/2) = n^2 \* (log(n) - 1).

- Since a\*f(n/b) <= k\*f(n) for k = 1/2 < 1, we fall into Case 3.

- Therefore, the complexity is T(n) = Θ(n^2 \* log(n)).

Remember that the Master Theorem applies to specific types of recurrences and may not cover all cases. When in doubt, you may need to use other methods like the Recursion Tree or Substitution method for analysis.

**Vectors Cheat Sheet**

Before you can use vectors, you need to include the `<vector>` header:

```cpp

#include <vector>

### Declaring a Vector

You can declare a vector using the following syntax:

```cpp

std::vector<datatype> vectorName;

For example, to declare an integer vector:

```cpp

#include <vector>

std::vector<int> myVector;

```

### Initializing a Vector

You can initialize a vector in several ways:

1. \*\*Empty Initialization\*\*: Create an empty vector.

```cpp

std::vector<int> myVector;

```

2. \*\*Initialization with Size\*\*: Create a vector with a specified size and default-initialized elements.

```cpp

std::vector<int> myVector(5); // Creates a vector with 5 elements, all initialized to 0

```

3. \*\*Initialization with Values\*\*: Create a vector and initialize it with specific values.

```cpp

std::vector<int> myVector = {1, 2, 3, 4, 5};

```

### Accessing Elements

You can access vector elements using the `[]` operator or the `at()` function. Remember that vectors are zero-indexed.

```cpp

std::vector<int> myVector = {1, 2, 3, 4, 5};

int element1 = myVector[0]; // Access the first element (1)

int element2 = myVector.at(2); // Access the third element (3)

```

### Modifying Elements

You can modify vector elements just like you access them:

```cpp

std::vector<int> myVector = {1, 2, 3, 4, 5};

myVector[0] = 10; // Modify the first element (1 -> 10)

myVector.at(2) = 30; // Modify the third element (3 -> 30)

```

### Vector Functions and Operations

Vectors provide several useful functions and operations:

- `push\_back()`: Add an element to the end of the vector.

- `pop\_back()`: Remove the last element from the vector.

- `size()`: Get the number of elements in the vector.

- `empty()`: Check if the vector is empty.

- `clear()`: Remove all elements from the vector.

- `resize()`: Change the size of the vector.

### Iterating Through a Vector

You can iterate through a vector using a `for` loop or iterators:

```cpp

std::vector<int> myVector = {1, 2, 3, 4, 5};

// Using a for loop

for (int i = 0; i < myVector.size(); i++) {

std::cout << myVector[i] << " ";

}

// Using iterators (C++11 or later)

for (const auto& element : myVector) {

std::cout << element << " ";

}

```

### Vector of Custom Data Types

You can create vectors of custom data types (e.g., structures or classes) in the same way you create vectors of built-in data types. Just replace `datatype` with your custom type.

```cpp

struct Person {

std::string name;

int age;

};

std::vector<Person> people; // Vector of Person objects

```

These are the basics of using vectors in C++. Vectors are a powerful data structure for dynamically managing collections of elements.

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedA close-up of a text

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

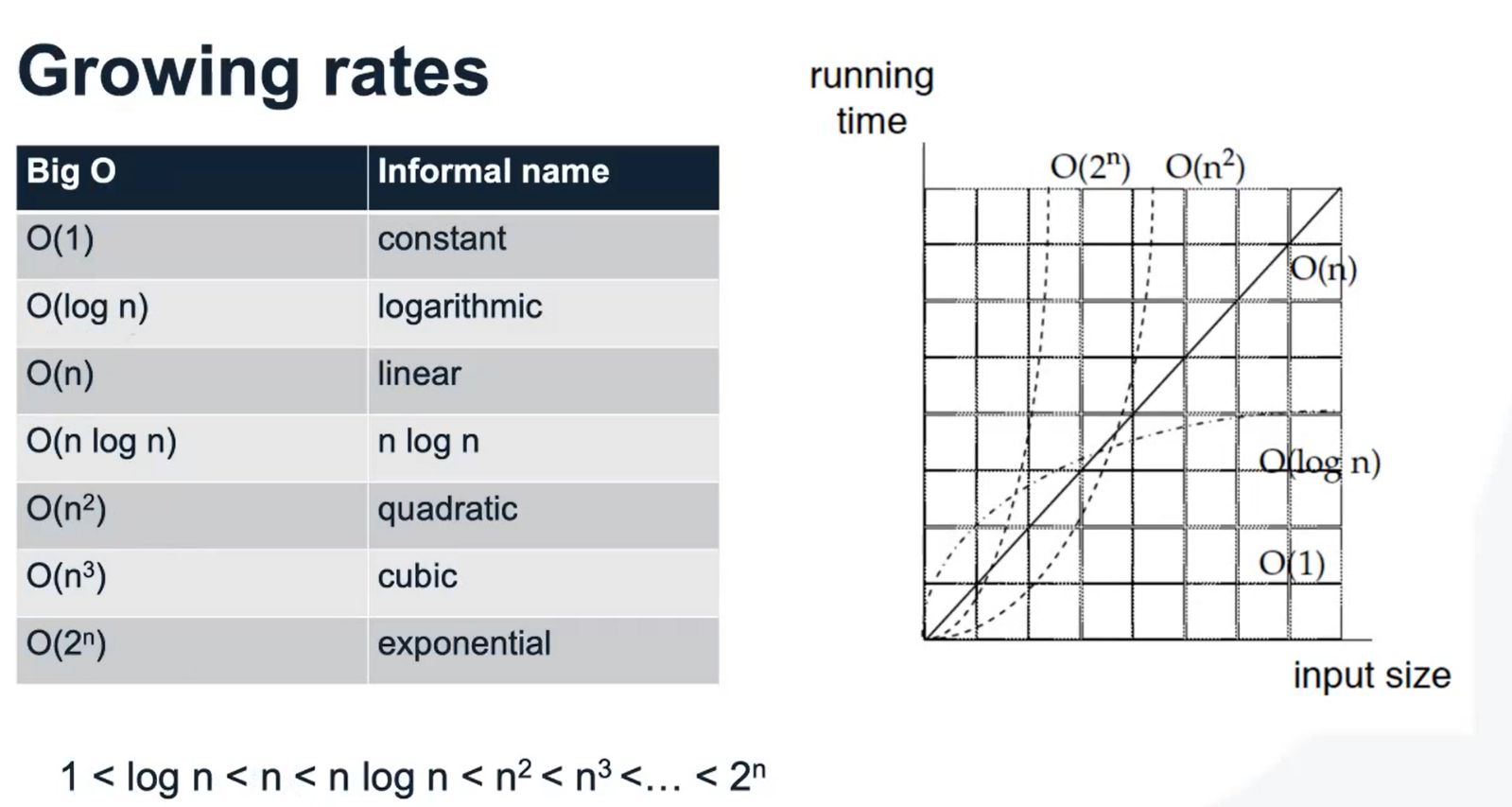
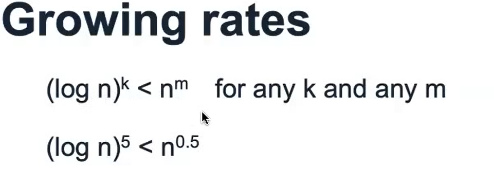
A screenshot of a computer

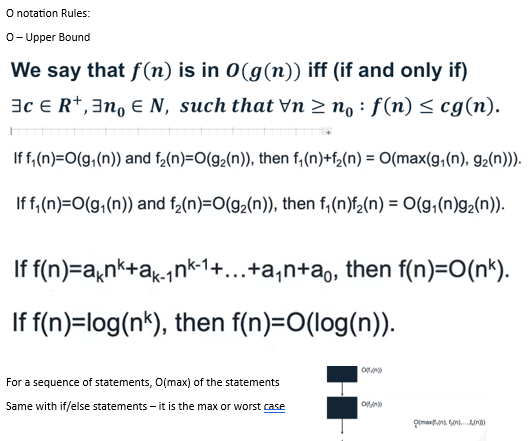
Description automatically generated

A close-up of a math problem

Description automatically generatedA screenshot of a math test

Description automatically generated



A screenshot of a graph

Description automatically generated

A white paper with black text

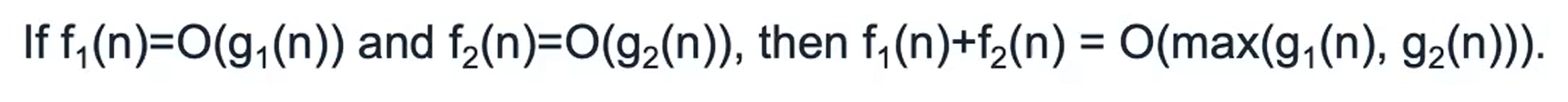
Description automatically generated

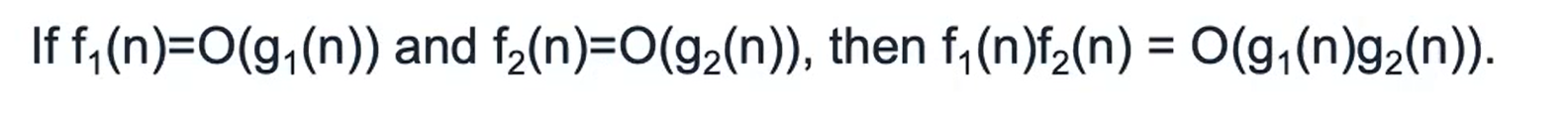
O notation Rules:

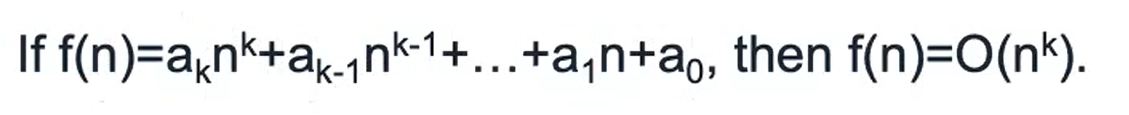
O – Upper Bound

A close up of a math problem

Description automatically generated with medium confidence







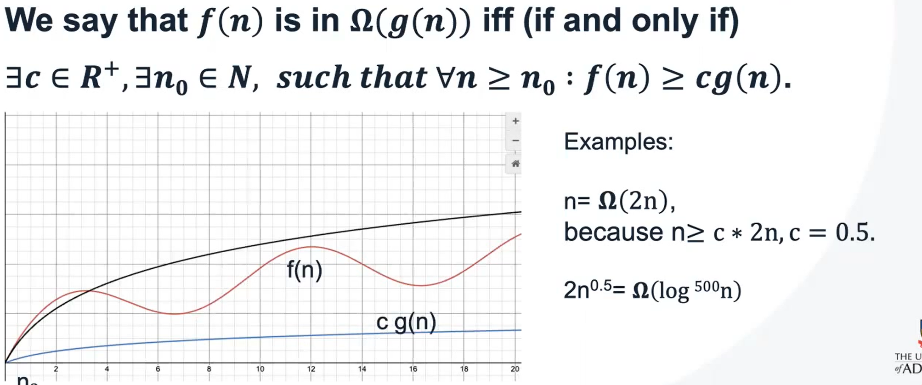
A screenshot of a computer

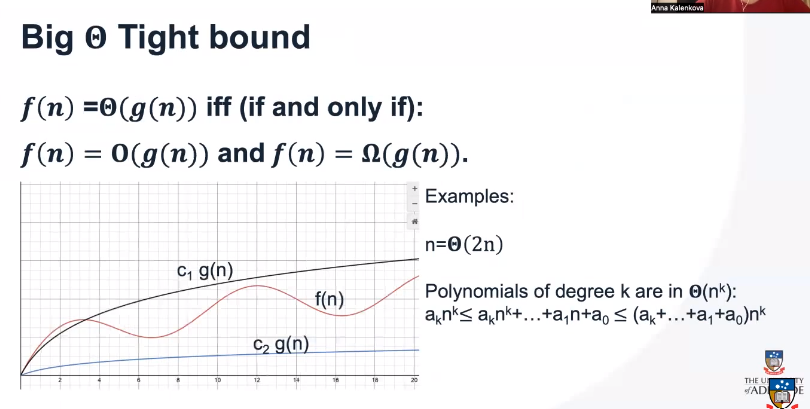
Description automatically generatedA close up of a word

Description automatically generated

For a sequence of statements, O(max) of the statements

Same with if/else statements – it is the max or worst case

Omega Ω – Lower Bound

 Tight Bound when f(n) = g(n) and they only differ by a coefficient.

We use master theorem as a method of solving for computational complexity, by breaking the algorithm into sub problems (of size n/b) and by iterating it through ‘a’ recursive calls. F(n) is used to calculate the computational complexity when combining all of the sub-problems.

A math equations on a white background

Description automatically generated