# CSCI 200: Foundational Programming Concepts & Design Lecture 37

Recursion[1]

&

Merge Sort

[1] Recursion, *see Recursion.*

# Previously in CSCI 200

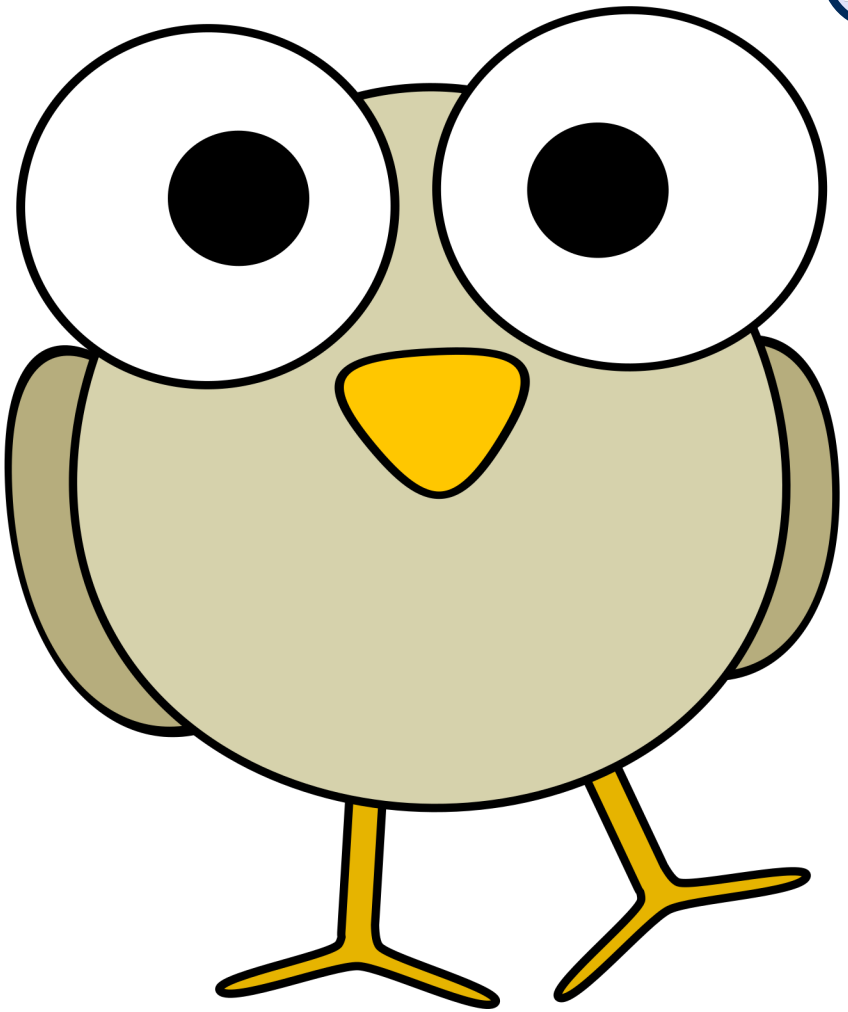
- Sorting Algorithms
  - Selection Sort
  - Insertion Sort
  - Bubble Sort

# Sorting Complexities

| Algorithm | Worst Case | Best Case | Average Case |
|---|---|---|---|
| Selection Sort | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ |
| Insertion Sort | $O(n^2)$ | $O(n)$ | $O(n^2)$ |
| Bubble Sort | $O(n^2)$ | $O(n)$ | $O(n^2)$ |
| | | | |

- Not ideal

# Questions?



# Learning Outcomes For Today

- Explain how sorting a list affects the performance of searching for a value in a list.
- Define recursion.
- Explain the meaning of a stopping condition, the base case, and the recursive case.
- Evaluate the resultant output of a given code containing a recursive function.
- Write a program that implements the pseudocode and solves the recursive problem.
- Implement the merge sort algorithm using recursion.
- Define recursion & unwinding.

# On Tap For Today

- Sorting
  - Merge Sort
- Recursion
- Recursive Merge Sort
- Practice

# On Tap For Today

- Sorting
  - Merge Sort
- Recursion
- Recursive Merge Sort
- Practice

# Merge Sort Idea

1. Split list in half
2. Sort each half
3. Merge the two halves

# Merge Sort Idea

1. Split list in half
2. Sort each half
   - for each half

3. Merge the two halves

# Merge Sort Idea

1. Split list in half
2. Sort each half
   - for each half
     1. Split half in half (into quarters)
     2. Sort each quarter
     3. Merge the two quarters
3. Merge the two halves

# Merge Sort Idea

1. Split list in half
2. Sort each half
   - for each half
     1. Split half in half (into quarters)
     2. Sort each quarter
     3. Merge the two quarters
3. Merge the two halves

# Merge Sort Idea

1. Split list in half
2. Sort each half
   - for each half
     1. Split half in half (into quarters)
     2. Sort each quarter
     3. Merge the two quarters
3. Merge the two halves

# Merge Sort Idea

1. Split list in half
2. Sort each half
   - for each half
     1. Split half in half (into quarters)
     2. Sort each quarter
     3. Merge the two quarters
3. Merge the two halves

# Merge Sort Idea

1. Split list in half
2. Sort each half
   - for each half
     1. Split half in half (into quarters)
     2. Sort each quarter
     3. Merge the two quarters
3. Merge the two halves

- Defined in terms of itself →Recursion!

# On Tap For Today

- Sorting
  - Merge Sort
- Recursion
- Recursive Merge Sort
- Practice

# Recursive _____

1. Recursive Data Structures

2. Recursive Functions

# Node Struct

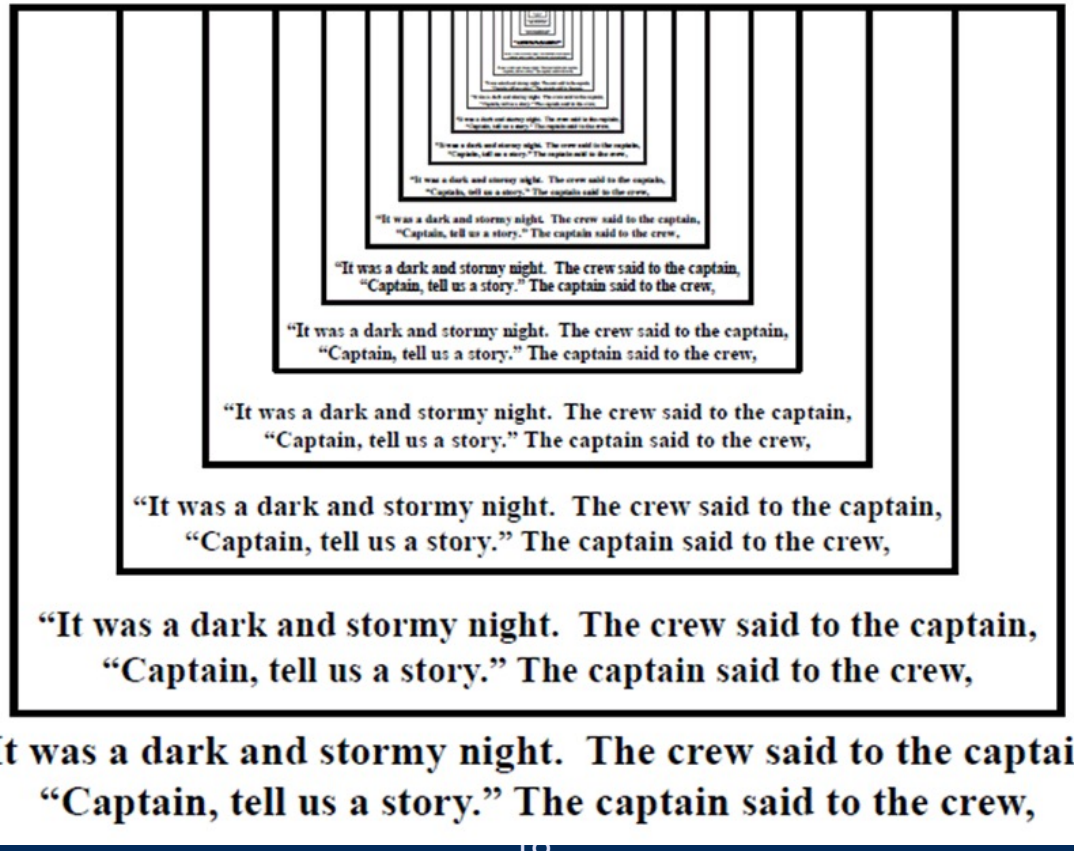
- A “recursive” data structure

```
template<typename T>
struct Node {
  T value;
  Node<T> *pNext;
  Node<T> *pPrev;
};
```

- Recursive Data Structure:
  - Defined in terms of itself, contains reference to itself
  - composed of instances of the same data structure

# Recursive Functions

- Simply put
  - Are functions that call themselves



# Simplest Example!

```
void recursiveFunction() {
    recursiveFunction();          // recursion!
}
```

# Better Example!

```
void countDown( int counter ) {
     cout << counter << endl;
     countDown( counter - 1 );      // recursion!
}


int main() {
     cout << "Let's recurse!" << endl;
     countDown( 10 );
     cout << "WOW!  That was fun." << endl;
     return 0;
}
```

**What's the problem?**

# Infinite Recursion ☹

- Recursion simulates loops
  - Without a stopping condition, a loop will iterate forever
- Recursive functions without a stopping condition…
  - …will recurse forever forever forever forever forever forever forever…

# Better Example!

```
void countDown( int counter ) {
    if( counter < 0 ) {                 // stopping condition
        return;
    } else {
        cout << counter << endl;
        countDown( counter - 1 );       // recursion!
    }
}
int main() {
    cout << "Let's recurse!" << endl;
    countDown( 10 );
    cout << "WOW!  That was fun." << endl;
    return 0;
}
```

# Better Example!

```
void countDown( int counter ) {
    if( counter < 0 ) {          // stopping condition
        return;
    }
    else {
        cout << counter << endl;
        countDown( counter - 1 );
    }
}
```

Base Case

Recursive Case

# Alternative Recursive Definition

- Defined in terms of itself
  - Function is applied within its own definition
- (Poor*) Math Examples
  - Exponent / Factorial / Fibonacci
- (Poor*) CS Example
  - isPalindrome

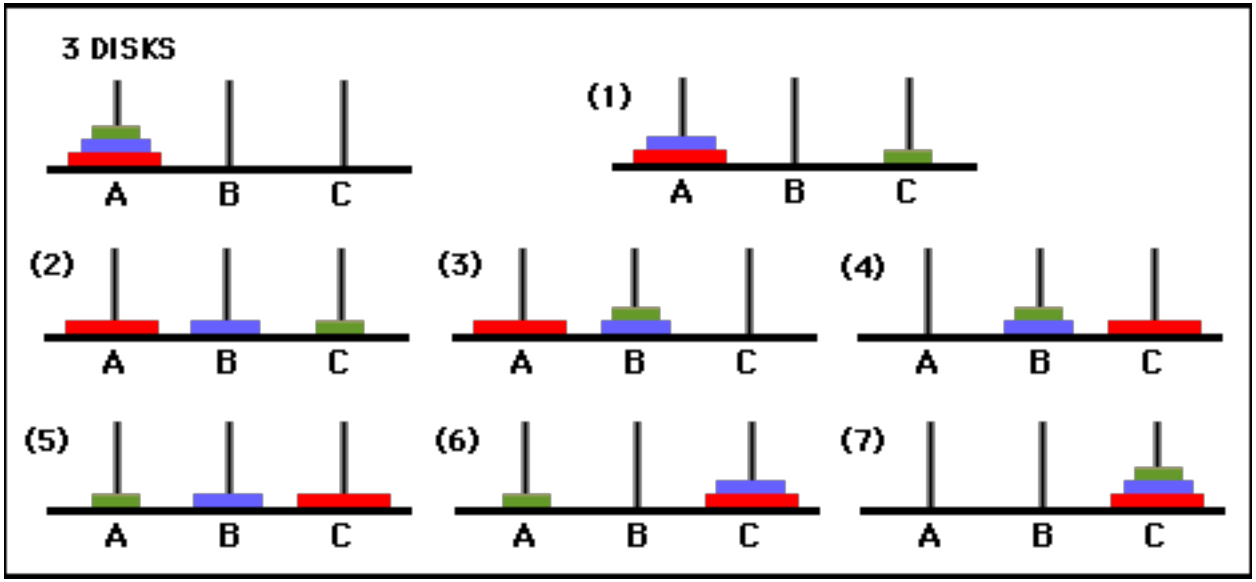
# Better Recursive Definition

- Defined in terms of itself
  - Function is applied within its own definition

- Solve a problem by solving a smaller instance of the same problem

# Better Example

- Towers of Hanoi



# Better Recursive Definition

- Defined in terms of itself
  - Function is applied within its own definition
- Solve a problem by solving a smaller instance of the same problem
  - Divide-and-conquer
  - Decrease-and-conquer

# Divide-and-Conquer

- Divide
  - Break big problem into smaller problems of the same type until it is trivial to solve
- Conquer
  - Combine sub-solutions to form solution to original problem

# Merge Sort

- Sorts in a divide-and-conquer fashion
  - Divide: Split the list in half until sublist is of size 1 (which is naturally already sorted)
  - Conquer: Merge the two sorted lists by grabbing the smaller front element (via insertion sort)

# On Tap For Today

- Sorting
  - Merge Sort
- Recursion
- Recursive Merge Sort
- Practice

# Merge Sort Pseudocode

```
function mergeSort( List list ) {




}
```

# Merge Sort Pseudocode

```
function mergeSort( List list ) {
  // base case
  if( list.size() <= 1) {} // do nothing, it's already sorted
  // recursive case
  else {
    // divide
    List halves[2] = split(list)
    // recurse
    mergeSort(halves[0]) // first half
    mergeSort(halves[1]) // second half
    // conquer
    list = merge(halves[0], halves[1])
  }
}
```

# LinkedList **Pseudocode** (POP Style)

```
template<typename T>
void merge_sort( LinkedList<T>* const P_list  ) {
  // base case
  if(P_list == nullptr
       || P_list->size() <= 1) {return;} // already sorted
  // divide & split
  LinkedList<T> *pLeft = new LinkedList<T>,
       *pRight = new LinkedList<T>;
  split_list(P_list, pLeft, pRight);
      // P_list now empty, pLeft & pRight hold both halves
  // recurse
  merge_sort(pLeft);
  merge_sort(pRight);
  // conquer & merge
  merge_lists(pLeft, pRight, P_list);
      // P_list now sorted, pLeft & pRight empty
}
```

# LinkedList **Pseudocode** (OOP Style)

```
template<typename T>
void LinkedList<T>::mergeSort() {
  // base case
  if(_size <= 1) {return;} // already sorted

  // divide & split
  LinkedList<T> *pLeft = new LinkedList<T>,
       *pRight = new LinkedList<T>;
  _splitList(pLeft, pRight);
      // callee now empty, pLeft & pRight hold both halves
  // recurse
  pLeft->mergeSort();
  pRight->mergeSort();
  // conquer & merge
  _mergeLists(pLeft, pRight);
      // callee now sorted, pLeft & pRight empty
}
```

# Naïve Cost Analysis

- $T(n) = 2T(n/2) + n$
- $T(1) = 1$
  - Follows the form of the "Master Theorem"

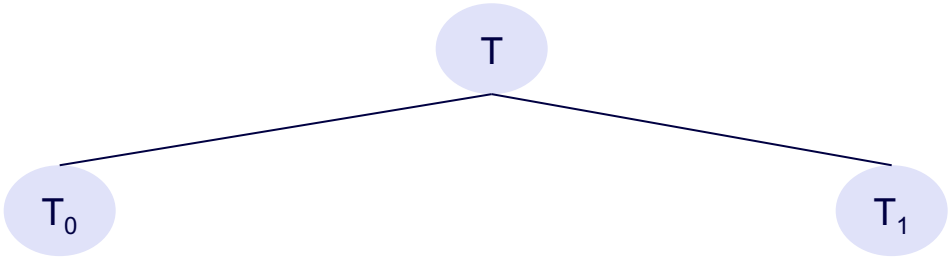
# Naïve Cost Analysis

- $T(n) = 2T(n/2) + n$
- $T(1) = 1$
  - Follows the form of the "Master Theorem"

T

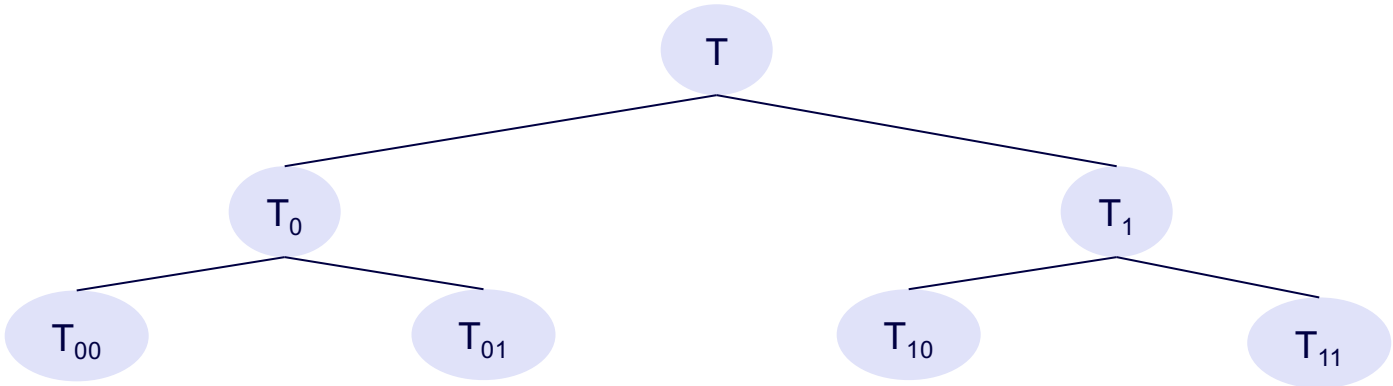
# Naïve Cost Analysis

- $T(n) = 2T(n/2) + n$
- $T(1) = 1$
  - Follows the form of the "Master Theorem"



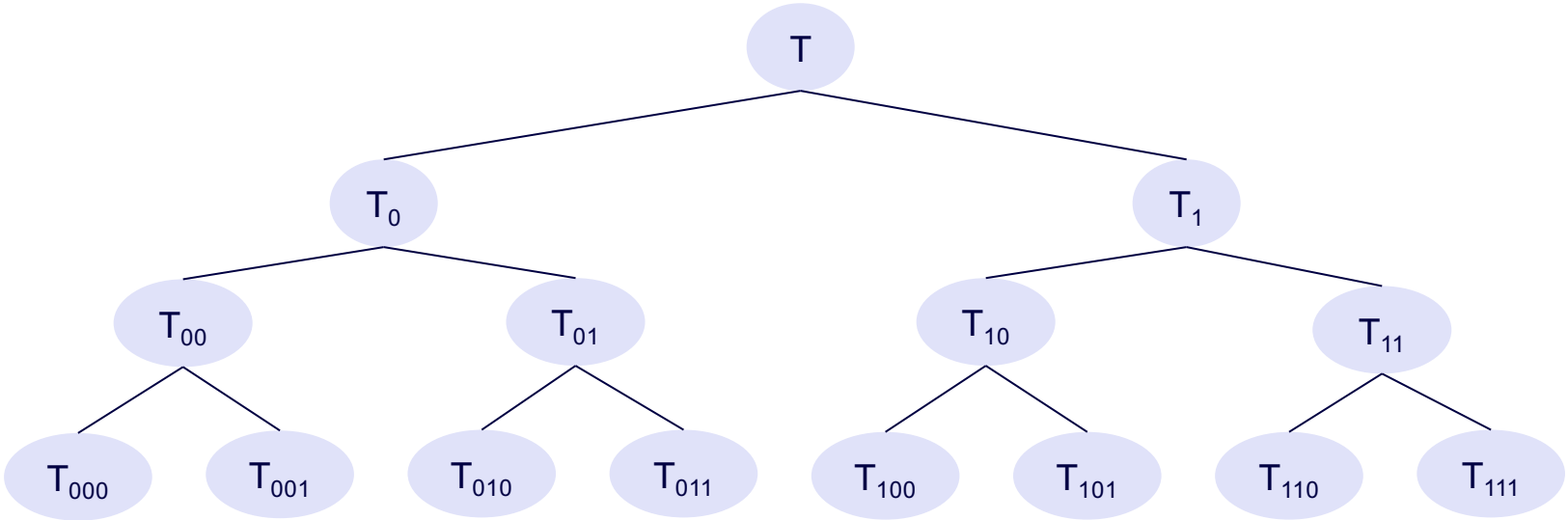
# Naïve Cost Analysis

- T(n) = 2T(n/2) + n
- T(1) = 1
  - Follows the form of the “Master Theorem”



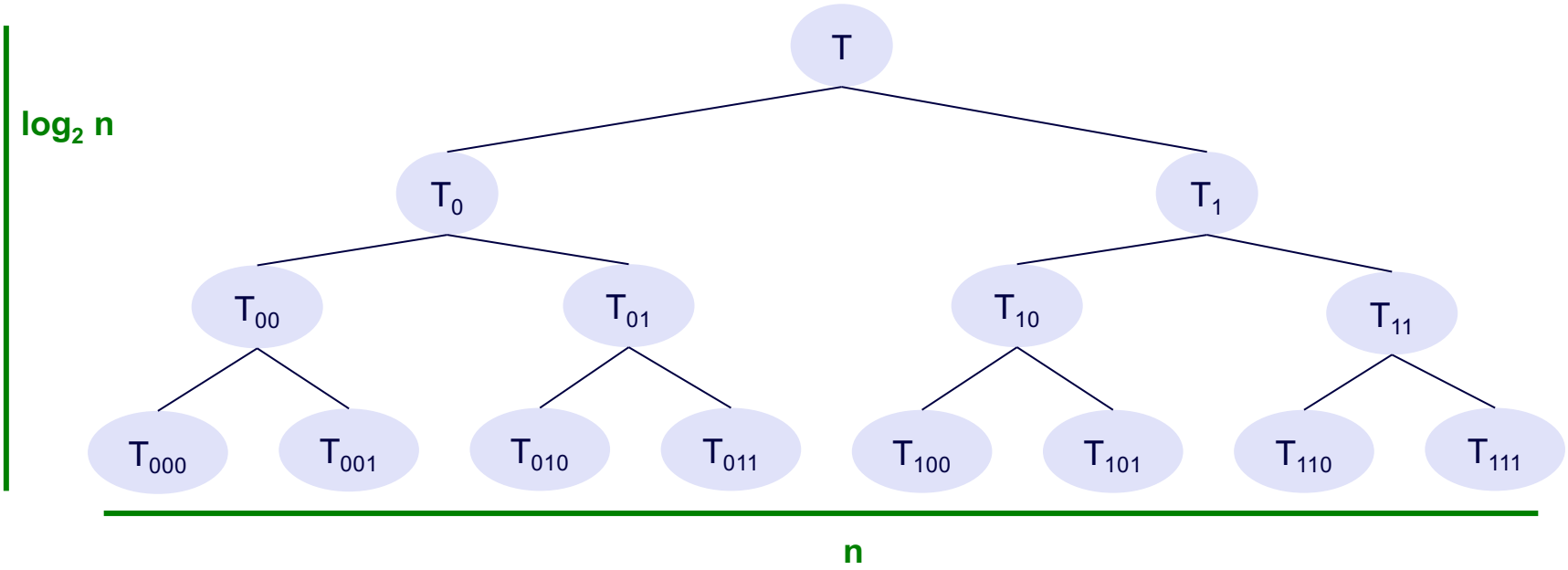
# Naïve Cost Analysis

- T(n) = 2T(n/2) + n
- T(1) = 1
  - Follows the form of the "Master Theorem"



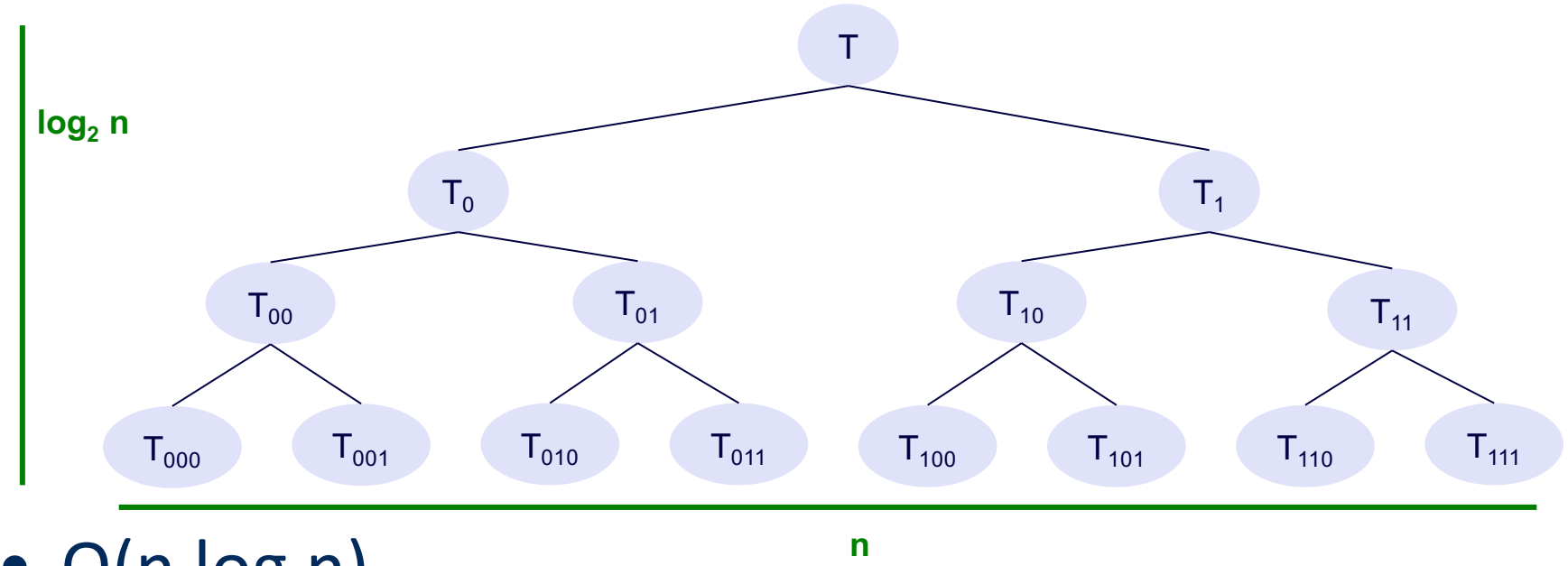
# Naïve Cost Analysis

- T(n) = 2T(n/2) + n
- T(1) = 1
  - Follows the form of the "Master Theorem"



# Naïve Cost Analysis

- T(n) = 2T(n/2) + n
- T(1) = 1
  - Follows the form of the "Master Theorem"




- O(n log n)

# Sorting Complexities

| Algorithm | Worst Case | Best Case | Average Case |
|---|---|---|---|
| Selection Sort | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ |
| Insertion Sort | $O(n^2)$ | $O(n)$ | $O(n^2)$ |
| Bubble Sort | $O(n^2)$ | $O(n)$ | $O(n^2)$ |
| Merge Sort | $O(n \log n)$ | $O(n \log n)$ | $O(n \log n)$ |

# Sorting Complexities

| Algorithm | Worst Case | Best Case | Average Case |
|---|---|---|---|
| Selection Sort | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ |
| Insertion Sort | $O(n^2)$ | $O(n)$ | $O(n^2)$ |
| Bubble Sort | $O(n^2)$ | $O(n)$ | $O(n^2)$ |
| Merge Sort | $O(n \log n)$ | $O(n \log n)$ | $O(n \log n)$ |

- In practice, choose threshold ($T$) based on task
  - If $n < T$, perform insertion sort
  - Else perform merge sort
- $O(T) < O(T^2) < O(n \log n)$

# On Tap For Today

- Sorting
  - Merge Sort
- Recursion
- Recursive Merge Sort
- Practice

# To Do For Next Time

- Have a great Thanksgiving!