CSCI 200: Foundational Programming Concepts & Design Lecture 38

Searching Algorithms

Previously in CSCI 200

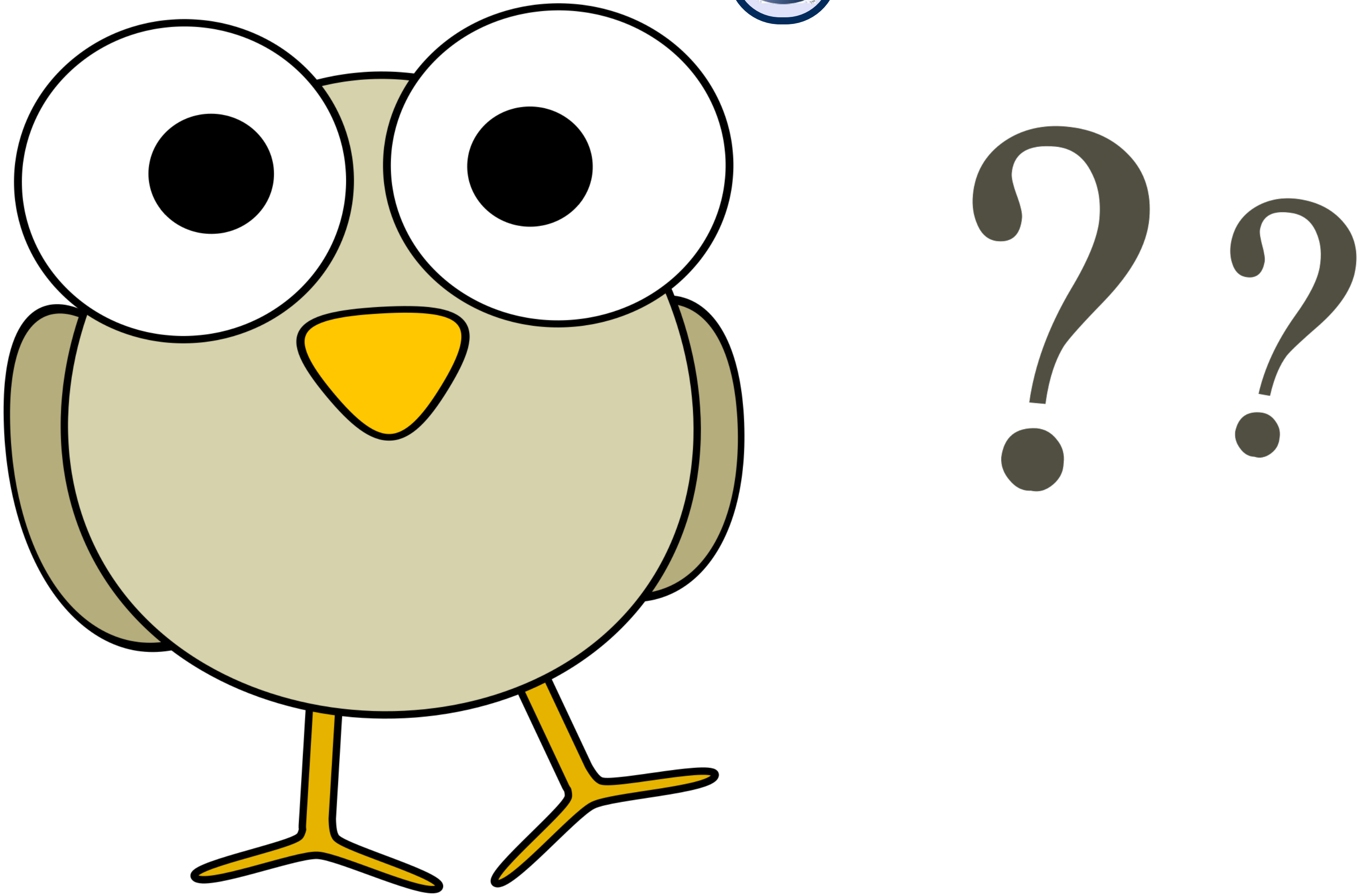
- Merge Sort
  - split()
  - merge()

- Recursion
  - Defined in terms of self
  - Solve smaller version of same problem
    - Divide-and-Conquer
    - Decrease-and-Conquer

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	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

Questions?



Learning Outcomes For Today

- Explain how sorting a list affects the performance of searching for a value in a list.
- Implement linear and binary search.

On Tap For Today

- Searching
  - Linear Search
  - Binary Search
- Practice

On Tap For Today

- Searching
  - Linear Search
  - Binary Search
- Practice

Searching

- Different Types of Searches

Linear (unordered list)

Binary (ordered list)

On Tap For Today

- Searching
  - Linear Search
  - Binary Search
- Practice

Linear Search

- No knowledge of list contents
  - No requirement of list ordering
    - List is unsorted (or sorted)

Algorithm 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	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

Algorithm	Worst Case	Best Case	Average Case
Linear Search			

Algorithm 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	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

Algorithm	Worst Case	Best Case	Average Case
Linear Search	$O(n)$	$O(1)$	$O(n)$

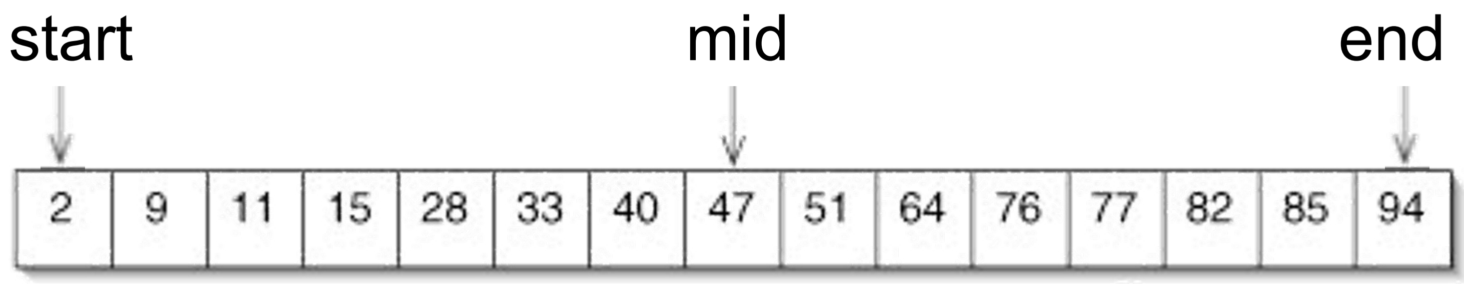
On Tap For Today

- Searching
  - Linear Search
  - Binary Search
- Practice

Binary Search

- List must be sorted

Binary Search



Binary Search

- Values to keep track of
  - The list to search
  - Target value
  - Index to start search
  - Index to end search

Binary Search Pseudocode

- Examine middle element
  - If equals target
    - item found, return location
  - If greater than target
    - Ignore top half of list, continue search on bottom half
  - If less than target
    - Ignore bottom half of list, continue search on top half
- Repeat until
  - Target is found
  - start and end cross (not found)

Binary Search Pseudocode

- Examine middle element
  - If equals target
    - item found, return location
  - If greater than target
    - Ignore top half of list, **continue search on bottom half**
  - If less than target
    - Ignore bottom half of list, **continue search on top half**
- Repeat until
  - Target is found
  - start and end cross (not found)

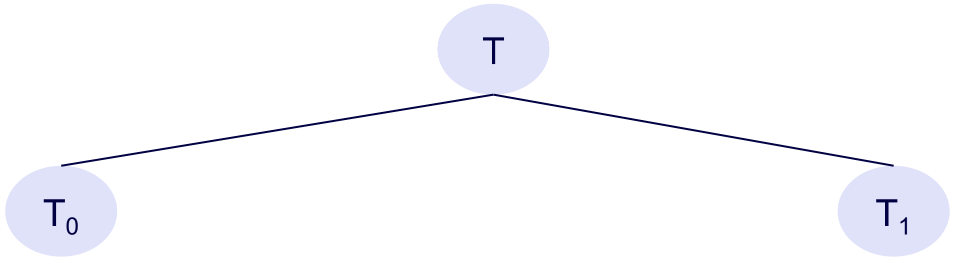
Sounds Recursive!

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

T

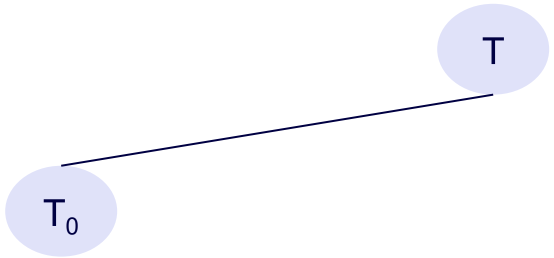
Sounds Recursive!

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



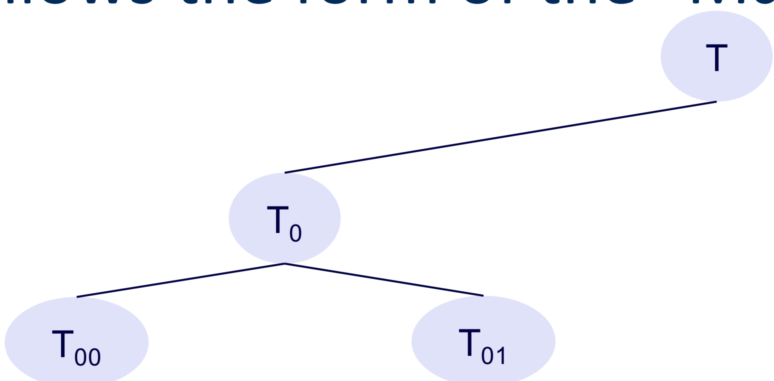
Sounds Recursive!

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



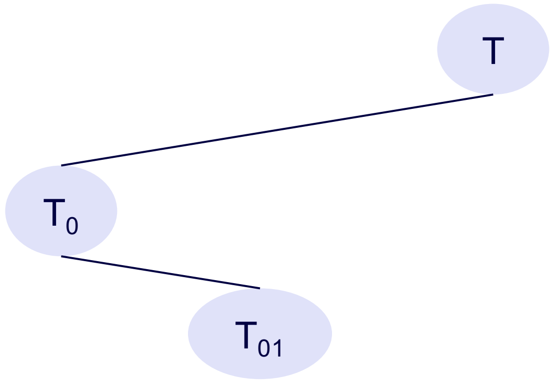
Sounds Recursive!

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



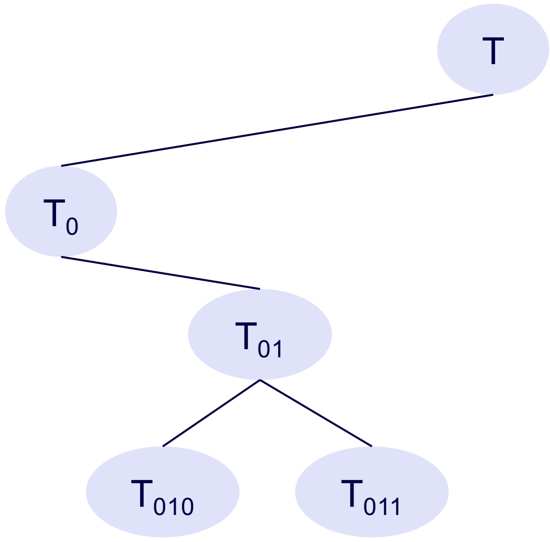
Sounds Recursive!

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



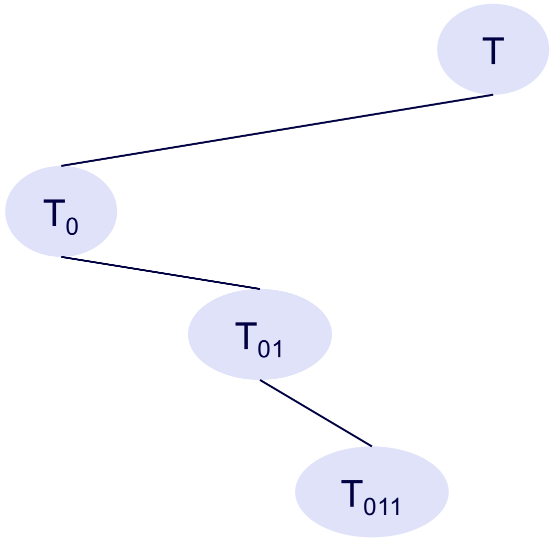
Sounds Recursive!

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



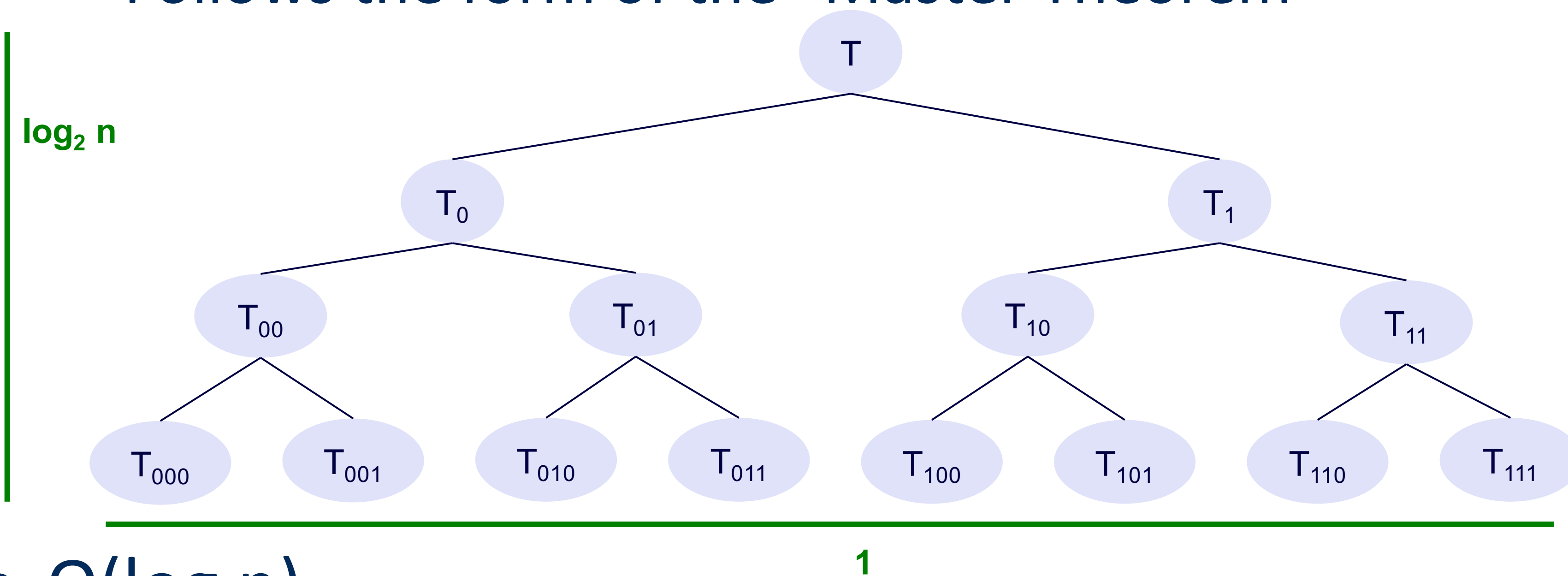
Sounds Recursive!

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



Sounds Recursive!

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



- O(log n)

Algorithm 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	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

Algorithm	Worst Case	Best Case	Average Case
Linear Search	$O(n)$	$O(1)$	$O(n)$
Binary Search			

Algorithm 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	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

Algorithm	Worst Case	Best Case	Average Case
Linear Search	$O(n)$	$O(1)$	$O(n)$
Binary Search	$O(\log n)$	$O(1)$	$O(\log n)$

Algorithm 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	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

Algorithm	Worst Case	Best Case	Average Case
Linear Search	$O(n)$	$O(1)$	$O(n)$
Binary Search	$O(\log n)$	$O(1)$	$O(\log n)$

- Other search algorithms exist
  (just as other sort algorithms exist too)

Binary Search

- Values to keep track of
  - The array to search, Target value, Index to start search, Index to end search
- Examine middle element
  - If equals target
    - item found, return location
  - If end < start
    - Item not found
  - If greater than target
    - Ignore top half of list, continue search on bottom half
  - If less than target
    - Ignore bottom half of list, continue search on top half

Binary Search

- Values to keep track of
  - The array to search, Target value, Index to start search, Index to end search
- Examine middle element
  - If equals target
    - item found, return location
  - If end < start
    - Item not found
  - If greater than target
    - Ignore top half of list, **<u>continue search on bottom half</u>**
  - If less than target
    - Ignore bottom half of list, **<u>continue search on top half</u>**

Binary Search

- Values to keep track of
  - The array to search, Target value, Index to start search, Index to end search
- Examine middle element
  - If equals target
    - item found, return location
  - If end < start
    - Item not found
  - If greater than target
    - Ignore top half of list, **continue search on bottom half**
  - If less than target
    - Ignore bottom half of list, **continue search on top half**
- Decrease-and-Conquer

Binary Search

- Values to keep track of
  - The array to search, Target value, Index to start search, Index to end search
- Examine middle element
  - If equals target
    - item found, return location
  - If end < start
    - Item not found
  - If greater than target
    - Ignore top half of list, **continue search on bottom half**
  - If less than target
    - Ignore bottom half of list, **continue search on top half**
- Decrease-and-Conquer
- So tempting to make recursive function…

Recursive Binary Search

```
template<typename T>
int binary_search(const List<T>& LIST, const T TARGET,
                  const int START_POS, const int END_POS) {
  const int MIDDLE_POS = (END_POS - START_POS) / 2 + START_POS;
  if(END_POS < START_POS)
    return -1;
  if(LIST[MIDDLE_POS] == TARGET)
    return MIDDLE_POS;
  if(LIST[MIDDLE_POS] > TARGET)
    return binary_search(LIST, TARGET, START_POS, MIDDLE_POS - 1);
  if(LIST[MIDDLE_POS] < TARGET)
    return binary_search(LIST, TARGET, MIDDLE_POS + 1, END_POS);
}

int targetPos = binary_search(myList, target, 0, myList.size() - 1);
```

Recursive Binary Search

```
template<typename T>
int binary_search(const List<T>& LIST, const T TARGET,
                  const int START_POS, const int END_POS) {
  const int MIDDLE_POS = (END_POS - START_POS) / 2 + START_POS;
  if(END_POS < START_POS)
    return -1;
  if(LIST[MIDDLE_POS] == TARGET)
    return MIDDLE_POS;
  if(LIST[MIDDLE_POS] > TARGET)
    return binary_search(LIST, TARGET, START_POS, MIDDLE_POS - 1);
  if(LIST[MIDDLE_POS] < TARGET)
    return binary_search(LIST, TARGET, MIDDLE_POS + 1, END_POS);
}

int targetPos = binary_search(myList, target, 0, myList.size() - 1);
```

- Concern/danger of recursion?

The Call Stack

```
void print_space(const int N) {
   for(int i = 0; i < N; i++) cout << " ";
}
void recurse( const int N ) {
   if(N <= 1) {
      print_space(N);  cout << "Done!" << endl;
   } else {
      print_space(N);  cout << "Start " << N << endl;
      recurse(N-1);
      print_space(N);  cout << "End " << N << endl;
   }
}
int main() {
   recurse(6);
   return 0;
}
```

The Call Stack

```
void print_space(const int N) {
  for(int i = 0; i < N; i++) cout << " ";
}
void recurse( const int N ) {
  if(N <= 1) {
    print_space(N);  cout << "Done!" << endl;
  } else {
    print_space(N);  cout << "Start " << N << endl;
    recurse(N-1);
    print_space(N);  cout << "End " << N << endl;
  }
}
int main() {
  recurse(6);
  return 0;
}
```

```
./recurse
      Start 6
     Start 5
    Start 4
   Start 3
  Start 2
 Done!
  End 2
   End 3
    End 4
     End 5
      End 6
```

The Call Stack

```
void print_space(const int N) {
  for(int i = 0; i < N; i++) cout << " ";
}
void recurse( const int N ) {
  if(N <= 1) {
    print_space(N);  cout << "Done!" << endl;
  } else {
    print_space(N);  cout << "Start " << N << endl;
    recurse(N-1);
    print_space(N);  cout << "End " << N << endl;
  }
}
int main() {
  recurse(6);
  return 0;
}
```

```
./recurse
      Start 6
     Start 5
    Start 4
   Start 3
  Start 2
 Done!
  End 2
   End 3
    End 4
     End 5
      End 6
```

Recursive Binary Search

```
template<typename T>
int binary_search(const List<T>& LIST, const T TARGET,
                  const int START_POS, const int END_POS) {
  ...
    return binary_search(LIST, TARGET, START_POS, MIDDLE_POS - 1);
}
```

- What is the run time?

Recursive Binary Search

```
template<typename T>
int binary_search(const List<T>& LIST, const T TARGET,
                  const int START_POS, const int END_POS) {
  ...
    return binary_search(LIST, TARGET, START_POS, MIDDLE_POS - 1);
}
```

- What is the run time? O(*log n*)

Recursive Binary Search

```
template<typename T>
int binary_search(const List<T>& LIST, const T TARGET,
                  const int START_POS, const int END_POS) {
  ...
    return binary_search(LIST, TARGET, START_POS, MIDDLE_POS - 1);
}
```

- What is the run time? O(*log n*)
- What is the extra memory usage?

Recursive Binary Search

```
template<typename T>
int binary_search(const List<T>& LIST, const T TARGET,
                  const int START_POS, const int END_POS) {
  ...
    return binary_search(LIST, TARGET, START_POS, MIDDLE_POS - 1);
}
```

- What is the run time? O(*log n*)
- What is the extra memory usage? O(*log n*)

Recursive Binary Search

```
template<typename T>
int binary_search(const List<T>& LIST, const T TARGET,
                  const int START_POS, const int END_POS) {
  ...
    return binary_search(LIST, TARGET, START_POS, MIDDLE_POS - 1);
}
```

- What is the run time? O(*log n*)
- What is the extra memory usage? O(*log n*)
- Is recursion necessary?

Recursive Binary Search

```
template<typename T>
int binary_search(const List<T>& LIST, const T TARGET,
                  const int START_POS, const int END_POS) {
  ...
    return binary_search(LIST, TARGET, START_POS, MIDDLE_POS - 1);
}
```

- What is the run time? O(*log n*)
- What is the extra memory usage? O(*log n*)
- Is recursion necessary?
  - Is there any backtracking going on?

Recursive Binary Search

```
template<typename T>
int binary_search(const List<T>& LIST, const T TARGET,
                  const int START_POS, const int END_POS) {
  ...
    return binary_search(LIST, TARGET, START_POS, MIDDLE_POS - 1);
}
```

- What is the run time? O(*log n*)
- What is the extra memory usage? O(*log n*)
- Is recursion necessary?
  - Is there any backtracking going on?
  - Any post-recursive work?

Recursive Binary Search

```
template<typename T>
int binary_search(const List<T>& LIST, const T TARGET,
                  const int START_POS, const int END_POS) {
  ...
    return binary_search(LIST, TARGET, START_POS, MIDDLE_POS - 1);
}
```

- What is the run time? O(*log n*)
- What is the extra memory usage? O(*log n*)
- Is recursion necessary? No
  - Is there any backtracking going on? No
  - Any post-recursive work? No

Iteration v Recursion

- If `Task()` definition is of form

```
A()
Task()
B()
```

  use recursion

- If `Task()` definition is of form

```
A()
Task()
```

  use iteration in place of tail recursion

Iteration v Recursion

- If `Task()` definition is of form

  ```
  A()
  Task()
  B()
  ```

  use recursion
- If `Task()` definition is of form

  ```
  A()
  Task()
  ```

  use iteration in place of tail recursion
- *(NOTE: general rule of thumb…iteration can always replace recursion…)*

Iterative Binary Search

- Run Time is still O(*log n*)
- Extra memory usage is now O(*1*)

```
template<typename T>
int binary_search(const List<T>& LIST, const T TARGET) {
  int startPos = 0, endPos = myList.size() - 1;
  int targetPos = -1;
  while( true ) {
    // perform search...
  }
  return targetPos;
}
```

Algorithm Complexities

- Scenario A
  - Unsorted list of n elements
  - Need to check if m values exist
  - Total Cost?

- Scenario B
  - Sort list of n elements
  - Need to check if m values exist
  - Total Cost?

Algorithm Complexities

- Scenario A
  - Unsorted list of n elements O(*1*)
  - Need to check if m values exist O(*mn*)
  - Total Cost? O(*mn*) → O(n^2)

- Scenario B
  - Sort list of n elements O(*n log n*)
  - Need to check if m values exist O(*m log n*)
  - Total Cost? O(*max(m,n) log n*) → O(*n log n*)

On Tap For Today

- Searching
  - Linear Search
  - Binary Search
- Practice

To Do For Next Time

- Rest of semester
  - W 11/29: 2D Lists + BFS/DFS
  - F 12/01: Stack & Queue
  - M 12/04: Trees & Graphs, Quiz 6
  - W 12/06: Exam Review
  - R 12/07: Set6, SetXP, Final Project due
  - M 12/11: Final Exam