Module (3)

Lecture Notes on Space Complexity of Operations

Understanding Space Complexity

Definition and Scope

- **Space complexity** refers to the amount of memory space an operation or algorithm consumes during execution.
- While previously we considered the entire program's space, we will now focus on the space used by individual operations.
- This estimation of space complexity should have been covered in your algorithms course.

Process Memory Allocation

- A **process** is a program in execution. The operating system allocates a chunk of memory when a process is created.
- This memory chunk is divided into four sections:
  1. **Text/Code:** Stores the executable code of the program.
  2. **Data:** Stores global variables defined in the program.
  3. **Heap:** Stores dynamically allocated memory during runtime (e.g., memory allocated using `malloc` or `calloc`).
  4. **Stack:** Stores activation records of active functions.

Activation Records

- An **activation record** for a function contains:
  - Local variables and parameters defined within the function's scope.
  - Return address to the calling function.
  - Content of CPU registers.

Example: Factorial Calculation

C

```
int n; // Global variable

int factorial(int n) {
    if (n == 0) {
```

```
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main() {
    int result = factorial(5);
    return 0;
}
```

- **Memory Allocation:**
  - **Text:** Stores the executable code of the `main` and `factorial` functions.
  - **Data:** Stores the global variable `n`.
  - **Heap:** Not used in this example (no dynamic memory allocation).
  - **Stack:**
    - Initially stores the activation record of `main`.
    - When `factorial` is called, its activation record is pushed onto the stack.
    - The stack grows and shrinks as functions are called and return.

Space Complexity Analysis

- **Space complexity** is determined by the rate of growth of the space used in the process (stack, heap, and data) with respect to the input size.
- In the factorial example:
  - The size of activation records is constant (independent of the input `n`).
  - The size of the heap and data segments is also constant.
  - Therefore, the space complexity of both the `factorial` function and the entire program is **O(1)** (constant space complexity).

Key Points

- Stack and heap can grow and shrink during runtime, while data and text segments are typically fixed.
- Space complexity is defined by how the space used in the process grows in relation to the input size.
- In the factorial example, space complexity is O(1) because the memory usage doesn't change with the input.

Lecture Notes on Space Complexity of Recursive Functions

Recursive Functions and Space Complexity

Iterative vs. Recursive Factorial

- **Iterative Approach:** The previous factorial calculation used a loop, multiplying numbers iteratively.
- **Recursive Approach:** The new factorial function calls itself recursively, reducing the problem size by 1 in each call.

C

```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

How Recursion Affects the Process Space

- **Function Calls:**
  - Each recursive call to `factorial` creates a new activation record on the stack.
  - The stack grows with each recursive call, consuming more memory.
- **Stack Growth:**
  - In the factorial example, calculating the factorial of 6 (`factorial(6)`) will push 6 activation records onto the stack.
  - The maximum number of activation records on the stack directly depends on the input value (`n`).
- **Space Complexity:**
  - The space complexity of the recursive `factorial` function is **O(n)**, as it requires storing `n` activation records on the stack.
  - The space complexity of the `main` function is also **O(n)**, as it includes the activation records of all the recursive calls.

Visualizing Space Usage

1. `main` function starts, its activation record is pushed onto the stack.
2. `factorial(6)` is called, its activation record is pushed.
3. ... `factorial(1)` is called, its activation record is pushed. (Stack at maximum size)

4. Recursive calls return, activation records are popped from the stack one by one.
5. `main` function returns, its activation record is popped. (Stack empty)

Comparing Space Complexity

- **Recursive functions** often have higher space complexity than iterative functions due to the storage of activation records on the stack.
- **Iterative factorial:** O(1) space complexity.
- **Recursive factorial:** O(n) space complexity.

Key Points

- Each recursive call creates a new activation record, increasing space usage on the stack.
- Space complexity of a recursive function depends on the maximum depth of the recursion (related to the input size).
- Recursive functions can lead to higher space complexity compared to iterative solutions.

Lecture Notes on Space Complexity of Recursive Functions (Continued)

Recursive Factorial Example

Code

C

```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

Recursive Function Calls

1. `main` calls `factorial(6)`.
2. `factorial(6)` calls `factorial(5)`.
3. ... and so on until `factorial(1)` calls `factorial(0)`.

Activation Records on the Stack

- Each function call pushes an activation record onto the stack, containing:
  - The value of `n` for that call.
  - The return address (where to go back after the recursive call).
  - Any local variables (none in this example).
- The stack grows with each recursive call, reaching its maximum size when `factorial(0)` is reached.

Return Values and Stack Unwinding

- When `factorial(0)` returns 1, its activation record is popped off the stack.
- The previous call (`factorial(1)`) receives the return value, calculates `1 * 1`, and returns 1. Its activation record is also popped.
- This process continues, with each call receiving the return value from the previous call, calculating its own result, and then returning, while its activation record is popped.

Space Complexity

- **Maximum Stack Size:** The maximum number of activation records on the stack is equal to the input value `n`.
- **Space Complexity:** Therefore, the space complexity of the recursive factorial function is O(n) (linear space complexity).

Calculating Space for `main` Function

- The `main` function has its own activation record.
- It also calls `factorial(n)`, which leads to a maximum of `n` additional activation records on the stack.
- Total space for `main` function: 1 (for `main`) + n (for `factorial`) = n + 1
- Space complexity of `main`: O(n)

Key Takeaway

The space complexity of recursive functions is often higher than that of iterative functions due to the storage of activation records on the stack during recursive calls.

Let me know if you'd like me to continue summarizing more of the lecture!

Lecture Notes on Finding Minimum in a Collection Using Arrays

Problem Statement

Given a collection of integer numbers, find the minimum number.

Choosing a Data Structure

- **Array or Linked List:** Both are viable options for this problem.
- **Key Considerations:**
  - **Visiting all elements:** Yes, to find the minimum, we need to examine every element.
  - **Time complexity of traversal:** Same for both arrays and linked lists (O(n)).
  - **Access method:** Sequential access is sufficient for both data structures.

Array-Based Solution

Algorithm

1. **Store elements:** Define an array and store the elements of the collection in it.
2. **Scan and compare:**
   - Initialize a variable `min` with the first element of the array.
   - Iterate through the remaining elements, starting from the second element.
   - For each element, compare it with the current value of `min`.
   - If the element is smaller than `min`, update `min` with the new minimum value.
3. **Return the minimum:** After traversing all elements, `min` will hold the minimum value.

Code

C

```
int findMin(int arr[], int size) {
    int min = arr[0]; // Assume the first element is the minimum
    for (int i = 1; i < size; i++) {
        if (arr[i] < min) {
            min = arr[i]; // Update min if a smaller element is found
        }
    }
    return min;
}

int main() {
    int arr[] = {5, 8, 7, 3, 9};
    int size = sizeof(arr) / sizeof(arr[0]);
```

```
    int min = findMin(arr, size);
    // ... (rest of the main function)
}
```

Space and Time Complexity

- **Time Complexity:** O(n), as we need to iterate through all elements once.
- **Space Complexity:**
  - `findMin` function: O(1), as it only uses a constant amount of space (for variables `min` and `i`).
  - `main` function: O(n), due to the space used to store the array.

Key Points

- The array-based solution provides a straightforward and efficient way to find the minimum value in a collection.
- The space complexity of the `findMin` function is constant, but the overall space complexity is O(n) due to the array storage.

Let me know if you'd like a similar analysis for a linked list-based solution!

Lecture Notes on Finding Minimum in a Collection

Problem Definition

Given a collection of integer numbers, find the minimum value.

Choosing the Data Structure: Array vs. Linked List

Considerations:

- **Need to visit all elements?** Yes
- **Time complexity of traversal?** Same for arrays and linked lists (O(n))
- **Access method?** Sequential access

Conclusion: Both arrays and linked lists are suitable for this problem, with equivalent efficiency.

Array-Based Solution

Algorithm

1. **Store elements:** Create an array and populate it with the collection's elements.

2. **Scan and compare:**
    - Initialize a variable `min` to the first element of the array.
    - Iterate through the remaining elements, comparing each to `min`.
    - If a smaller element is found, update `min`.
3. **Return minimum:** The final value of `min` is the minimum value in the collection.

Code

C

```
int findMin(int arr[], int size) {
    int min = arr[0];
    for (int i = 1; i < size; i++) {
        if (arr[i] < min) {
            min = arr[i];
        }
    }
    return min;
}

int main() {
    int arr[] = {5, 8, 7, 3, 9};
    int size = sizeof(arr) / sizeof(arr[0]);
    int min = findMin(arr, size);
    // ... rest of main function
}
```

Space and Time Complexity Analysis

- **Time Complexity (findMin):** O(n), as it needs to scan all elements.
- **Space Complexity (findMin):** O(1), as it only uses a constant amount of space for variables `min` and `i`.
- **Space Complexity (main):** O(n) due to the space used to store the array.

Linked List-Based Solution (Optional)

Algorithm

1. **Store elements:** Create a linked list and populate it with the collection's elements.
2. **Traverse and compare:**

- Initialize a variable `min` to the data of the first node.
- Traverse the linked list, comparing the data of each node to `min`.
- If a smaller value is found, update `min`.

3. **Return minimum:** The final value of `min` is the minimum value in the collection.

Code (Conceptual)

C

```
int findMin(Node* head) {
    int min = head->data;
    Node* current = head->next;
    while (current != NULL) {
        if (current->data < min) {
            min = current->data;
        }
        current = current->next;
    }
    return min;
}
```

Space and Time Complexity Analysis

- **Time Complexity:** O(n), as it needs to traverse all nodes.
- **Space Complexity (findMin):** O(1), as it only uses a constant amount of space (for variables `min` and `current`).
- **Space Complexity (Overall):** O(n) due to the space used to store the linked list.

Key Takeaways

- Both array and linked list solutions have the same time complexity O(n).
- The space complexity of the functions themselves is constant (O(1)), but the overall space complexity is O(n) due to the data structure storage.
- The choice between array and linked list might depend on other factors, such as the need for dynamic resizing.

Lecture Notes on Binary Search and Data Structure Selection

Problem Statement

Given a collection of integer numbers, search for a specific element within the collection.

Linear Search vs. Binary Search

Linear Search

- **Approach:** Sequentially check each element until a match is found or the entire collection is exhausted.
- **Time Complexity:**
  - Best Case: O(1) (element found at the beginning)
  - Worst Case: O(n) (element not found or at the end)
  - Average Case: O(n)
- **Space Complexity:** O(1) for iterative implementation, O(n) for recursive implementation
- **Disadvantage:** Inefficient for large collections, as it may require visiting all elements.

Binary Search

- **Approach:**
  - Works on sorted collections (ascending or descending order).
  - Starts at the middle element.
  - If the middle element matches the target, the search is successful.
  - Otherwise, the search continues in either the left or right half of the collection, depending on whether the target is smaller or larger than the middle element.
  - This process is repeated until the element is found or the search space is exhausted.
- **Time Complexity:**
  - Best Case: O(1) (element found at the middle)
  - Worst Case: O(log n)
  - Average Case: O(log n)
- **Space Complexity:** O(1) for iterative implementation, O(log n) for recursive implementation
- **Advantage:** Much more efficient than linear search for large, sorted collections.

Example: Binary Search

1. **Collection:** {2, 5, 8, 12, 16, 23, 38, 56, 72, 91}
2. **Target:** 23

3. **Iterations:**
    - Compare 23 to middle element 16. Since 23 > 16, search the right half.
    - Compare 23 to middle element 56. Since 23 < 56, search the left half.
    - Compare 23 to middle element 23. Match found!

Divide and Conquer

- Binary search is an example of a **divide and conquer** algorithm.
- **Divide:** Split the problem into smaller subproblems.
- **Conquer:** Solve the smaller subproblems (recursively).
- **Combine:** Combine the solutions of the subproblems to solve the original problem.

Time and Space Complexity of Binary Search

- **Iterative:**
    - Time: O(log n)
    - Space: O(1)
- **Recursive:**
    - Time: O(log n)
    - Space: O(log n)

Array vs. Linked List for Binary Search

- **Array:** More suitable for binary search due to its direct/random access capability.
- **Linked List:** Not ideal for binary search because accessing the middle element requires traversing the list.

Conclusion

The choice of data structure can significantly impact the efficiency of algorithms. For searching in sorted collections, binary search on arrays is more efficient than linear search or binary search on linked lists.