Recursive Depth of Quicksort and Nonrecursive Quicksort

Recursive Depth of Quicksort

The recursive depth of quicksort refers to the maximum depth of the recursion stack, which determines the system's memory usage to keep track of recursive calls. The depth depends on how balanced the partitions are at each step.

1. Best Case: $O(\log n)$

In the best case, each partition splits the array into two nearly equal parts, reducing the size of the problem by half at each step. The number of recursive calls is logarithmic, leading to a recursive depth of $O(\log n)$. This happens when the pivot is always chosen close to the median.

2. Worst Case: $O(n)$

In the worst case, the pivot is consistently chosen as the smallest or largest element, leading to highly unbalanced partitions (one subarray has size $n-1$ and the other has size 0). This results in $n-1$ recursive calls, and the recursive depth is $O(n)$.

3. Average Case: $O(\log n)$

On average, quicksort splits the array into reasonably balanced subarrays, leading to a recursive depth of $O(\log n)$.

Exercise 2.3.20: Guaranteeing Logarithmic Recursive Depth

Exercise 2.3.20 from Robert Sedgewick's *Algorithms* book suggests a way to guarantee that the recursive depth of quicksort is logarithmic even in the worst case. The key idea is to push the larger of the two subarrays onto the stack first, ensuring that the smaller subarray is processed next. This guarantees that the stack size remains $O(\log n)$, even in the worst case.

Nonrecursive Quicksort

We can implement a nonrecursive version of quicksort by using an explicit stack to simulate the system's recursive call stack. The main loop pops subarrays from the stack to partition, and then pushes the resulting subarrays back onto the stack for further processing. By always pushing the larger subarray first, we ensure that the stack has at most $O(\log n)$ entries.

Pseudocode for Nonrecursive Quicksort

```
function nonrecursive_quicksort(A, low, high):
    stack = new empty stack
    push the initial subarray (low, high) onto the stack

    while stack is not empty:
        (low, high) = pop from stack

        if low < high:
            pivot_index = partition(A, low, high)

            left_size = pivot_index - low
            right_size = high - pivot_index

            if right_size > left_size:
                push (pivot_index + 1, high) onto the stack
                push (low, pivot_index - 1) onto the stack
            else:
                push (low, pivot_index - 1) onto the stack
                push (pivot_index + 1, high) onto the stack
```

Explanation

- **Stack Setup:** An empty stack is initialized to simulate the recursive process. Instead of making recursive calls, we push the indices of subarrays onto the stack.
- **Partitioning:** Each iteration pops a subarray from the stack, partitions it using the same method as recursive quicksort, and pushes the resulting subarrays back onto the stack.
- **Pushing Subarrays:** After partitioning, the larger of the two subarrays is pushed onto the stack first, followed by the smaller subarray. This ensures that the stack depth remains small, at most $O(\log n)$.
- **Termination:** The loop terminates when the stack is empty, meaning all subarrays have been processed.

Time and Space Complexity

- **Time Complexity:** The time complexity of nonrecursive quicksort is the same as standard quicksort: $O(n \log n)$ in the average case, and $O(n^2)$ in the worst case.
- **Space Complexity:** The space complexity is reduced to $O(\log n)$ since the stack never holds more than $O(\log n)$ entries, even in the worst case.

Conclusion

1. **Recursive Depth:**

- Best case: $O(\log n)$
- Worst case: $O(n)$
- Average case: $O(\log n)$

2. **Exercise 2.3.20:** By pushing the larger subarray onto the stack first, we guarantee that the recursive depth remains logarithmic.

3. **Nonrecursive Quicksort:** A nonrecursive version of quicksort can be implemented using an explicit stack. By pushing the larger subarray first, we ensure that the stack size remains $O(\log n)$, improving space efficiency.