Noah Caulfield

Dr. Essa Imhmed

CS 451

4/3/2024

Homework 3: Divide-and-conquer algorithms.

**1. Here is an array which has just been partitioned by the first step of**

**Quicksort:**

**3, 0, 2, 4, 5, 8, 7, 6, 9**

**Which of these elements could have been the pivot? (if there are more**

**than one possibility, list them all.)**

**4** works as a pivot because every number before it in the array (3, 0, 2) is smaller, and every number after it (5, 8, 7, 6, 9) is larger. So, 4 divides the array into a smaller side and a larger side.

**5** can be a pivot. The numbers before it (3, 0, 2, 4) are all smaller, and the numbers after it (8, 7, 6, 9) are all larger. Five also neatly splits the array into groups of smaller and larger numbers.

**9** is at the end of the array, but it still counts as a pivot. All the numbers before it (3, 0, 2, 4, 5, 8, 7, 6) are smaller, and there are no numbers after it, making 9 an effective point for dividing the array, even if creating only one side.

**2. What is the running time of Quicksort when all elements of array A have**

**the same value? Justify your answer.**

When Quicksort sorts an array where every item is the same, it ends up working slower, taking time. This happens because, instead of breaking the list into smaller parts, Quicksort tries to sort the array one item at a time, putting all other items on one side. So, instead of quickly dividing and conquering, it slowly goes through the whole array, making it much less efficient for this specific situation.

**3. Given a sorted array of distinct integers A[1, . . . , n], you want to find out**

**whether there is an index i for which A[i] = i. Give a divide-and-conquer**

**algorithm that runs in time O(log n).**

**# copy/paste version of code; image below**

**def findFixedIndexIterative(A):**

**low, high = 0, len(A) - 1**

**while low <= high:**

**mid = (low + high) // 2**

**if A[mid] == mid:**

**return mid # Fixed point found**

**elif A[mid] > mid:**

**high = mid - 1 # Search left half**

**else:**

**low = mid + 1 # Search right half**

**return None # No fixed point found**

**# Example usage**

**A = [-10, -1, 0, 3, 10, 11, 12, 13] # Example array, assuming A[i] starts from A[0]**

**result = findFixedIndexIterative(A)**

**if result is not None:**

**print(f"A fixed point is at index {result}, where A[{result}] = {result}.")**

**else:**

**print("There is no index i such that A[i] = i.")**

**A screenshot of a computer program

Description automatically generated**

**4. Suppose we are comparing implementations of insertion sort and merge**

**sort on the same machine. For inputs of size n, insertion sort runs in 8n2**

**steps, while merge sort runs in 64n lg n steps. For which values of n does**

**insertion sort beat merge sort?**

(Next page)

Insertion Sort Complexity 8n^2

Merge sort complexity 64n lg n (log2 specifically)

-Insertion Sort runs faster on smaller datasets because it has less overhead.

-Merge Sort is more efficient for larger datasets due to its (0(n log n) time complexity.

For small values of *n:* The Insertion Sort's steps (based on *8n^2* are fewer than merge sort's step because the square of a small number *n^2* grows more slowly than *n* lg n when *n* is large.

Transforming *n*: When we express *n* as *2^k* (Assuming *k* is some power of *2*), both sorting algorithms' steps change then Insertion Sort's steps become based on *4^k*, and Merge Sort's steps involve *k* and (*2^k*). The precise point of crossover depends on the figures input for computation.

Finding the Crossover: For very small arrays used, the quicker operation of Insertion Sort (which doesn't have to do as much splitting and merging when compared to merge sort) makes it faster. But, when the array size grows when *k* increases, merge sort becomes faster because its more efficient *0(n log n)* time complexity overtakes the *0(n^2)* complexity of insertion sort.

*SOLUTION:*

Thus, in theory, insertion sort beats merge sort for n = 1 to 7, and merge sort becomes more efficient for n ≥ 8.