Homework Assignment Week 2: Divide and Conquer Algorithm, and Recurrence Relations

Hyuntaek Oh

ohhyun@oregonstate.edu

CS 514_400 Algorithm

October 14, 2024

1. In an array of distinct elements, implement an algorithm (using Python) $rank\_smallest(A[1..n], b, k)$ using the divide and conquer technique, where: b is a positive integer. k is a non-negative integer such that $kb \leq n$. The algorithm should print the elements of A whose ranks are $b, 2b, 3b, \ldots, kb$ in sorted order. If $k = 0$, the algorithm should print nothing. You must include your own test cases (you are allowed to use Numpy and Pandas libraries).

```python
def merge(left, right):
    ans = []
    i = j = 0

    # Merging elements of left and right array
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            ans.append(left[i])
            i += 1
        else:
            ans.append(right[j])
            j += 1

    # Merging remaining elements
    ans += left[i:]
    ans += right[j:]

    return ans
```

```python
def mergeSort(A):
    # Base case: only one element in the array
    if len(A) <= 1:
        return A

    # Mid-point for dividing
    mid = len(A) // 2

    # Merge left-sorted array and right-sorted array
    return merge(mergeSort(A[:mid]), mergeSort(A[mid:]))
```

Figure 1. Merge and merge sort function

```python
def rank_smallest(A, b, k):

    ans = []

    # Print nothing
    if k == 0:
        return []

    # Sorting an array with merge sort
    sortedArray = mergeSort(A)

    # Print elements if the conditions are satisfied
    if k * b <= len(A):
        for i in range(0, k):
            ans.append(sortedArray[(b-1) + i*b])
    else:
        print(f"Warning: Should be k * b <= {len(A)}, "
              f"previous k * b was {k*b}.")
        return []

    return ans
```

```python
if __name__ == "__main__":
    Arr = [7, 1, 4, 5, 2, 3, 6, 8, 9]
    k = 3
    b = 3
    print(rank_smallest(Arr, b, k))
```

```
C:\Python312\python.exe C:\2024_Fall\
[3, 6, 9]

Process finished with exit code 0
```

Figure 2. Rank_smallest function, test cases, and result

2. Analyze the time complexity of the algorithm. Next, plot the running times of the algorithm against the input size (up to 10^7 numbers) for both randomly sorted input arrays and already sorted input arrays. Identify which functions best represent the running times. Finally, compare your findings with the expectations based on theoretical analysis.

(a) Analyze the time complexity of the algorithm

In the rank_smallest function, there are two parts to analyze total time complexity: merge sorting and print ranking. To sort a given unsorted list, a merge sorting algorithm is used.

Merge sorting function has two steps that dividing a given list and adding the sorted elements from left and right halves list to the result list called ans, which is a conquer and combine part. Dividing a given list into left and right halves takes $O(1)$ because calculating the middle index takes constant time.

Conquer part is recursively solving two sub-problems, each of size $\frac{n}{2}$, which takes $T\left(\frac{n}{2}\right)$ time complexity. After conquering sub-problems, combine part is gathering each sub-problem, which takes $O(n)$ time complexity.

Lastly, in the rank_smallest function, one for loop calculates how many numbers and intervals print. The term k means the number of printing satisfied numbers, and b is the distance between first and next printing numbers. In the loop, the time complexity depends on the number k and b, meaning that the worst case can be $O(k)$ since the product of k and b is equal to or less than the length of the given list, n, which means that k is generally much smaller than n.

Thus, the total time complexity is:

$$T(n) = O(1) + 2T\left(\frac{n}{2}\right) + O(n) + O(k)$$

$$= 2T\left(\frac{n}{2}\right) + O(n)$$

$$= 2T\left(\frac{n}{2}\right) + cn$$

$$= O(nlogn)$$

The last statement in the total time complexity is analyzed by using the second case of Master Theorem from exploration on Canvas.

(b) Plot the running times of the algorithm against the input size (up to 10^7 numbers) for both randomly sorted input arrays and already sorted input arrays. Identify which functions best represent the running times.

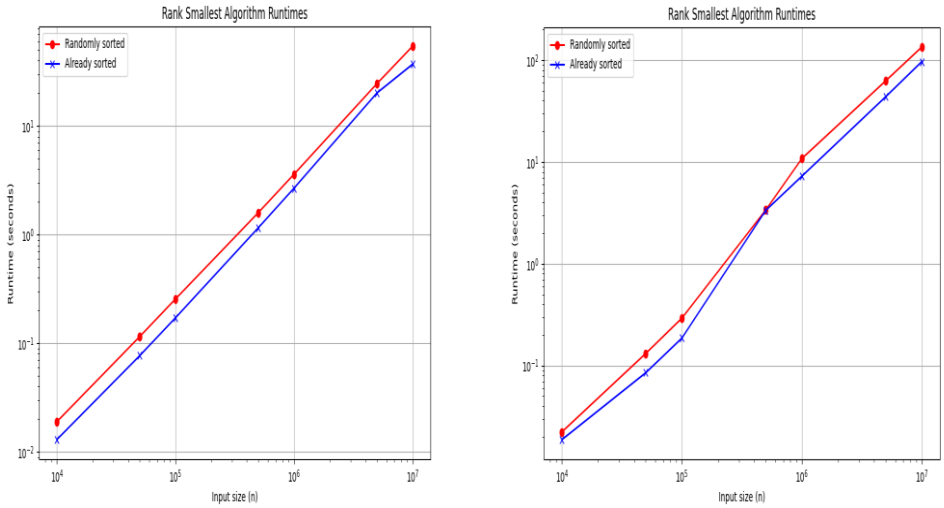


Figure 3. Trials for runtimes of randomly sorted and already sorted array

As can be seen in Figure 3., the sorted input array function shows the best running time. In Figure 3, the already sorted graph shows better performance most of the running time, which is faster runtime than randomly sorted.

(c) Compare your findings with the expectations based on theoretical analysis.

In the theoretical analysis, the algorithm involves sorting the input array with merge sort, which has $O(nlogn)$ time complexity. After that, it looks for the desired elements, which takes $O(k)$ time, where k is the number of elements to print. Since the value of k is smaller than n, the total time complexity is dominated by sorting, making the algorithm's time complexity $O(nlogn)$.

However, unlike my expectation, the plot above shows similar runtime growth for both randomly sorted and already sorted input arrays. The already sorted arrays consistently take slightly less time than the randomly sorted arrays, showing logarithmical increase. At least, I believe that the time complexity of already sorted arrays would show much better performance than randomly sorted arrays since there is no need to sort the elements in the array, only checking.

3. Specify loop invariants to justify the correctness of any loops used in the algorithm. Prove that the algorithm is correct using your previously defined invariants.

In the rank smallest algorithm, we need to verify loop invariants of merge sorting and smallest ranking, which affect primarily the total time complexity. The loop invariant of merge sorting is a result array, called ans that is combined with left and right halves smaller arrays divided by the middle point. There are three steps to identify loop invariants. In the initialization, before the loop starts, the result array is empty. In the maintenance, during each iteration, one element is added to the result

array from either left or right halves array, which are both smaller. Since both subarrays are individually sorted, the element added is generally the smallest value. So, the new result array contains sorted elements after insertion of the new element, maintaining the invariant. In the termination, the loop ends when the two pointers i, j are the end of the left and right halves array respectively. Remaining elements in both sub arrays are appended to the result array.

Moreover, in the smallest ranking part, the loop variant is that the array ans contains the first i elements selected according to the formula $(b-1)+i*b$, and these elements are in non-decreasing order. One of the properties of the invariant in the initialization is that ans is an empty array before starting iteration and a given array is sorted. In the maintenance, during each iteration, the algorithm appends the element from the sorted array to ans. Since the elements in sorted array are already sorted, the element at index $(b-1)+i*b$ is larger than the previously selected element at index $(b-1)+(i-1)*b$. Therefore, after each iteration, the array ans contains the sorted elements. In the termination, the loop ends after k iterations. At this point, the invariant ensures that ans contains the first k smallest elements, which are ascending order.

4. Write the recurrence relation of the below pseudocode that calculates x^n, and solve the recurrence relation using three methods that we have seen in the exploration.

```
power2(x,n):
    if n==0:
        return 1
    if n==1:
        return x
    if (n%2)==0:
        return power2(x, n//2) * power2(x,n//2)
    else:
        return power2(x, n//2) * power2(x,n//2) * x
```

(a) Recurrence relation

The function $power2(x,n)$ can be expressed with $T(n)$ with certain conditions. It returns a constant value if $n=0$ or $n=1$, so time complexity is $T(n)=O(1)$. On the other hand, the function makes two recursive calls to $T\left(\frac{n}{2}\right)$ and some multiplication, which can be a constant time, if $n>1$. So, we can get:

$$T(n)=2T\left(\frac{n}{2}\right)+O(1)$$

(b) Substitution Method

From the recurrence above, we can expand it by substitution:

$$T(n)=2T\left(\frac{n}{2}\right)+O(1)=2\left(2T\left(\frac{n}{4}\right)+O(1)\right)+O(1)=2^2\left(\frac{n}{4}\right)+3O(1)$$

$$= 2^2\left(2T\left(\frac{n}{8}\right) + O(1)\right) + 3O(1) = 2^3T\left(\frac{n}{8}\right) + 7O(1)$$

$$T(n) = 2^kT\left(\frac{n}{2^k}\right) + (1 + 2^{k-1})O(1)$$

Let's assume $n = 1$. Then, $T(n) = O(1)$ since the base case takes constant time. We can change the recurrence relation for the base case like:

$$\frac{n}{2^k} = 1 \Rightarrow 2^k = n \Rightarrow k = \log_2 n$$

We can use it like:

$$T(n) = 2^{\log_2 n}T\left(\frac{n}{2^{\log_2 n}}\right) + \left(1 + 2^{\log_2 n - 1}\right)O(1)$$

This can be simplified as follows:

$$T(n) = nT(1) + \left(1 + \frac{n}{2}\right)O(1) = nO(1) + \left(1 + \frac{n}{2}\right)O(1)$$

$$T(n) = O(n) + O(n) = O(n)$$

Thus,

$$T(n) = O(n)$$

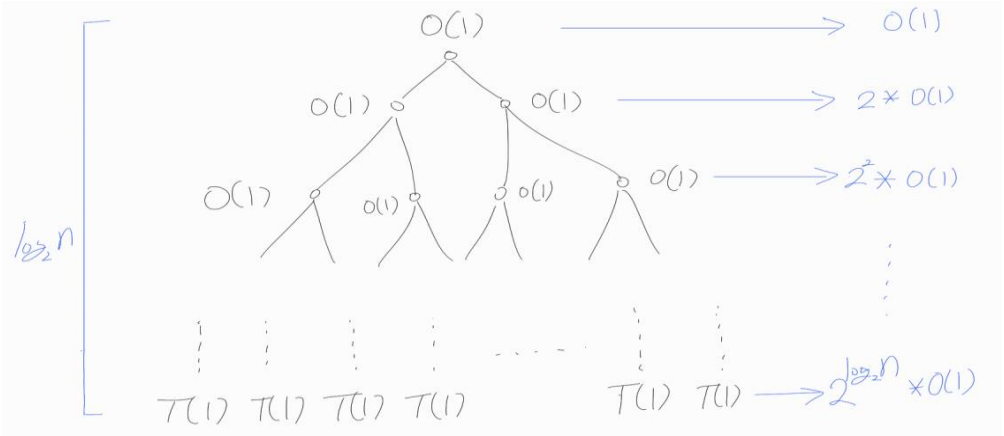
(c) Recursion Trees

We can write the recurrence:

$$T(n) = \begin{cases} O(1) & if\ n = 0\ or\ 1 \\ 2T\left(\frac{n}{2}\right) + O(1) & if\ n > 1 \end{cases}$$

We use $T(n) = 2T\left(\frac{n}{2}\right) + O(1)$ $if\ n > 1$ to develop the recursion tree.

The recursion tree is:



The tree above is based on the second term in $T(n) = 2T\left(\frac{n}{2}\right) + O(1)$, which is $O(1)$. From the tree, adding up the levels, we get:

$$T(n) = O\left(1 + 2 + 4 + \cdots + 2^{\log_2 n}\right)$$

$$= \sum_{i=0}^{\log_2 n - 1} 2^i + O\left(2^{\log_2 n}\right)$$

The mathematical expression above can be simplified by applying equation as follows:

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$$

$$\sum_{i=0}^{\infty} 2^i + O\left(2^{\log_2 n}\right) = \frac{1}{1-2} + O(n)$$

$$= -1 + O(n)$$

$$\in O(n)$$

Thus, $T(n) = O(n)$

(d) Master Theorem and Method

We use a divide and conquer recurrence of the form:

$$T(n) = aT\left(\frac{n}{b}\right) + O\left(n^{\log_b a - \varepsilon}\right) \rightarrow T(n) = 2T\left(\frac{n}{2}\right) + O(1)$$

$$, where\ a > 1, b > 1.$$

The term $O(1)$ is $f(n)$. We compute $\log_b a = \log_2 2 = 1$ and compare 1 to $n^{\log_b a} = n$. For holding Case 1 in Master theorem and Method to find the value of ε, the equation should be like:

$$1 = n^0 = n^{\log_2 2 - \varepsilon}$$

$$0 = 1 - \varepsilon$$

$$\varepsilon = 1$$

When the value of ε is equal to 1, the condition of Case 1 holds since $\varepsilon > 0$.

Thus, solution is:

$$T(n) = \theta(n)$$