Problem session 9

Dibran Dokter 1047390

November 22, 2019

9

9.1

First we split up the input into evenly sized blocks that are smaller than 100MB. After this we sort them using heap-sort while only having one into memory at a time by loading and saving to disk between sorts. Since heap-sort is an in-place algorithm we should not exceed the 100MB limit. After this we take the first 10MB from every list, up to 100MB. Since these are already sorted the lowest value should be at the front of one of these lists. We use insertion-sort to take the lowest value from the front of the lists and place them into the resulting sorted list. When the resulting list is 100MB we write it to disk. If one of the 10MB lists is empty we take the next 10MB from the same list. We do this until all the input is empty and we have sorted the list.

The space complexity of this algorithm is at most 100MB so it fits into main memory. The time complexity for this algorithm is $\mathcal{O}(1)$ for splitting the list, $\mathcal{O}(n \log n)$ for heap-sorting the list and $\mathcal{O}(n^2)$ for insertion-sorting the list.

9.2

The worst time complexity is $\mathcal{O}((n \cdot k) \log k)$ since we split the list k into smaller parts until we have a single string. This gives the height of the tree $\log k$. For every level in the tree we need to do lexicographical ordering, which takes $\mathcal{O}(k \cdot n)$. Thus giving us $\mathcal{O}((n \cdot k) \log k)$

9.3

512 is the closest to 6 minutes.
$64 \log 64 = \ 115, 12 \cdot 115 = 1400, 512 \log 512 = \ 1387$.
Which is closest to 1400.

9.4

We split the list into two parts, then we look at the first value in the list and compare it with the last index value. If the value is greater than the index then it is impossible to find an index where the value is equal to the index in that part of the list.
If the value is equal or smaller to the index it is still possible that the index is equal to its value and we should continue dividing.
We either find that it is impossible after we have split until we have leafs with a single value or we find out that it is impossible.
In this way we create a tree like structure where we do a constant amount of work for every layer, giving us the time complexity $\mathcal{O}(\log n)$.

9.5

We split the list of values into halves and then compute the maximum of the following values:
- Maximum sum in the left half
- Maximum sum in the right half
- Maximum sum when we look at the left side and the right side, so we work outwards from the middle.

In the end we take the maximum value of these when we merge all the values and that gives the highest value. The path that gives us this value is the best path.

The complexity of this algorithm is $\mathcal{O}(n \log n)$ since we split the input until we get to single values and then find the sum, which is $\mathcal{O}(n)$ giving us the complexity $\mathcal{O}(n \cdot \log n)$.

9.6

1:

Mark : Since splitting (n-1) results in n layers, it comes to $\mathcal{O}(1 \cdot n)$ which is $\mathcal{O}(n)$.
John : Since splitting into n/3 results in n/3 layers, and we do something for n/3 layers, namely the n^2 operation $\mathcal{O}(n/3 \cdot n^2)$.
David : Since splitting into n/2 results in n/2 layers, and we do something for every layer, namely something with time complexity $\mathcal{O}(n)$, this results in $\mathcal{O}(n/2 \cdot n)$.

2: Marks solution is best since it gives the time complexity $\mathcal{O}(n)$ which is always lower than $\mathcal{O}(c \cdot n)$.

9.7

The algorithm is as follows:
We sort the list using merge-sort, giving the time complexity $\mathcal{O}(n \log n)$.
Then we jump to the middle of the list and calculate the sum of the value to the left and to the right.
If this value is higher than the value we are looking for, we should move to the left and try again. If it is lower than we want, we should move to the right.
When we move to the left or the right we move by half of the list that is left in that direction.
In this way we can hop through the list in $\mathcal{O}(\log n)$. And we do some action for every jump.
This gives the complexity $\Theta(n \log n)$, namely $\Theta(2n \log n)$.