CSE 102 Homework 2

Dongjing Wang

Student ID: 1823945

January 19, 2023

Ex.1 (More Recursions)

1.1 $T(n) = 2T(\sqrt{n}) + \log(n)$

Answer: Since the form of the original formula does not belong to the form of Master Theorem (that is, it cannot be directly applied), we can try to bring $n = 2^m$ into it. After bringing in $T(n) = 2T(\sqrt{n}) + \log n$ will become $S(m) = 2T\left(\frac{m}{2}\right) + m$. At this point we can apply Master Theorem and know that $a = 2$, $b = 2$, $d = 1$. At the same time, since $a = b^d$, the formula of the final result is $O\left(n^d \log(n)\right) = O(m \log(m)) = \log(n) \log(\log(n))$.

1.2 $T(n) = 2T(\sqrt{n}) + n \log n$

Answer: Assuming $O(n)$ can be established. First confirm whether the base case is true: $T(1) \leq c \times 1$. It can be found that $T(1) = \log 1 = 0$, and it can be deduced that $0 \leq c \times 1$ holds. Next, you need to find the inductive step. Assume that $T(n) \leq c \times n$ holds for some c and some n. So it is necessary to show whether $T\left(n^2\right) = cn^2$ is true. $T\left(n^2\right) = 2T\left(\sqrt{n}^2\right) + \log n^2 = 2T(n) + \log n^2 \leq 2cn + \log n^2$. So you can finally get a formula that is $2cn + \log n^2 \leq cn^2$. This can be true when assuming $c = 1$. So by pushing it, $T(n) \leq O(n)$ is a valid solution for $T(n) = 2T(\sqrt{n} + \log n)$.

1.3 $T(n) = 2T(\sqrt{n}) + 1$

Answer: Since this formula is not in the same format as Master Theorem, it may be a good choice to try substitution. First, assume that $O(n)$ is a solution to $T(n) = 2T(\sqrt{n}) + 1$. Then we need to find the base case of this hypothesis, that is, to confirm that $T(1) \leq c \times 1$ for some constant. $T(1) = \log 1 = 0$, so $0 \leq c \times 1$ is established, that is to say, this base case can be established. The next thing is inductive step. Suppose $T(n) \leq c \times n$ holds for some constant c and some values of n. We need to prove that $T\left(n^2\right) \leq cn^2$.

$T\left(n^2\right) = 2T\left(\sqrt{n}^2\right) + 1 = 2T\left(n\right) + 1 \leq 2cn + 1$. So we need to prove that $2cn + 1 \leq cn^2$. This holds true when $c = 1$. So $T\left(n\right) \leq O\left(n\right)$ can be used as a valid solution of $T\left(n\right) = 2T\left(\sqrt{n}\right) + 1$.

1.4 $T\left(n\right) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{3}\right) + T\left(\frac{n}{10}\right) + T\left(\frac{n}{20}\right) + n$

Answer: This formula does not seem to be directly solved by conventional Master Theorem, so using the substitution method is still a very good method. Assuming $O\left(20n\right)$ can be established. First of all, try to find the base case, that is, $T\left(1\right) \leq c \times 1$. $T\left(1\right) = 1$, so $1 \leq c \times 1$ holds. Suppose $T\left(n\right) \leq 20c$, and we need to prove that $T\left(\frac{n}{2}\right) + T\left(\frac{n}{3}\right) + T\left(\frac{n}{10}\right) + T\left(\frac{n}{20}\right) + n \leq 20c\left(n/2\right) + 20c\left(\frac{n}{3}\right) + 20c\left(\frac{n}{10}\right) + 20c\left(\frac{n}{20}\right) + 20n$. We can directly determine the size of the two equations through the comparison between the various items. It is easy to find that the sum on the right is much larger by comparison. So it can be concluded that $T\left(n\right) \leq O\left(20n\right)$ is a solution for $T\left(n\right) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{3}\right) + T\left(\frac{n}{10}\right) + T\left(\frac{n}{20}\right) + n$.

Ex.2 (Binary Search)

Answer: First, align the pointer to the first element. When the first element is smaller than the target, align it with the second element. If it's still smaller, it's aligned to the fourth element, then the eighth, and so on. When the pointer points to a number greater than the target, a binary search is performed between this pointer and the number pointed by the previous pointer (if the number pointed to by the first one is greater than that, it directly indicates that it does not exist). At the same time, if the number pointed to reaches ∞, it means that the number does not exist in the array.

Ex.3 (Index-Value Match)

Answer: The core of the algorithm still uses binary search. The double pointer method may be a more effective way to solve this problem. First set the left pointer to the minimum value in the array and the right pointer to the maximum value in the array. Three situations may occur next:

1. If A[middle] == middle, it means that it is found.
2. If A[middle] < middle, set the lower bound to middle + 1
3. If A[middle] > middle, set the upper bound to middle - 1

Similarly, if the left pointer is no longer on the left, the right pointer is no longer on the right, or if the two pointers coincide, it means that no eligible element can be found in the array. number does not exist in the array.

Ex.4 (k-way merge operation)

(i) Answer: The time complexity of this algorithm is $O(kn^2)$. Here is the derivation: The cost of merging each array is $O(n)$. Since there are k arrays in total, the total cost is $O(kn)$. At the same time, since this method is to merge one by one, it needs to be repeated $k-1$ times. So the final time complexity of the whole scheme is $O(kn^2)$.

(ii) Answer: We can use a divide and conquer method to merge all arrays summing up to k two by two. Time complexity analysis: Like the above method, the time complexity required for each step is $O(n)$. Since we have a total of k arrays, the total time cost is $O(kn)$. However, since this is a two-by-two merge, we need to repeat fewer steps than the previous method, which is $\log k$. In this way, we can get the final result: the time complexity of this method is $O(nk \log k)$.

Ex.5

Answer: Due to the limitation of time complexity, ideally we can use binary search in the two arrays and use recursion to reduce some of the extra complexity caused by touching the two arrays at the same time. Suppose two arrays are `a`, `b`, and the lengths are m, n respectively. Next we can try to deal with some base cases. If there is an array with a length of 0, then directly find the number with sequence $k-1$ from another array. If $k = 0$, directly return the minimum value of the first number in the two arrays. At this time, set i and j to the minimum value of $\frac{k}{2}$ and the length of the array (corresponding to `a` and `b` respectively). If $\texttt{a}[i-1]$ is less than $\texttt{b}[j-1]$, remove the a before sequence i elements in and continue comparing. On the contrary, if $\texttt{a}[i-1]$ is greater than or equal to $\texttt{b}[j-1]$, remove the elements in b before the sequence j and continue the comparison. Don't worry about this method not returning a value. Since this is a recursive function, continuing the comparison means re-calling the function itself with new parameters. In other words, it will eventually be a kind of base case. And the following is the program composition itself:

```
#include <iostream>
#include <vector>

using namespace std;

int findKthSmallest(const vector<int>& A, const vector<int>& B, int k) {
    int m = A.size();
    int n = B.size();
    if (m > n) return findKthSmallest(B, A, k);
    if (m == 0) return B[k - 1];
    if (k == 1) return min(A[0], B[0]);
    int i = min(m, k / 2);
```

```
    int j = min(n, k / 2);
    if (A[i − 1] < B[j − 1]) {
        return findKthSmallest(vector<int>(A.begin() + i, A.end()), B, k − i);
    } else {
        return findKthSmallest(A, vector<int>(B.begin() + j, B.end()), k − j);
    }
}
```

Ex.6

Answer: By definition, n! can be expressed as $1 \times 2 \times 3... \times n$. Similarly, $\log(n!) = \log(1 \times 2 \times 3 \times ... \times n) = \log(1) + \log(2) + \log(3) + ... + \log(n)$. At this point the equation can be changed to the following form: $\log(n!) = \frac{\log(n!)}{\log(e)}$. According to the nature of logarithms, $\frac{\log(n!)}{\log(e)}$ can be converted into $n\log(n) - n\log(e)$. At this point, you can go to the last step to get the answer directly: $\Theta(n\log(n))$

Ex.7

Answer: $e^n > 2^{2^n} > n2^n > 3n^6 > (\log n)^{10} > n! > \left(\frac{\log(n)}{\log(\log(n))}\right)^{\frac{\log(n)}{\log(\log(n))}} > \sqrt{n!} > 10n > \log n! > \frac{n!}{4!(n-4)!} > \frac{n}{\log n} > n^{\log(\log(n))} > \frac{n^2}{(\log n)^{10}} > \log n^3 > \frac{\log n}{n}$. There may be mistakes here, because I compared them one by one when I set n to be close to infinity. If $\frac{a}{b}$ is zero, b is large. If $\frac{a}{b}$ is a constant, it means that it is basically the same. If $\frac{a}{b}$ is infinite, it means that a is large.