Homework 3

Jiahao Liu January 27, 2024

1. Consider a collection of n ropes which have lengths $L_1, L_2, \cdots, L_n$, respectively. Two ropes of length L and L' can be connected to form a single rope of length L + L', and doing so has a cost of L + L'. We want to connect the ropes, two at a time, until all ropes are connected to form one long rope. Design an efficient algorithm for finding an order in which to connect all the ropes with minimum total cost. You do not need to prove that your algorithm is correct. (20 points)

If we try to build a binomial tree with all the n rope length. The total cost is just the length of the weighted path of the tree. The tree with minimized length is known as Huffman Tree.

Shortest Two Ropes Frist O(nlogn):

Build a Min-heap with all the n rope length. O(n)
Initialize the total cost to 0.
While the heap size is greater than 1:
    Extract the first two ropes with the shortest length. O(logn)
    Connect the two ropes, record the cost and add the new rope to the heap. O(logn)
Return the total cost.

2. Suppose you want to drive from USC to Santa Monica. Your gas tank, when full, holds enough gas to drive p miles. Suppose there are n gas stations along the route at distances $d_1 \leq d_2 \leq \cdots \leq d_n$ from USC. Assume that the distance between any neighboring gas stations, and the distance between USC and the first gas station, as well as the distance between the last gas station and Santa Monica, are all at most p miles. Assume you start from USC with the tank full. Your goal is to make as few gas stops as possible along the way. Design an efficient algorithm for determining the minimum number of gas stations you must stop at to drive from USC to Santa Monica. Prove that your algorithm is correct. Analyze the time complexity of your algorithm. (25 points)

When you are at the i^{th} gas station, if you have enough gas to go to the $(i+1)^{th}$ gas station, then skip the i^{th} gas station. Otherwise stop at the i^{th} station and fill up the tank.

```cpp
const int N = 10010;
int n, d[N], p;
int main(){
    scanf("%d%d", &n, &p); // number of gas stations and the tank capacity for miles
    for(int i = 0; i < n; i++) scanf("%d", &d[i]);
    int stop = 0, miles = p; // start with tank full
    for(int i = 0; i < n; i++){
        if(d[i] > miles){ // stop to fill gas
            stop++;
            miles += p;
        }
        miles -= d[i];
    }
    cout << stop << endl;
    return 0;
}
```

Proof Correctness: We first show by mathematical induction that our gas stations are always not to the left of the corresponding base stations in any optimal solution. Using this fact, we can then easily show that our solution is optimal using proof by induction. Suppose the algorithm end with k stops in station $s_i \in [1, \cdots, n], i = 1, \cdots, k$, and there is a better solution with k-1 stops in stations $S_j \in [1, \cdots, n], j = 1, \cdots, k-1$. Then we proof that $\forall i = 1, \ldots, k-1, S_i \leq s_i$ must be true:
Base case: Since it is not possible to get to the $(s_1 + 1)^{th}$ gas station without stopping, any solution should stop at either s_1 or a gas station before s_1, thus $S_1 \leq s_1$.
Induction hypothesis: Assume that for the greedy strategy taken by our algorithm $S_c \leq s_c$.
Inductive step: We want to show that $S_{c+1} \leq s_{c+1}$, let's start from S_c, when we pass s_c, we should have at most as much fuel as we did if we had refilled at s_c. However, it is not possible to get to s_{c+1} without stopping, any solution should stop at $S_{c+1} \leq s_{c+1}$.
Now assume that our solution requires k gas stations and the optimal solution requires fewer gas stations:
Because our algorithm add one more stop at s_k, the distance from s_{k-1} to Santa Monica must be greater than p. Consider $S_{k-1} \leq s_{k-1}$, it is impossible to reach Santa Monica from S_{k-1} with the one full tank gas. So, the algorithm returns the minimum number of gas stop.
Time Complexity: Because we just go through the whole distance array, the algorithm runs in O(n) time.

3. Suppose you are given two sequences A and B of n positive integers. Let a_i be the i^{th} number in A, and b_i be the i^{th} number in B. You are allowed to rearrange the numbers within each sequence, but not swap numbers between sequences, then you receive a score of $\prod_{1\leq i\leq n} a_i^{b_i}$. Design an efficient algorithm for permuting the numbers to maximize the resulting score. Prove that your algorithm maximizes the score and analyze your algorithm's running time (25 points)

Sort A and B in the same order O(nlogn):

Sort A and B in ascending or descending order. O(nlogn). Return $\prod_{1\leq i\leq n} a_i^{b_i}$.

Proof Correctness: Suppose our algorithm end with A and B in ascending order, and there is a better solution with A in ascending order while B contains an inversion pair $i < j, b_i > b_j$, then $1 \leq a_i^{b_i - b_j} < a_j^{b_i - b_j} \rightarrow 1 \leq a_i^{b_i} * a_j^{b_j} < a_i^{b_j} * a_j^{b_i}$. We get that by swapping b_i and b_j in B, the score will be greater than the original one.

4. The United States Commission of Southern California Universities (USC-SCU) is researching the impact of class rank on student performance. For this research, they want to find a list of students ordered by GPA containing every student in California. However, each school only has an ordered list of its own students sorted by GPA and the commission needs an algorithm to combine all the lists. Design an efficient algorithm with running time O(mlogn) for combining the lists, where m is the total number of students across all colleges and n is the number of colleges. (20 points)

Min-Heap of size n:

Build a min-heap of size n with the 1^{st} student in each college. O(n)
Initialize the sorted list to empty.
Iterate through the $i^{th} \in [2, \cdots, maxNum]$ student's GPA in each college, where $maxNum$ is the max number of students among n colleges: (loop m times)
    If the GPA is greater than the top of the heap, add the top of heap to ans list, pop the top of the heap and push the GPA into the heap. O(logn)
    else If the GPA is equal to the top of the heap, add the top of heap to ans list twice and pop the top of the heap. O(logn)
    else If the GPA is not 0, add the GPA to ans list.
Add the rest of the heap to ans list. O(n)
Return ans list.

Construct a min-heap H **nitialized with the lowest-GPA student** from each college. Iteratively pop the lowest-GPA student x in H and append it to the end of L, then push the student with the **next higher GPA in x's college** (unless all the students from that college are already in L).
Since the heap H initially contains n students, the runtime complexity of initializing the heap is O(nlogn) = O(mlogn). Since there are at most n students in the heap H at any time, each pop and push heap operation takes O(logn) time. Since we perform m pop and push operations, the runtime complexity of the algorithm is O(mlogn).

5. The array A below holds a max-heap. What will be the order of elements in array A after a new entry with value 19 is inserted into this heap? Show all your work. A = {16, 14, 10, 8, 7, 9, 3, 2, 4, 1}

The following is the change of the heap after inserting 19, and the array implementation of Heap in C++.
A = {16, 14, 10, 8, 7, 9, 3, 2, 4, 1} -> A = {16, 14, 10, 8, 7, 9, 3, 2, 4, 1, 19} -> A = {16, 14, 10, 8, 19, 9, 3, 2, 4, 1, 7} -> A = {16, 19, 10, 8, 14, 9, 3, 2, 4, 1, 7} -> A = {19, 16, 10, 8, 14, 9, 3, 2, 4, 1, 7}

```cpp
const int N = 10010;
int heap[N], hsize;
void down(int x){
    int t = x; // the index of largest element in heap[x], heap[x*2], heap[x*2+1]
    if(x*2 < hsize && heap[x*2] > heap[t]) t = x*2;
    if(x*2+1 < hsize && heap[x*2+1] > heap[t]) t = x*2+1;
    if(t != x){
        swap(heap[t], heap[x]);
        down(t);
    }
}
void up(int x){
    while(x/2 && heap[x/2] < heap[x]){ // greater than its parent
        swap(heap[x/2], heap[x]);
        x /= 2;
    }
```

```
}
int main(){
    scanf("%d", &hsize); // initial heap size
    for(int i = 1; i <= hsize; i++) scanf("%d", &heap[i]); // heap elements
    for(int i = hsize/2; i > 0; i--) down(i); // build max-heap
    heap[++hsize] = 19; // insert 19
    up(hsize);
    for(int i = 1; i <= hsize; i++) printf("%d ", heap[i]); // print the heap
    return 0;
}
```