CSCI-SHU 220: Algorithms @ NYU Shanghai — Last Updated: October 16, 2020
Homework 3 — Due Date: November 9, 2020, 11:00pm

You are allowed to discuss with others but not allow to use references other than the course notes and reference books. Please list your collaborators for each questions. Write your own solutions and make sure you understand them.

There are 55 marks in total (including the bonus). The full mark of this homework is 50.

Enjoy :).

Please specify the following information before submission:

- Your Name: Liangzu Peng
- Your NetID: lp2528
- Collaborators:

Problem 1: Interval scheduling [10 marks]

In the problem of interval scheduling, suppose that instead of always selecting the first interval to finish, we select the last interval to start that is compatible with all previously selected intervals. Describe how this approach is a greedy algorithm, and prove that it yields an optimal solution.

Solution. **Please write down your solution to Problem 1 here.**
We begin by writing down the algorithm.

```
function greedySchedule{
    sort R by starting time;
    A ← set of interval that are compatible;
    while R is not empty{
        select the interval i ∈ R with biggest starting time;
        enter i to A;
        delete from R all interval j that overlaps with i, that is f_j > s_i;
    }
    return A;
}
```

Why this is greedy: by picking the last interval to start, we try to ensure that our resource is not occupied until as late as possible, which give space to more activities before our time interval is occupied.
Proof of optimality: We first show that our greedy solution stays ahead. Suppose there exists an optimal solution O, $\{O_1, \cdots, O_j\}$, with each interval denoted as $[s(O_i),f(O_i)]$. Our greedy algorithm generates a solution of $\{A_1, \cdots, A_k\}$. with each interval denoted as $[s(A_i),f(Ai)]$. We would like to show that $|k| = |j|$.

Theorem 1. *for any $i \leq k$, we have $s(A_i) \geq s(O_i)$.*

We prove this by induction.
base case: $s(A_1) \geq s(O_1)$. Which is true, since our strategy always select the interval with the largest starting time.
inductive case: let $i > 1$, suppose our theorem is true for $i-1$, so that $s(A_{i-1}) \geq s(O_{i-1})$, we need to show that $s(A_i) \geq s(O_i)$.
Note that the optimal solution O is also a set of compatible intervals. So

$$s(O_{i-1}) \geq f(O_i)$$

, since our $O_{i-1}$is the event that happens later, and it can only take place after the previous event O_i finished. I take the optimal solution in its reversed order here, so O_1 is the interval that happens last, andO_j is the event that happens first.
And according to our induction hypothesis, using $s(A_{i-1}) \geq s(O_{i-1})$, we get

$$s(A_{i-1}) \geq f(O_i)$$

So we can see that the interval O_i is in the avaliable sets of choice, since it finishes before A_{i-1} take place, but our greedy algorithm did not pick it, instead we pick A_i, that is because

$$s(A_i) \geq s(O_i)$$

Proof done.
Now we want to show that A is an optimal solution. First, we suppose for the sake of contradiction that A is not optimal. Applying the theroem we achieved above with $i = k$,

$$s(A_k) \geq s(O_k)$$

. Since the size of the optimal solution O, denote by m is greater than k, there is an interval O_{k+1} in the compatible sets and we did not pick it. So

$$f(O_{k+1}) \leq s(A_k)$$

But again our algorithm stops when there is nothing compatible left, so there cannot be such an O_{k+1}. A contradiction.

Problem 2: The weighted total completion time [12 marks]

Suppose that there are n jobs labeled as $1, 2, \ldots, n$. Each job j has length ℓ_j, which is the amount of time required to process the job. Also, each job j has a weight w_j, with higher weights corresponding to jobs of higher priority. A schedule σ specifies an order in which to process the jobs. For example, if $n = 3$ and $\sigma = (2, 1, 3)$, then we first process job 2, then job 1, and finally job 3. The completion time $C_j(\sigma)$ of a job j in a schedule σ is the sum of the lengths of the jobs preceding j in σ, plus the length of j itself. Our goal is to design an algorithm that takes as inputs the weights $w_1, \ldots, w_n$ and the lengths $\ell_1, \ldots, \ell_n$, and outputs a schedule σ^* which minimizes the following objective function:

$$\min_{\sigma} \sum_{j=1}^{n} w_j C_j(\sigma). \tag{1}$$

(a) (4 marks) Consider the greedy algorithm that schedules the jobs in decreasing order of $w_j - \ell_j$. Is this greedy algorithm optimal (in the sense that it always find a minimizer of the above objective function)? Prove or disprove it.

(b) (8 marks) Consider the greedy algorithm that schedules the jobs in decreasing order of w_j/ℓ_j. Is this greedy algorithm optimal? Prove or disprove it.

Solution. **Please write down your solution to Problem 2 here.**

(a) **part a** According to the definition, $C_j(\sigma) = l_1 + \cdots + l_j$, and

$$\min_{\sigma} \sum_{j=1}^{n} w_j C_j(\sigma) = w_1 l_1 + w_2(l_1 + l_2) + \cdots + w_n(l_1 + \cdots + l_n)$$

If we schedule the job in decreasing order of $w_j - l_j$, we denote the job sequence in the form

$$C_1, C_2, \cdots, C_n$$

and for $j < r \leq n$, we have

$$w_j - l_j \geq w_r - l_r$$

To show whether this greedy algorithm proposes an optimal solution, we suppose there exists an optimal solution

$$A_1 \cdots A_n$$

and there exists A_i, A_{i+1} that is not scheduled in order of $w_j - l_j$.We now try to see if we swap this pair A_i, A_{i+1}, the weighted total completion time in the optimal solution gets better. Before swap, for the pair A_i, A_{i+1}, we have

$$w_i - l_i < w_{i+1} - l_{i+1}$$

, and

$$\sum_{j=1}^{n} w_j C_j(\sigma) = \cdots + \cdots + w_i(l_1 + \cdots + l_i) + w_{i+1}(l_1 + \cdots + l_i + l_{i+1}) + \cdots \tag{2}$$

Now we swap A_i, A_{i+1}:

$$\text{After swap:} \cdots + w_{i+1}(l_1 + \cdots + l_{i+1}) + w_i(\cdots + l_{i+1} + li) + \cdots \tag{3}$$

Using equation $3 - 2$, we get

$$w_{i+1} \cdot l_i - w_i \cdot l_{i+1}$$

We would like to determine the sign of this equation. However, the only information we can use is $w_i - l_i < w_{i+1} - l_{i+1}$, which is not enough for us to determine the sign of the above equation. So by swap the order, we do not necessarily find a minimizer solution. For $w_{i+1} \cdot l_i - w_i \cdot l_{i+1}$to be ≥ 0, we need

$$w_{i+1} \cdot l_i > w_i \cdot l_{i+1}$$

divide both sides with $l_i \cdot l_{i+1}$, we find that we need

$$\frac{w_{i+1}}{l_{i+1}} > \frac{w_i}{l_i}$$

(b) **part b** Now we have our greedy schedule A, with the jobs scheduled in decreasing order of $\frac{w_j}{l_j}$, so that for $i < r < n$, we have $\frac{w_i}{l_i} > \frac{w_j}{l_j}$. Suppose there is also an optimal solution O. If A is not optimal, then in O there is a local inversion pair, namely at index $i, i+1$, such that $\frac{w_i}{l_i} < \frac{w_{i+1}}{l_{i+1}}$. We show by exchange argument that if the greedy algorithm is better.

Before swap: at position O_i and O_{i+1}, the partial total completion time is

$$w_i(C + l_i) \text{ and } w_j(C + l_i + l_{i+1}) \tag{4}$$

Swap them,

$$w_{i+1}(C + l_{i+1}) \text{ and } w_i(C + l_{i+1} + l_i) \tag{5}$$

Using $4 - 5$, we get

$$w_{i+1} \cdot l_i - w_i \cdot l_{i+1}$$

knowing that we have $\frac{w_i}{l_i} < \frac{w_{i+1}}{l_{i+1}}$, multiply both sides with $l_{i+1} \cdot l_i$, we have

$$w_{i+1} \cdot l_i > w_i \cdot l_{i+1}$$

so after swap, we obtain a better solution.

Problem 3: Merging lists [11 marks]

Consider the problem of merging k sorted lists $L_1, \ldots, L_k$ of sizes $n_1, \ldots, n_k$. Let $n = n_1 + \cdots + n_k$.

(a) (3 marks) Consider the algorithm that repeatedly merge take two lists from $L_1, \ldots, L_k$ and merge them so that we obtain a single sorted list after $k - 1$ merges. What is the complexity of this algorithm? Assume that merging two lists of sizes ℓ_1 and ℓ_2 respectively entails a cost $\ell_1 + \ell_2$. What is the cost of this algorithm?

(b) (8 marks) The above cost would depend on the order in which the lists are taken and merged. For example, when $k = 3$, there are six possible orders for merging: $(1, 2, 3)$, $(1, 3, 2)$, $(2, 1, 3)$, $(2, 3, 1)$, $(3, 1, 2)$, and $(3, 2, 1)$. Here $(1, 2, 3)$ means that we first merge L_1 and L_2, and the resulting list is taken to be merged with L_3. It is not hard to observe that the orders $(1, 2, 3)$ and $(2, 1, 3)$ have the same cost, and that all these orders give three potentially different costs, say $2n - n_1$, $2n - n_2$, and $2n - n_3$. One can verify that the minimum cost is $2n - \max\{n_1, n_2, n_3\}$, attained when we first merge the two shortest lists. Based on this discussion, give a greedy algorithm to merge the lists, and prove that it is optimal in the sense that it entails the smallest possible cost.

Solution. **Please write down your solution to Problem 3 here.**

(a) **part a** The time complexity of this algorithm is $O(nk)$
Explanation:We take a scenario where we have four lists of size l_1, l_2, l_3, l_4 to merge.
First merge: $l_1 + l_2 \quad t(n - l_4 - l_3)$

Second Merge: $l_1 + l_2 + l_3 \quad t(n - l_4)$
Third Merge: $l_1 + l_2 + l_3 + l_4 \quad t(n)$
Total time: O(kn).
The cost depend on the order we merge the lists. I wrote it as

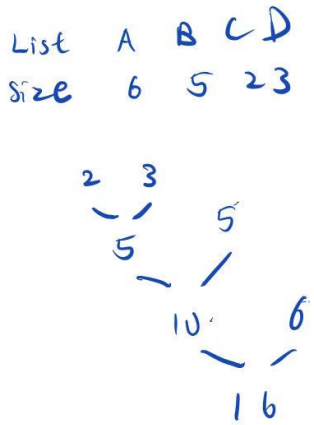
$$(k-1)n - (k-2)l_{last\,merged} - \cdots - 1 \cdot l_{thirdly\,merged}$$

(b) **part b** We first give out the algorithm.

```
OptimalMerge(node Nodes[k]){
    \\each node in Nodes store a pointer to the list
    \\and the length of the list
    T ← BuildMinHeap(Nodes);
    while T.size >= 2 {
        node l1 ← ExtractMin(T);
        node l2 ← ExtractMin(T);
        int newlist[] ← merge(l1.list,l2.list);
        new node n;
        n.list = newlist;
        n.length = (l1.length+l2.length);
        enterMinHeap(n);

    }
    return root(T); // the final merged list
}
```

The process of merging the lists using a min-heap to keep track of the smallest two lists everytime, is actually the process of building a binary merge tree. the cost of the merging



process is the sum of paths from root to external nodes. Assuming that each node contributes to the cost by l_i and the path from the root to that node has depth p_i. Then, the optimality requires a pattern that minimize

$$L = \sum_{i=1}^{n} l_i\, p_i$$

Theorem 2. *We are given an optimal merge tree T*, suppose u and v are leaves of T*, and depth(u) < depth(v), suppose that leaf u represent list y and leaf v represents leaf z, then* $l_y > l_z$

Proof. If $l_y < l_z$, then consider the cost obtained by exchange leaf u and v.

$$\text{Total cost:} \sum_{i+1}^{n} p_i \cdot l_i$$

with p_i the cost and l_i the depth. After exchange, the multiplier on l_y would increase(from depth(u) to depth(v)), and the multiplier on l_z would decrease. Thus the change to the overall sum is (depth(v)-depth(u))$(l_y - l_z)$. If $l_y < l_z$, this change is negative number, contradicting the supposed optimality of the tree before the change. □

The next thing we would like to argue is that, consider a leaf v in $T*$ whose epth is as large as possible, leaf v has a parent u, and we know that $T*$ should be a full binary tree, which could be proved by exchange argument.So u has another child w. Now, we should have

Theorem 3. *w is a leaf of* $T*$.

Proof. If w is not a leaf, then there would be some leaf w' in the subtree below, but then w' would have a depth greater than that of v, contradicting our assumption that v is a leaf of maximum depth in $T*$. □

Now that we could conclude that **there is an optimal binary merge tree** $T*$**, in which the two shortest lists are assigned to leaves that are siblings in** $T*$.

Proof of optimality
We prove the optimality of our merge pattern using induction.

Base Case: when there is only two lists, clearly it's optimal to directly merge them.

Inductive case: we suppose our algorithm is optimal for $k-1$ lists, and we would like to prove that it's also optimal for k lists. We represent the cost obtained by our algorithm as Cost(T) and Cost(T'). There is a relationship between Cost(T') and Cost(T), since the algorithm merges the two shortest lists y*, z* into a single list w, and iteratively finishing the merging process on the smaller set of lists S', by induction hypothesis generating an optimal merge pattern for the rest $k-1$ lists. We conclude that

Theorem 4. *Cost(T') = Cost(T) -* l_w

Proof. since $l_w = l_{y*} + l_{z*}$, and the depth other list nodes stays the same, the depth of $y*$, $z*$

are 1 depth greater than w, we get

$$\begin{aligned} Cost(T) &= \sum_{i=1}^{n} depth(i) \cdot l_i \\ &= l_{y*} \cdot depth_T(y*) + l_{z*} \cdot depth_T(z*) + \sum_{x \neq y*, z*} depth_T(x) \cdot l_x \\ &= l_w(1 + depth_{T'}(w) + \sum_{x \neq y*, z*} depth_{T'}(x) \cdot l_x \\ &= l_w + \sum_{x \in S} l_x \cdot depth_{T'}(x) \\ &= l_w + Cost(T') \end{aligned}$$

□

Now we would prove optimality of our algorithm. Suppose for the sake of contradiction that the merge tree T produced by our greedy algorithm is not optimal, there is an optimal merge tree Z with Cost(Z)¡Cost(T). As we have proved before, there exists such a tree Z such that the two shortest lists $y*$,$z*$ are siblings in the lowest level.
Now, if we delete the leaves $y*$ and $z*$, with their parent as w, we would obtain a tree Z' for the set of lists S'. Similarly, Cost(Z')=Cost(Z)-l_w.
So Cost(Z') would be smaller than Cost(T'), but we have assumed that T' generate an optimal merge pattern for S', which contradicts. We have now finished with proving the optimality of our merge pattern.

Problem 4: Guards for the farmer [11 marks]

A farmer who grows watermelons wants to put guards on her warehouse in the time interval $[0, M]$. There are n guards and the i-th guard can work during time interval $[s_i, f_i]$. The farmer wants to hire a minimum number of guards, such that in any time point $t \in [0, M]$ there is at least one guard whose time interval covers t, i.e., $t \in [s_i, f_i]$. For example, for $M = 15$ and $n = 6$, the guards can work at time intervals $[0, 3], [5, 9], [9, 13], [0, 7], [10, 15]$, and $[7, 12]$, there is a solution of size 3 by hiring 3 guards who work at times: $[0, 7], [7, 12]$, and $[10, 15]$.

(a) (3 marks) under what conditions on the time intervals the farmer can not hire a minimum number of guards, such that in any time point $t \in [0, M]$ there is at least one guard whose time interval covers t, i.e., $t \in [s_i, f_i]$?

(b) (8 marks) Assume that it is possible for the farmer to hire such a minimum number of guards. Help the farmer design a greedy algorithm to hire, suggest an implementation of complexity $O(n \log n)$, and prove its correctness.

Solution. **Please write down your solution to Problem 4 here.**

(a) **part a** When the interval that the guards can work at are disjoint are no guards work at an interval that could cover another one's work.

(b) **part b**

```
optimalHire([[s_1,f_1],···,[s_n,f_n]], s, f)
{
    //sort work interval in ascending order of left ends,
    // for intervals with the same left ends,
    //sort descendingly with respect to right ends
    SORT([[s_1,f_1],···,[s_n,f_n]]);

    S← ∅, t← s;

    for i in 1 to n{

    \\if the point that we already covered have exceeded end point f
    \\end for loop, we are done
        if(t >= f){
            break;
        }

    \\if the skip indicator is not 0, we do not have to
    \\look at current element, it is ought to
    \\be skipped, since we have made the greedy choice
        if(skip > 0){
            skip -= 1;
            continue;
        }


        else{
            container = []; \\we prepare a contianer to store
                            \\all the element that have s(i)<=t
            skip = 0;

            \\traverse, while current element
            \\overlaps with the farthest right we have already reached

            while intervals[i][0] <= end{
                add A[i] into container;
                skip += 1; \\ skip one element in the future
                i +=1;
            }

            \\we find the interval with maximum finishing time
            \\add it to the result set
            \\update our covered right bound t to be finish time
            \\ of our choice

            choice ← max(finish(i)) for i in container;
            S ← S ∪ {choice};
            end ← choice[1];

            \\we need to minus 1 on skip, since we in fact already
```

```
                passed current element in the for loop
                skip← skip−1;
            }
        }
        return S;
    }
```

we start from the left end, we always pick the interval with the largest right end while covering the current starting point. We first sort the input interval array in ascending order according to left end value, for identical left end value pair, we sort in descending order of right end value. The sorting operation take $O(nlogn)$ time, and the second while loop take $O(n)$ time.
further explanation: in the code there is a **skip** flag, this is to indicate if current interval have been looked at and was decided not to choose, since it give ways to our greedy choice – the interval with largest right end while still covering the starting point(0 when no interval have been picked or the right end of the previously picked interval).
If the skip flag is 0, that means we have to start looking at next possible solution. We introduce a **container** array to store all the succeeding interval, whose left end overlaps with our current starting point, we know that next greedy choice has to lie in there, or there would be no solution if a point on the interval is missing.
Then, we determine the interval with the maximum right end inside the container,add this to the result, and set our new starting point to be its finishing time.

Proof. If we take the interval I_{max} which contains the starting point s and has the largest end point, we can change the first interval in any optimal solution to be I_{max}, and it is still an optimal solution. This step could be repeated for the rest interval $I - I_{max}$ and so on, Therefore, our solution is optimal. □

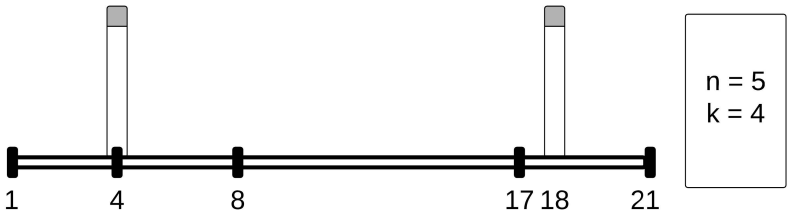
Problem 5: Oh! Beautiful Spring! [11 marks]

Happy Rotter loves his life, so he spends one of his weekends in the park enjoying the brilliant spring time. However, he comes along with two problems during his trip. Can you help him solve them?

(a) (3 marks) On the way the the park Happy Rotter heard many birds singing vividly, and it seems that the birds sang with a pattern! Assume different birds a, b, c, d, e, f, g sing with frequencies $0.15, 0.23, 0.07, 0.18, 0.09, 0.11, 0.17$ respectively, show an optimal binary decoding tree for this pattern. Just draw the tree and label the leaves. No explanations or intermediate calculations required.

(b) (8 marks) Along the long long road in the park many cherry trees are planted, with locations $x_1 \leq x_2 \leq ... \leq x_n$. Now all the trees have blossomed and the view is brilliant! The park staffs set up stalls along the road to serve visitors, but Happy Rotter is not very satisfied – they are either too far away from the trees or too close to each other. There must be a better way to arrange these stalls.

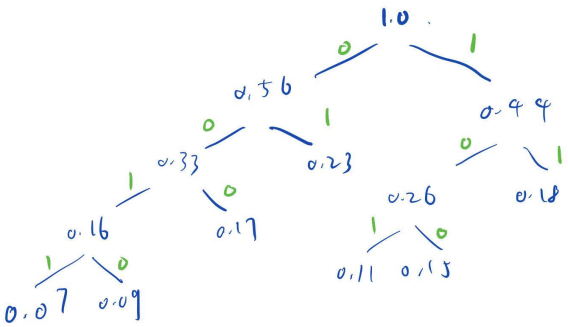
Assume that each stall can serve visitors within k miles, and you may consider that the visitors only gather nearby the cherry trees. Your goal is to serve more visitors using fewer stalls. Given $x_1 \leq ... \leq x_n$ and k, design an efficient algorithm to output a minimum set of stall locations to achieve the goal. State your algorithm clearly. Provide a brief explanation of its correctness. You will get full marks if the time complexity is $O(n)$ and the explanation is good.



Explanation: In the above example, we have $n = 5, k = 4, x = [1, 4, 8, 17, 21]$. One of the optimal solution will use 2 stalls, putting them at $x = 4$ and $x = 18$.

Solution. **Please write down your solution to Problem 5 here.**

(a) **part a**



(b) **part b**

I propose a greedy strategy that every time we meet a point that is not covered by any known stall, we place it as far from this point as possible, exactly at the distance limit for serving this point. Below we give the algorithm.

```
optimalStall(int points[],k){
    stalls = [];
    last_stall = none;

    for point in points{
        if (point is covered by last_stall){
            continue; //if current point has been covered by previous stall
                      // when point-last_stall <=k
        }
```

```
            else{
                new_stall = point + k;
                \\the stall is placed as far to the right as possible
                \\ in order to maximize the effect of this stall

                add new_stall to stalls;
                last_stall = new_stall;
            }
        }

    }
```

Proof. We prove the correctness for our algorithm by induction.
base case:when there is only one point, trivially, it is correct.
inductive case: we suppose that for $k-1$ points our algorithm generates the optimal results. Now we try to prove it also works for the kth element.
Case1:the point is within the service range of the last added stall, then the *if* statement's condition is satisfied, and nothing would be done, no stall would be added.
Case2: the point is outside the previous stall's range, then, since our set of stalls for the previous $k-1$ elements are already optimal, that all the stalls were set to meet the requirement at best, that they are placed ahead as far as possible, it is impossible to make any arrangement to cover the new point. So we must add another stall. In this condition, the stall count goes up by 1, and we again obtained the optimal solution.
We have proven the inductive case. □