Problem 1 (15 points).

Order the following functions in increasing order by their asymptotic growth rate. You do not need to illustrate the middle steps leading to your answer.

1. $f_1(n) = n$
2. $f_2(n) = (\log\log n)^{\log n}$
3. $f_3(n) = (\sum_{1\le k\le n} 1/k)^2$
4. $f_4(n) = \log(n!)$
5. $f_5(n) = (1+1/n)^n$

Solution. The order is $(f_5, f_3, f_1, f_4, f_2)$.

Remarks. $f_5(n) = \Theta(1)$, as $\lim_{n\to\infty}(1+1/n)^n = e$.

$f_3(n) = \Theta((\log n)^2)$, as $\sum_{1\le k\le n} 1/k = \Theta(\log n)$.

$f_4(n) = \log(n!)$, grows slower than $\log(n^n) = n \cdot \log n$, and grows faster than $\log(2^n) = n$.

$\log(f_2(n)) = \log n \cdot \log\log\log n$, while $\log(f_4(n))$ grows slower than $\log n + \log\log n$. Therefore, f_2 grows faster than f_4.

Problem 2 (20 points).

You are given an array $A = (a_1, a_2, \cdots, a_n)$ with n (possibly negative) integers. Design an algorithm to find a subsequence A_1 of A such that A_1 does not contain adjacent elements of A (i.e., if $a_k \in A_1$ then $a_{k-1} \notin A_1$ and $a_{k+1} \notin A_1$) and that the sum of all integers in A_1 is maximized. Your algorithm should run in $O(n)$ time.

Solution. Define $S[i]$ as the maximum sum that can be achieved by a subsequence of $(a_1, ..., a_i)$. For each integer a_i, we have two choices. If we choose a_i, $S[i] = S[i-2] + a_i$; if not, $S[i] = S[i-1]$.

The *pseudo-code*, which includes three steps, is as follows.

Initialization:

```
$S[0] = 0$
$S[1] = max\{0, a_1\}$
```

Iteration:

```
for $i = 2 \cdots n$ :
    $S[i] = \max\{S[i-1], S[i-2] + a_i\}$
endfor
```

Termination: $S[n]$ gives the maximum sum that can be obtained.

Running time: $O(n)$.

Problem 3 (25 points).

There are n rooms $(R_1, R_2, \cdots, R_n)$. In the beginning you are in R_1 and your goal is to reach R_n. If currently you are at R_k, $1 \le k \le n-1$, the maximum distance you can proceed is $p_k \ge 0$; in other words, the next room you can visit must be in $\{R_{k+1}, R_{k+2}, \cdots, R_{k+p_k}\}$. You are given $P = (p_1, p_2, \cdots, p_{n-1})$.

1. **(12 points).** Design an algoithm to decide whether you can reach R_n. Your algorithm should run in $O(n)$ time and use $O(1)$ space.

2. **(13 points).** Suppose that you can reach R_n (which can be determined by running your above algorithm). Design an algorithm to find a route to reach R_n such that the number of rooms you need to visit is minimized. Your algorithm should run in $O(n^2)$ time.

Example 1: if $n = 6$ and $P = (2,1,0,4,5)$, then you cannot reach R_6 (as you will get stuck at R_3).
Example 2: if $n = 6$ and $P = (2,4,1,1,1)$, then you can reach R_6 and the best route is to visit (R_1,R_2,R_6).

Solution.

1. This problem can be solved with a greedy algorithm. We examine all rooms from R_1 to R_n and we simply store the furthest room we can reach as we proceed. Specifically, we maintain a single variable $furthest$, initialized as 1, indicating that we can reach R_1. In th k-th iteration (checking room R_k), $1 \le k \le n$, we first check if R_k is reachable from starting point (i.e., R_1) by checking if $furthest \ge k$. If it is not reachable, we simply return false. Then, we update $furthest$ as $\max\{k + p_k, furthest\}$. We return true immediately if $furthest \ge n$ in any iteration.

   This algorithm takes linear time as it just traverses all rooms once and it takes $O(1)$ space as we only maintain a single variable.

2. We can convert this problem into a shortest path problem. We build a directed $G = (V,E)$ where each room corresponds to a vertex, i.e., $V = \{R_1,R_2,\cdots,R_n\}$ and we add edge from R_k to all vertices $\{R_{k+1},R_{k+2},\cdots,R_{k+p_k}\}$ with weight of 1. Clearly, the shortest path on G from vertex R_1 to R_n gives the optimal route. Notice that G is a directed acyclic graph, and therefore we can find the shortest path in $O(|E|) = O(n^2)$ time.

   (*Remarks: linear algorithms exists for this problem.* For example, we can use BFS. In the loop, we additionally store the layer number and furthest for each layer (we may initialize it to n). When we reaches the furthest of current layer ($currentfurthest$ = $step$), we advance the layer. We can obtain the optimal nodes by back tracking. We can also use a greedy algorithm: at R_k we look at all reachable room, and select the one with larger $k + p_k$.)

Problem 4 (25 points).

Recall that a *minimum spanning tree* of $G = (V,E)$ with edge weight $w(e) > 0$ for any $e \in E$ is defined as a spanning tree T of G such that its total weight, i.e., $\sum_{e\in T} w(e)$, is minimized (over all spanning trees). Consider two related definitions: a *min-max spanning tree* is defined as a spanning tree T of G such that $\max_{e\in T} w(e)$ is minimized; a *max-min spanning tree* is defined as a spanning tree T of G such that $\min_{e\in T} w(e)$ is maximized.

1. **(7 points).** Prove or give a counter-example: every minimum spanning tree is also a min-max spanning tree.

2. **(7 points).** Prove or give a counter-example: every min-max spanning tree is also a minimum spanning tree.

3. **(11 points).** Given $G = (V,E)$ with edge weight $w(e) > 0$ for any $e \in E$, design an algorithm to find a max-min spanning tree of G. Your algorithm should run in $O((|V| + |E|) \cdot \log |V|)$ time. Prove that your algorithm is correct.

Solution.

1. Every minimum spanning tree is also a min-max spanning tree. Suppose a minimum spanning tree T is not a min-max spanning tree. It means there exist an edge e and a min-max spanning tree

T', where $e \in T$ and $w(e) > \max_{e' \in T'} w(e')$. If we remove e from T, T will be separated into two parts, say A and B. In T', there must be (at least) one edge connecting A and B; we denote this edge as e^*. Since e^* connect A and B, then $T - \{e\} + \{e^*\}$ is also an spanning tree of the graph, but its total weight is decreased, as we have $w(e) > w(e^*)$. This contradict to the fact that T is a minimum spanning tree.

2. Every min-max spanning tree is not necessarily a minimum spanning tree. Consider a graph with 4 vertices (A,B,C,D) and four edges and weights $w(A,B) = 10$, $w(B,C) = 1$, $w(C,D) = 2$, and $w(B,D) = 3$. The minimum spanning tree includes edges $\{(A,B),(B,C),(C,D)\}$; while $\{(A,B),(B,C),(B,D)\}$ is a min-max spanning tree but it is not a minimum spanning tree.

3. We can convert the problem of max-min spanning tree to the problem of min-max tree. For each edge $e \in E$, we set its weight $w(e)$ as $C - w(e)$ to get a new graph G', where $C = \max_{e \in E} w(e)$ is the largest weight. Clearly, the min-max spanning tree of G' is the max-min spanning tree of G. As we proved in (1), every minimum spanning tree is a min-max spanning tree. Hence, we can run kruskal's algorithm on G' to get one minimum spanning tree T', which will also be one max-min spanning tree of G.

   The construction of G' takes $O(|E|)$ time and the kruskal's algorithm runs in $O((|V|+|E|)\cdot\log|V|)$ time, so the total time complexity is $O((|V|+|E|)\cdot\log|V|)$.

   The correctness of above algorithm is implied by two facts: (a) any minimum spanning tree is also a min-max spanning tree, and therefore we can use any algorithm for minimum spanning tree to find a min-max spanning tree; (b), min-max spanning tree of G' (constructed above) is identical to the max-min spanning tree of G; this is because $\min_{T \in G'} \max_{e \in T} C - w(e) = C \cdot (|V| - 1) - \max_{T \in G} \min_{e \in T} w(e)$ as each spanning tree will have exactly $|V| - 1$ edges.

Problem 5 (15 points).

There are n trains $(X_1, X_2, \cdots, X_n)$ moving in the same direction on parallel tracks. Train X_k moves at constant speed v_k, and at time $t = 0$, is at position s_k. Train X_k will be awarded, if there exists a time period $[t_1, t_2]$ of length at least δ (i.e., $0 \le t_1 < t_2$ and $t_2 - t_1 \ge \delta$) such that during this entire time peroid X_k is in front of all other trains (it is fine if X_k is behind some other train prior to t_1 or X_k is surpassed by some other train after t_2). Given v_k and s_k, $1 \le k \le n$, and $\delta > 0$, design an algorithm to list all trains that will be awarded. Your algorithm should run in $O(n \cdot \log n)$ time.

Solution. Let $y_k(t)$ be the position of X_k at time t. Clearly $y_k(t) = s_k + v_k \cdot t$. We first find those trains that are ahead of other trains at some time point (for now not necessarily spanning δ). This can be solved by transforming into the half-plane-intersection problem. Note that train X_k is in front of other trains at time t if and only if $y_k(t) > y_i(t)$ for any $i \neq k$. Therefore, we compute the *upper envelop* of these n lines: $y_k(t) = s_k + v_k \cdot t$, $1 \le k \le n$, which is equivalent to finding the intersection of these n half planes: $y_k(t) \ge s_k + v_k \cdot t$, $1 \le k \le n$. Clearly, trains corresponding to those lines consisting of the upper envelop (i.e., on the boundary of the intersection) are trains that can be possibly awarded, as each such train is ahead of other trains at some time point. Trains do not correspond to any line on the boundary cannot be awarded.

We can use the divide-and-conquer algorithm (introduced in class) to find the intersection of above n half-planes. Recall that this algorithm returns two sorted lists of lines, representing the left boundary and right boundary of the intersecting region respectively. In fact in our case, the left boundary must be empty, as all lines have positive slope and all half-planes are upper part of the 2D space (i.e., in the form of $y \ge ax + b$). We now process the sorted list of lines in the right boundary to determine whether each train will be actually awarded. For each line L (corresponding to a train) we consider its left neighbor L_1 and right neighbor L_2 in the sorted list, and compute the interesecting point (t_1, y_1) between L and L_1 and the intersecting point (t_2, y_2) between L and L_2. (If L is the leftmost line in the sorted list, set $t_1 = 0$;

if L is the rightmost line set $t_2 = \infty$.) If we have $0 \leq t_1 < t_2$ and $t_2 - t_1 \geq \delta$, then the train (corresponding to line L) will be awarded. Examine all lines in the sorted list in this way gives us the set of trains that will be awarded.

The running time of finding the upper envelop takes $O(n \cdot \log n)$ time and the postprocessing of the sorted list takes linear time. Therefore, the entire algorithm runs in $O(n \cdot \log n)$ time.