Problem 1. (?%)

Optimal Substructure

首先，原問題可以先轉成以下的形式:“給定一個容量 C，以及一個樹狀的流量圖 T，求把所有的水流都堵住最少需要多少水壩 K。” 用數學形式表示的話，就是 $P(T, C) \to K$。接下來對 C 作二分搜，便可找到在水壩需求量不超過原本給定的 k 的情況下最小的 C。

接下來，定義 $f(v)$ 是流過點 v 的總流量，$Ch(v)$ 是 v 的 children 所形成的集合。然後給定一個樹狀的流量圖 T，定義 $V^* = \{v | f(v) \leq C\}$。

對於後面的問題，首先，選擇一個點 $v \in V^*$，把 T 中以 v 為 root 的子樹移除，然後對於 v 到 T 的 root 的路徑上的所有點 $u_1, u_2.....u_L$(其中 $u_1 = v, u_L = root$，L 是路徑長)，$f(u_i) - = f(v)$ for i from 1 to L。經過這些操作，可以得到一顆新的樹狀圖 T'。然後定義子問題如下: 給定一樣的 C 以及新的 T'‘, 求最小的需要的水壩數量 K',i.e.$P(T', C) \to K'$。

現在令 $P(T, C)$ 的其中一組最佳解為 R，R 是有放水壩的點 v 所成的集合。假設 R 中包含 $v^* \in V^*$。則利用 v^*，我們可以用上述的方法獲得一個樹狀圖 T'。給定 T' 及一樣的 C，假設 R' 是 $P(T', C)$ 的最佳解，則 $|R'| + 1 = |R|$。若 R' 不是 $P(T', C)$ 的最佳解，則存在有 R” 使得 $|R"| < |R'|$，但將 v^* 加入 R” 中，可以獲得另一個原問題的解 R^*，但是 $|R"| + 1 = |R^*| < |R|$，表示 R 不是最佳解，矛盾。所以 R' 是 $P(T', C)$ 的最佳解。

Greedy Property

我們宣稱 $P(T, C)$ 有 greedy property，而且定義 greedy choice 如下: 找到第一個點 v，使得 $f(v) > C$，但對於所有的 $v' \in Ch(v)$，$f(v') \leq C$。然後在這些小孩 $Ch(v)$ 中，找到流量最大的，叫它 v”，將他堵住，並更新 v” 到 T 的 root 的路徑上，所有點的流量。

我們可以發現，$Ch(v)$ 中，至少要有一點被放置水壩，不然我們沒有辦法將全部的水流都堵上。那麼，假設 R 是 $P(T, C)$ 的 optimal solution。如果 R 中有 v”，則得證。若 R 中包含一些點 $U = \{u_1, u_2...u_k\}$，使得所有的 u_i，$u_i \in Ch(v)$ 但 $u_i \neq v$”，則任選一個點 $u_j \in U$ 與 v” 交換，我們保證新的 solution R^* 也是合法的，因為後來的 $f(v") \leq$ 原本的 $f(v)$，所以若原本的 $f(v)$ 最後可以被堵上，則後來的 $f(v)$” 也可以被堵上。然後 $|R^*| = |R|$，所以 R^* 也是一個最佳解。

.

Problem 2. (15%)

pseudo code

```
1  // An undirectional graph G=(V,E) is given.
2  set Prim(set V, set E)
3  {
4  CHOSEN_VERTICES = ∅
5  CHOSEN_EDGES = ∅
6  pick some vertex v∈V,
7      CHOSEN_VERTICES += v
8
9  while( |CHOSEN_VERTICES| < |E| )
10     // 若某一edge連接的兩個vertices，一個在CHOSEN_VERTICES裡，一個不是，
11     // 則將此edge加到CANDIDATE_EDGES
12     CANDIDATE_EDGES = ∅
13     for all e in E
14         if ( (e.v1 ∈ CHOSEN_VERTICES) xor (e.v2 ∈ CHOSEN_VERTICES) )
15             CANDIDATE_EDGES += e
16
17     if(CANDIDATE_EDGES != ∅)
18         // find the edge with minimum cost
19         min_e = min(CANDIDATE_EDGES)
20         CHOSEN_EDGES += min.e
21         // e.v1 和 e.v2 其中一個是 v，重複加進SET也沒關係
22         CHOSEN_VERTICES += min_e.v1
23         CHOSEN_VERTICES += min_e.v2
24     else
25         // To reach this clause,
26         //  it means |CHOSEN_VERTICES|<|E| but |CANDIDATE_EDGES|==0,
27         //  which means G is not a connected graph.
28         // So we randomly pick another unchosen vertex to start again
29         pick some vertex v∈V && v∉CHOSEN_VERTICES,
30             CHOSEN_VERTICES += v
31 return CHOSEN_EDGES
32 }
```

optimal substructure

證明「母問題的最佳解」包含「子問題的最佳解」

我們要證明的是：

「$G_{(V,E)}$ 的最佳解」=「e」+「$G_{(V-v,E-e_v)}$ 的最佳解」

(選任一 $v \in V$；$v \in e$ 且 $e \in$「$G_{(V,E)}$ 的最佳解」; e_v 為含 v 的所有邊)

若 $CHOSEN\_EDGES_{(V,E)}$ 是「$G_{(V,E)}$ 的最佳解」，即為 $G_{(V,E)}$ 的最小生成樹之邊集合，

則 $CHOSEN\_EDGES_{(V,E)} = \{e\} \cup CHOSEN\_EDGES_{(V-v,E-e_v)}$,

($e \in CHOSEN\_EDGES_{(V,E)}$，$e_v$ 為含 v 的所有邊)

若 $CHOSEN\_EDGES_{(V-v,E-e_v)}$ 非「$G_{(V-v,E-e_v)}$ 的最佳解」，設 $MIN_{(V-v,E-e_v)}$ 為「$G_{(V-v,E-e_v)}$ 的最佳解」

則 $\{e\} + MIN_{(V-v,E-e_v)}$ 可構成比 $CHOSEN\_EDGES_{(V,E)}$ 更佳之解，矛盾

故 $CHOSEN\_EDGES_{(V-v,E-e_v)}$ 也是「$G_{(V-v,E-e_v)}$ 的最佳解」

greedy property

設我們能拿到一 MST_G，以之和我們用 prim 做出來的 $PRIM_G$ 相互比較

將 $PRIM_G$ 中每個 edge 依照加入的時間標上編號 e_1 至 e_n

從編號最大的開始比 (即從最後加入的邊開始比)

由上述 optimal substructure 的證明可知，若原本是 MST，則拿掉一屬於該 MST 的邊之後，所形成的也是該子問題的 MST

故，將 $PRIM_G$ 與 MST_G 比對的方法為，從 $PRIM_G$ 中編號最大的 edge 開始：

假如 e_i 也屬於 MST_G，則將之從 MST_G 和 $PRIM_G$ 中拿掉，並形成 MST'_G 及 $PRIM'_G$，再繼續用 MST'_G 和 $PRIM'_G$ 比對

如此不斷重複做，直到遇到 $e_i \notin MST_G$

因為此時 e_i 已為 $PRIM'_G$ 中編號最大的邊，即從 prim 演算法過程來看，是最後加入的邊，故：

(1) 在 $PRIM'_G$ 中，e_i 必為連接一點 v 及其他點集合 V 的一邊 (即 e_i 並不會是連接兩個點集合)

(2) 在 $PRIM'_G$ 中，v 只有 e_i 一邊向 V 連接

而在 MST'_G 中，必至少有一邊 e' 連接 V 與 v(因為 MST'_G 是 spanning tree)，且 $e' != e$

從 prim 演算法的過程可知，e_i 為所有一邊屬於 V、一邊屬於 v 的邊之中 cost 最小的

而 e' 也是一邊屬於 V、一邊屬於 v 的邊，故 cost(e_i) 必小於等於 cost(e')

所以我們可以將 MST'_G 中的 e' 置換成 e_i，且新形成的 spanning tree MST''_G 的 cost 會比 MST'_G 小

以此 MST''_G 再繼續與 $PRIM'_G$ 做下去 (遇到相同的邊拿掉，遇到不同的邊則以上述方法置換)

則最後我們可以把 MST_G 換成 $PRIM_G$，且我們可以保證在置換的過程中所形成的 MST'_G 或 MST''_G 的 cost 一定小於等於 MST_G

所以我們可以證明，依此 greedy choice 我們可以建構出 minimum cost spanning tree

Problem 3. Task Scheduler (20%)

1. (5%)

   (a) Greedy choice

   Before the poof, let us formulate what we want to prove. Given a set of tasks and task g with maximum f, there exists an optimal task schedule S^* satisfying the following condition: If task g has no dependency on other tasks, $t_g < t_i, \forall i \neq g$. Otherwise, task g is processed right after the task it depends on. Generally, task g is composed of many tasks denoted as $g_1, \ldots, g_k$.

   First, we prove the case that task g has no dependency on other tasks. Assume there exist a task e processed before task g in S^*. We can construct a new schedule $\hat{S}$ from S^* by making task g processed first. Denote the total cost of S^* as C^* and the total cost of $\hat{S}$ as $\hat{C}$. Then we have

   $$\begin{aligned}\hat{C} &= C^* + kf_e - \sum_{j=1}^{k} f_{g_j} \\ &= C^* + kf_e - kf_g.\end{aligned}$$

   If $f_e = f_g$, $\hat{C} = C^*$. Thus, $\hat{S}$ is also a optimal schedule. Otherwise, $\hat{C} < C^*$ due to task g with maximum f. It is a contradiction, so task g must finish before other tasks.

   Second, we prove the case that task g depends on a task p. Assume there exist a task e processed between task p and g in S^*. We can construct a new schedule $\hat{S}$ from S^* by making task g processed right after task p. Then we also have $\hat{C} = C^* + kf_e - kf_g$. It is as same as the first case. We deduce a contradiction, so task g must run right after task p.

   (b) Optimal substructure

   Given a set of tasks T and a selected task x, we assume that there exists an optimal task schedule S^* containing the order decided by task x. That is, If task x has no dependency on other tasks, task x finishes before other tasks. Otherwise, task x are processed right after the task it depends on. and the optimal task schedule of the subproblem constructed by task x as S^*_{T-x}. What we want to prove is that there exists an optimal task schedule S^* made up of S^*_{T-x} and the order decied by task x.

   We will prove it by contradiction. In Generall, task x is composed of many tasks, but for simplicity, we only consider task x is a single task. Assume S^* is made up of S'_{T-x} and the order decied by task x, and the schedule composed of S^*_{T-x} and the order decied by task x is not optimal for T. Denote the total cost of a schedule S as $C(S)$. For the

case that task x has no dependency on other tasks,

$$C(S^*) = f_x + C(S'_{T-x}) + \sum_{j \in T-x} f_j$$
$$> f_x + C(S^*_{T-x}) + \sum_{j \in T-x} f_j,$$

which is a contradiction. For the case that task x depends on a task p, $C(S^*) = C(S'_{T-x}) > C(S^*_{T-x})$, which is a contradiction.

2. (5%)

```
typedef struct PseudoTask {
        double f   \\ default f_i
        double sum_f  \\ default f_i
        double cost   \\ default f_i
        int size   \\ default 1
        bool visit \\ default False
} Task
Task tasks[n]
int depend[n]
\\ If task i depends task j, depend[i] = j.
\\ If task i has no dependency on other tasks, depend[i] = -1.
int total_cost = 0, total_size = 0
double Find_optimal_schedule()
for k = 1 to n
    \\Search the task with maximum f.
    double max_f = -1
    int max_task = -1
    for i = 1 to n
        if !tasks[i].visit && tasks[i].f > max_f
            max_f = tasks[i].f
            max_task = i

    tasks[max_task].visit = True
    parent = depend[max_task]
    if parent == -1
        \\ Delete max_task
        for i = 1 to n
            if depend[i] == max_tasks
```

```
                depend[i] = -1
        total_cost += tasks[max_task].cost
            + tasks[max_task].sum_f*total_size
        total_size += tasks[max_task].size
    else
        \\ Merge tasks
        for i = 1 to n
            if depend[i] == max_tasks
                depend[i] = parent
        tasks[parent].cost += tasks[max_task].cost
            + tasks[max_task].sum_f*tasks[parent].size
        tasks[parent].size += tasks[max_task].size
        tasks[parent].sum_f += tasks[max_task].sum_f
        tasks[parent].f = tasks[parent].sum_f/tasks[parent].size
return  total_cost
```

The procedure conducts n subproblems, and each subproblem requires $O(n)$ instructions. Therefore, the procedure runs in $O(n^2)$ times.

3. (5%)

```
disjoint_set dependency \\ child set
heap max_heap
\\ Assume all tasks have pushed into the heap.
\\ Compare tasks by f.
int total_cost = 0, total_size = 0
double Find_optimal_schedule()
for k = 1 to n
    \\Search the task with maximum f.
    max_tasks = max_heap.pop()

    parent = disjoint_set.find(max_task)
    if disjoint_set.find(parent) == deleted_flag
        disjoint_set.set_parent(max_task)
         \\ max_task.parent = -1
        parent = -1
    if parent == -1
        \\ Delete max_task
        disjoint_set.set_deleted_flag(max_task)
```

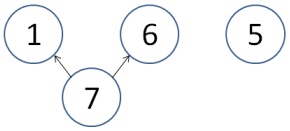

Figure 1: A counterexample

```
19          \\ max_task.parent = deleted_flag
20          total_cost += tasks[max_task].cost
21              + tasks[max_task].sum_f*total_size
22          total_size += tasks[max_task].size
23      else
24          \\ Merge tasks
25          disjoint_set.merge(parent, max_task)
26          tasks[parent].cost += tasks[max_task].cost
27              + tasks[max_task].sum_f*tasks[parent].size
28          tasks[parent].size += tasks[max_task].size
29          tasks[parent].sum_f += tasks[max_task].sum_f
30          tasks[parent].f = tasks[parent].sum_f/tasks[parent].size
31      return  total_cost
```

In the above pseudo code, we utilize heap to find the task with maximum f and apply a modified disjoint set to merge or delete tasks. Now, each subproblem requires $O(\log(n))$ time. Therefore, the procedure runs in $O(n\log(n))$ times.

4. (5%) By definition, we combine the task with maximum f with those tasks it directly depends on into a new task, and all sub-tasks in the new task must be processed consecutively. From this observation, we can find counterexamples, for instance, Figure **??**. The optimal schedule is $6 \to 5 \to 1 \to 7$, not $5 \to 6 \to 1 \to 7$.

Problem 4. Super Computer Revised(15%)

1. Solution:

   (1) Set $p = 0$. Sort the list of jobs with d_i in descending order.

   (2) Construct an empty max-heap of jobs that use p_i as the key.

   (3) Add all jobs in the list with $d_i \geq t$ into the heap and delete them from the list.

   (4) Do find_max and delete_max on the heap, set $p = p + p_i$ and set $t = t - 1$. If the heap is empty, just set $t = t - 1$.

   (5) Goto (3) if $t > 0$.

   (6) The profit is p.

   - subproblem: Let $t = \max_{1 \leq i \leq n} \{d_i\}, D = (d_1, d_2, d_3, ..., d_n), P = (p_1, p_2, p_3, ..., p_n)$.
     If job i is scheduled during $[t-1, t)$, the profit would be
     $$f(D, P, 0, t) = \begin{cases} p_i + f(D', P', 0, t-1) & \text{, if } t \geq 0 \\ 0 & \text{, otherwise} \end{cases}$$
     where D' is D without d_i, P' is P without p_i.
   - optimal substructure: To prove that the problem exhibits optimal substructure, we need to prove that the optimal solution is produced by combining optimal solutions of the subproblems. If an optimal solution O is produced by scheduling job i during $[t-1, t)$, then the schedule during $[0, t-1)$ in O must be an optimal solution of scheduling D', P' during $[0, t-1)$ or we will have a better solution. Therefore, this problem exhibits optimal substructure.
   - greedy choice property: If there exists an optimal solution O and the last job is i, which is different from job j, the job we scheduled.
     If job j is scheduled at other time in O, swap it with job i in O and forms O'. O' is still a valid solution, since both d_i and d_j is greater or equals to t. Since O is optimal, O' would also be optimal.
     If the job j is not scheduled at other time in O, replace job i with job j and forms O'. If $p_i > p_j$, our algorithm would have pick job i. If $p_i = p_j$, both O and O' are optimal solutions. If $p_i < p_j$, O' is optimal while O isn't.

2. Solution:

   (1) Set $c = 0$. Sort the list of power generators with s_i in ascending order.

   (2) Construct an empty max-heap of power generators that use f_i as the key.

   (3) Add all power generators in the list with $s_i \leq S$ into the heap and remove them from the list.

(4) If the heap is empty, there is no valid solution. If the heap is not empty, do find_max and delete_max on the heap, set $c = c + 1$ and set $S = f_{max}$.

(5) Goto (3) if $S < F$.

- subproblem: Let $S_g = (s_1, s_2, s_3, ..., s_n), F_g = (f_1, f_2, f_3, ..., f_n)$.
  If power generator i is the first power generator to be launched. We assume that $s_i \leq S$ or this won't be a valid solution. The nubmer of power generators required would be
  $$f(S_g, F_g, S, F) = \begin{cases} 1 + f(S'_g, F'_g, \max\{S, f_i\}, F) & \text{, if } S < F \\ 0 & \text{, otherwise} \end{cases}$$
  where S'_g is S_g without s_i, F'_g is F_g without f_i.
- optimal substructure: To prove that the problem exhibits optimal substructure, we need to prove that the optimal solution is produced by combining optimal solutions of the subproblems. If an optimal solution O is produced by launch power generator i first, then the power generators launched during $[0, t-1)$ in O must be an optimal solution of selecting power generators among the rest $n-1$ generators during time $[f_i, F)$. If we can find a better solution of the subproblem, we can use less power generators to maintain the power supply. Therefore, this problem exhibits optimal substructure.
- greedy choice property: Let an optimal solution O that select power generator i instead of power generator j where $s_i \leq S, s_j \leq S, f_i < f_j$. Then replacing power generator i with power generator j is still an optimal and valid solution.

Problem 5. (20%)

1. (5%) 首先定義 subproblem 為
   subproblem(i,j) = 找出把前 i 個 value 切成 j 段能得到的最小 q
   所以 S(i,j) = $\min_{m=[j-1 to i-1]}$(max(S(i-m,j-1),sum(seq[m to i])))
   要證明 problem 有 optimal substructure
   首先已知 S(i,j) 是最佳解且 S(i,j) = max(S(i-m,j-1),sum(seq[m to i]))
   假設 S(i-m,j-1) 不是最佳解
   則存在 S'(i-m,j-1) < S(i-m,j-1)
   如果 S(i-m,j-1) <= sum(seq[m to i])
   則 S(i,j) 不變仍為最佳解
   如果 S(i-m,j-1) > sum(seq[m to i])
   則 S(i,j) = S(i-m,j-1) > max(S'(i-m,j-1),sum(seq[m to i])) = S'(i,j)
   表示 S(i,j) 非最佳解 (矛盾)
   ⇒ S(i-m,j-1) 必為最佳解
   ⇒ 最佳解必包含 subproblem 的最佳解
   ⇒ optimal substructure 成立

2. (5%)

```
1  int dp(seq, size, k)
2      int table[MAXK][MAXN]
3      for j = 1 to size do
4          for i = 1 to min(k, j) do
5              if i == 1 then table[i][j] = sum(seq[1 to j])
6              else if i == j then table[i][j] = max(seq[1 to j])
7              else
8                  for p = 1 to j-1
9                      min = max(table[i-1][p], sum(seq[p+1 to j]))
10                 table[i][j] = min
11     return table[k][size]
```

3. (10%) 原問題可以轉換成” 給定一個 q, 找出每一段 sum 都 <=q 的情況下最少的分段數 k”
那對於每一個給定 q, 我們宣稱問題有 greedy property, 而且定義有一個 greedy choice 為從第一個點開始往後算分段和，如果分段和沒超過 q 就繼續，只要發現和會超過 q 就分段
為了證明這樣的 greedy choice(G) 做出來的是一種最佳解，我們先假設他不是最佳解, 則另有最佳解 G’
但在保證每一段都不大於 q 的情況下
G’ 在每一段的選擇就是
如果分段和超過 q 就分段 (1)
還可以再塞下一個 value 卻直接分段 (2)
當 G’ 選擇 (1) ⇒ 和 G 相同
當 G’ 選擇 (2) 後, 剩下所需分段的 seq’ 會比選擇 (1) 的剩餘 seq* 長, 那我們換成 (1) 的挑法仍不違反前提, 且剩餘 seq 較短 ⇒ |G’(1)|<=|G’(2)|
然後當 G’ 全部選擇 (1) 的情況下 =G 的挑法 ⇒ |G|<=|G’|
所以 G 仍為最佳解

```
1  int greedy(seq, size, k)
2      l = 0, r = sum(seq)
3      while l < r do
4          cnt = 0, mid = (l+r)/2, tmp = 0
5          for i = 1 to size do
6              if tmp+seq[i] > mid then
7                  cnt++, tmp = seq[i]
8              else tmp += seq[i]
9          cnt ++
10         if cnt == k then r = mid
11         else if cnt > k then l = mid + 1
12         else r = mid
13     return l
```