Problem 1:

(1)(Text Book p.44)

The *O*-notation gives an asymptotic upper bound $g(n)$ on a function $f(n)$, where

$O(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that

$0 \le f(n) \le cg(n)$ for all $n \ge n_0\}$.

(2)

$\because$ $f(n) = O(g(n))$

➔ there exist positive constants c_1 and n_1 such that $0 \le f(n) \le c_1 g(n)$ for all $n \ge n_1$

$\because$ $g(n) = O(h(n))$

➔ there exist positive constants c_2 and n_2 such that $0 \le g(n) \le c_2 h(n)$ for all $n \ge n_2$

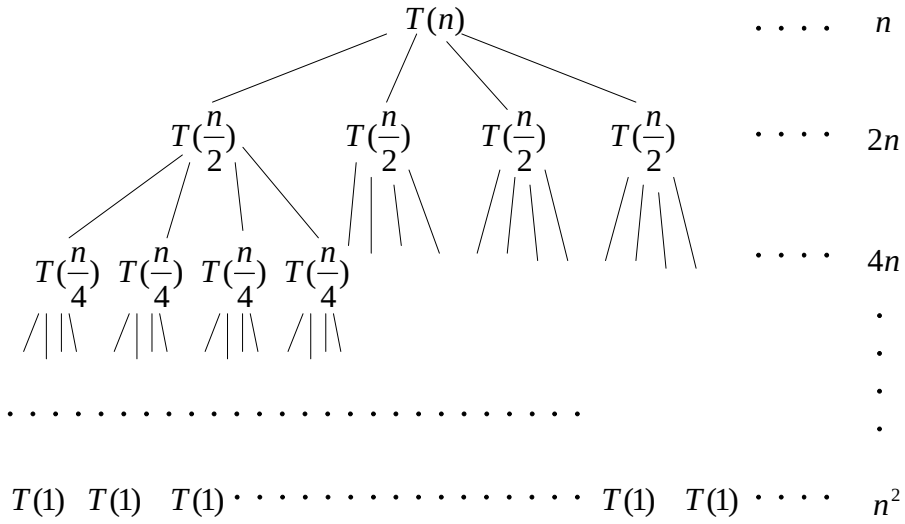
Let $n_0 = \max\{ n_1, n_2\}$ and $c = c_1 c_2$. We can conclude that

$0 \le f(n) \le c_1 g(n) \le c_1(c_2 h(n)) = ch(n)$ for all $n \ge n_0$

$\therefore$ $f(n) = O(h(n))$

Problem 2：

(1)



因爲題目上是floor，因此去掉floor來估計時間只會比原本大不會比原本小

$$\begin{aligned} T(n) &\le 4T(n/2) + n \\ &\le 4(4T(n/4) + n/2) + n \\ &\le n^2 T(1) + n^2/2 + \ldots + 4n + 2n + n \end{aligned}$$

$\leq 2n^2$

$= O(n^2)$

(2)

假設$T(n) \leq cn^2$

當$n = 1$時

$T(1) = 1 \leq c(1)$ 取$c \geq 1$即可

設$n \geq n_0 = 1$, $c \geq 1$時

$T(n) \leq cn^2$ 對所有$n = 1 \sim (k\text{-}1)$的正整數皆成立

則$n = k$時

$T(k) = 4T(\text{floor}(k/2)) + k$

$\leq 4c(\text{floor}(k/2))^2 + k$ 因爲$\text{floor}(k/2) < k$

$\leq 4c(k/2)^2 + k$

$= ck^2 + k$

$\leq ck^2$ 不可能，因爲$k \geq 2$

重新假設$T(n) \leq cn^2 - bn$

當$n = 1$時

$T(1) = 1 \leq c - b$ 取$c - b \geq 1$ 即可

設$n \geq n_0 = 1$, $c - b \geq 1$時

$T(n) \leq cn^2 - bn$ 對所有$n = 1 \sim (k\text{-}1)$的正整數皆成立

則$n = k$時

$T(k) = 4T(\text{floor}(k/2)) + k$

$\leq 4c(\text{floor}(k/2))^2 - 4b(\text{floor}(k/2)) + k$ 因爲$\text{floor}(k/2) < k$

$\leq 4c(k/2)^2 - 4b(k/2) + k$

$= ck^2 - 2bk + k$

$= ck^2 - (2b - 1)k$

$\leq ck^2 - bk$

取$b \geq 1$即可

故取$b = 1$, $c = 2$即可同時符合$c - b \geq 1$, $b \geq 1$

原式$T(n) \leq 2n^2 - n$在$n \geq 1$時恆成立

得證$T(n) = O(n^2)$

Problem 3

(1)

1.把每一個 array A_i 的第一個 element 放進一個 binary heap，此時 heap 裡面有 k 個 elements。

2.Exact-Min 取出 heap 的最小值放進 array B 中，在看取出的最小值原本是在哪一個 array A_i 中，再從 A_i 取下一個 element 插入到 heap 中。如果該 array 已經空了，就不用再插入新的 element。

3.一直重複步驟 2 直到 heap 空了爲止。

把每個 array 中最小的取出來，共有 k 個候選值，因爲都是各個 array 最小的值，所以全部 n 個 elements 中的最小值一定會在這 k 個裡面。將這 k 個候選值一起比較，然後挑出其中最小的值，這會是所有 element 中最小的。此時，剩下 k – 1 個候選值，將剛剛被挑走的那個 array 在提供一個候選人出來補齊。第二小的就一定是在這 k 個候選人最小的一個，因爲最小的已經在第一回合被挑走了，且補上了一個僅次於剛剛被挑走最小的候選值。以此類推，每次都維持好這些候選值，從中挑出最小的，就可以從小到大的挑出所有人。

(2)

Make-Heap 用了 $O(k)$。接下來每次 heap 都固定拿掉一個 element，插入一個 element 或沒插入，所以 heap 還是維持小於等於 k 個 elements。所以跑一次步驟 2 需要 $O(\log k)$。總共需要 n 次的步驟 2，所以 total time = $O(k) + n * O(\log k) = O(n\log k)$

Problem 4:

請參考課本及講義 Section 7.4

Problem 5：

方法

1. 把每個數字另存到一個 structure，這個 structure 紀錄這個數字原本屬於哪個 group，以及這個數字是多少。
2. 把每個 structure 中的數字轉成 n 進位額外存起來。
3. 對這個 structure 中的 n 進位數字套用 Radix Sort，得到全體依照 structure 中數字從小到大排序的 structure。
4. 對目前這個排序的 structure 原本 group 的數值套用 Bucket Sort，把這些 structure 分成各個組別。
5. 最後的結果就是答案，如果要顯示數字就把 structure 中的數字

額外再印出來即可

正確性

經過第三個步驟，我們會得到全體從小到大排序好的數列，接下來第四個步驟我們從小往大看過去，利用 Bucket Sort 分成各個組別，每個組別很明顯也會是從小到大排序好的，因此最後就是題目要的結果。

時間分析

1. 多複製一份而已，每個數字看過一次即可，$O(n)$
2. 因爲題目範圍是[1, n^3]，每個數字一直除 n 即可轉成 n 進位，最多只會除 3 次，所以時間是 $O(3n) = O(n)$
3. 對這 n 個 n 進位的數字套用 Radix Sort，每次排一位，最多只有三位，每次範圍是 1 ~ n，時間是 $O(3(n + n)) = O(n)$
4. n 個數字，原本頂多屬於 n 種不同的 group，套用 Bucket Sort $O(n+n) = O(n)$
5. 印出答案每個數字只要 $O(1)$，總共 n 個數字，$O(n)$

因此全部是 $O(n)$

Problem 6:

(1)

分別對 A 跟 B 做 sorting。

因爲題目並無規定是 bounded integer，所以無法用 linear time integer sorting。

time complexity：$O(n\log n) + O(n\log n) = O(n\log n)$

(2)

證明兩件事情：

i. 令 a_p 是 set A 裡面最大的 element，b_p 是 set B 裡面最大的 element。則存在一組 optimal payoff，其中有一項乘數爲 $a_p^{b_p}$ 。

ii. 令 set $A' = A - \{a_p\}$ and set $B' = B - \{b_p\}$。則原本 A 跟 B 的 optimal payoff 等於新的 A'跟 B'所求得的 optimal payoff 乘上 $a_p^{b_p}$ 。

如果上列兩項都成立，則可以套用 greedy，每次都取兩個 set 的最大 element 就可以了。

proof i: 假設有一個 optimal payoff *OPT_PAYOFF* 其中有一項乘數爲 $a_p^{b_q}$ ，則我們可以找到另一項乘數爲 $a_r^{b_p}$ ，因爲 a_p 跟 b_p 皆爲 A、B 中的最大值，所以 $a_r \le a_p$ 且 $b_q \le b_p$。將此兩項乘數的指數互換，則新的 payoff NEW_PAYOFF 的兩項乘數分別爲 $a_p^{b_p}$ ， $a_r^{b_q}$ 。

$$OPT\_PAYOFF - NEW\_PAYOFF$$

$$= \frac{OPT\_PAYOFF}{a_p^{b_q} a_r^{b_p}} \left(a_p^{b_q} a_r^{b_p} - a_p^{b_p} a_r^{b_q} \right)$$

$$= \frac{OPT\_PAYOFF}{a_p^{b_q} a_r^{b_p}} \left(a_p^{b_q} a_r^{b_q} \left(a_r^{b_p - b_q} - a_p^{b_p - b_q} \right) \right)$$

由於 $a_r \le a_p$ 且 $b_q \le b_p$ 上面的式子會≤ 0，因爲 optimal payoff 是最大的關係，所以新的 payoff 也會是一組 optimal payoff，得証。

proof ii: 由上面的証明得知，A、B 一定有組 optimal payoff 的乘數是 $a_p^{b_p}$ ，所以假設 A 跟 B 的 optimal payoff 是 $a_p^{b_p} \times pay^*$，如果 pay^*不是剩下的 sub problem 中最大的 payoff，那一定可以從 A'、B'中找到另外一組更大的，來替換掉原本的 pay^*。所以 A 跟 B 的 optimal payoff 一定會等於 $a_p^{b_p}$ × (A'跟 B'的 optimal payoff)，得証。

Problem 7: (10%)
請參考課本及講義 Section 15.4

Problem 8：

證明：必定存在一個 Optimal Schedule 每個工作之間都連著做

假如存在一個 Optimal Schedule 工作之間不是連著做(有休息)，那今天我把這些休息時間都拿掉，Schedule 順序不變，原本在 deadline 之前作完的一樣會作完，答案只有可能更好不可能更差，因此這新的全部連著做

的也是個 Optimal Schedule，故必定存在一個 Optimal Schedule 每個工作之間都連著做

因此由以上的証明接下來我們只考慮每個工作連著做的情況

證明：必定存在一個 Optimal Schedule 每個工作之間都連著做且在 deadline 之前作完的工作都位於 Schedule 的前方，其餘沒有在 deadline 之前作完的都在後方

假如有一個 Schedule 是有得到分數(profit)和沒得到分數(超過 deadline)的交錯，那我把沒得到分數的全部都排到這個 Schedule 尾巴，其餘有得到分數的都依照原本順序往前緊靠在一起，很明顯的，有得到分數的依然得的到分數，因爲只會往前移動，至於沒得到分數的放哪邊都不影響答案，因此這新的 Schedule 符合我們證明的要求。

因此由以上的証明接下來我們只考慮每個工作連著做的情況且有得到分數的都在前方連續，剩下的在後方。

證明：必定存在一個 Optimal Schedule 每個工作之間都連著做且在 deadline 之前作完的工作都位於 Schedule 的前方，其餘沒有在 deadline 之前作完的都在後方，且前方有得到分數的那些工作，是依照 deadline 的時間先後排序，先結束的在前方，後結束的在後方。

假如今天有一個 Schedule 前方得到分數的並不是依照上述 deadline 順序排序，那我可以把他調一調，以兩個有得到分數的工作來看，若 deadline 後結束的排前面先結束的排後面但還是兩個都有分數，那對調之後兩個一定還是得到分數，往前移動的很明顯會有分數，而往後移動的因爲原本的順序 deadline 比較早的排後面都能得到分數，那我 deadline 比較晚放到後面當然也能得到分數，因此我們可以不斷調整，調成我們要的順序，但是卻不影響到原本的分數，故證明成立。

根據以上三點證明我們知道，必定存在一個 Optimal Schedule 每個工作之間都連著做且在 deadline 之前作完的工作都位於 Schedule 的前方，其餘沒有在 deadline 之前作完的都在後方，且前方有得到分數的那些工作，是依照 deadline 的時間先後排序，先結束的在前方，後結束的在後方。

我們先把原本的工作依照 deadline 時間排序，並依照這順序重新把工作叫做 $a_1, a_2, \ldots, a_n$，並且原本的 t_j, p_j, d_j 也跟著這順序重新編號

定義 $score[i,j]$表示，只靠前面 $a_1, a_2, \ldots, a_i$ 個工作，排出一個 Optimal Schedule 其中前段有得到分數的工作，最後結尾的時間在 j

整理之後會得到下述 recurrence

$$score[i,j] = \max \begin{cases} 0 & \text{if } i = 0 \text{ and } j = 0 \\ score[i-1,j] & \text{if } i > 0 \\ score[i-1,j-t_i] + p_i & \text{if } i > 0 \text{ and } t_i \le j \le d_i \\ -\infty & \text{others} \end{cases}$$

因爲有上述三個證明的特性，因此一個新的工作 a_i 如果是屬於最後 Schedule 有得到分數的前半段，一定會接在之前 i-1 個工作有得到分數最棒排序的後面，反之如果這新的工作不屬於有得到分數的群組，前方有得到分數最好的排法會和 i-1 個一樣，而當這新的工作做完的時間已經超過 deadline 的話，他就沒有機會是前方提供分數的那群。

表格每一行只需要開 n^2，因爲每個工作連續，因此結束的時間不可能超過 n^2，由於最後我們不知道前方那群結尾是在哪個位置，因此最後我們對表格的最後一行，$score[n, i]$, $i = 0$ to n^2 全部掃描，找個最大的答案就是我們要的，要印出完整的 Schedule 我們只要從那格 Back Track 回去，把前方提供分數的工作找出來，剩下的工作隨便接在尾巴即可。

時間分析

一開始依據 deadline 從小到大 sort 套用 merge sort 時間是 $O(n\log n)$
表格總共要填 n^3 格，每格 $O(1)$，因此時間是 $O(n^3)$
最後找答案以及印答案時間不會超過 $O(n^2)$
因此總共是 $O(n^3)$

空間分析

如果不用 Back Track 只需要留兩行，也就是 $O(n^2)$，反之則整個留，$O(n^3)$

正確性

請參考 recurrence 以及前述三個證明，中途任何 sub problem 一定也都是 Optimal 解，不然整體不會是 Optimal

Problem 9

作業題，請參考之前的作業解答。

Bouns：

請參考課本 p.262 263

1.砍掉的那個 node 若是 leaf(沒有 child)則直接砍掉。
2.若是有一個 child 則砍掉後把下面整群往上接。
3.若是有兩個 child 則從左邊 subtree 選個最大的或是右邊 subtree 選個最小的 node 拿來補這個 node，而這個被拿去補而砍掉的 node 下面頂多只有一個或是沒有 child，因此套用上面 1 or 2 的作法即可。