P1

$$f_7 < f_1 < f_4 < f_{96} < f_{03} < f_2 < f_8 < f_5$$

P2

(1) ~~Algorithm~~

P2 (1) Algorithm

① Sort the weight from small to large. the sorted list is S_i, $0 \le i \le n-1$

② Let Height = 0, T = {∅}, ~~Sum_w = 0~~

for k = 0 → n-1

if ~~Sum_w~~ $\le S_k$

T.add(S_k); Sum_w = Sum_w + S_k.weight;

Height = Height + 1;

end if

end for

return Height

Time complexity: O(nlogn) + O(n) = O(nlogn)

(2) Proof,

Assume greedy is not optimal, and let's ~~see~~ G_0 G_1 ... G_k be the stones selected by Greedy algorithm

$OPT_0, OPT_1 \cdots OPT_m$ denotes the stones selected by optimal solution

Suppose that $OPT_0 = G_0$ ... $OPT_j = G_j$, for k, the largest value of j

Then we have $OPT_{j+1} > G_{j+1}$, However, since both OPT_{j+1} & G_{j+1} can hold the [0~j] previous stones, and G_{j+1} is lighter than OPT_{j+1} which is better for latter layers, therefore, we can replace OPT_{j+1} by G_{j+1} which also maintains the optimal ~~is~~ $OPT_{j+1} = G_{j+1}$, contradiction

∴ greedy is optimal

P3 (1) Algorithm

Sort the timetable from small start time to large, as set (A)

```
then; M = ∞ , L = {∅} , check_p = {∅}
  for  i = 0 : n-1
      if Ai_start ≤ M

          L.add(Ai) ;
          M = min(Ai_finish, M) ;

      else
          clear L, i
          check_p : add {M} ;
          M = ∞ ;
      end if
  end for
  return check_p
```

(2) proof. Assume Greedy is not the optimal solution, and let $G_0 \cdots G_k$ be the check point selected by our algorithm, $OPT_0 \cdots OPT_m$ be the check point selected by optimal solution, then assume $G_0 = OPT_0 \cdots G_j = OPT_j$ for the largest possible value of j.

Then assume that $r^{th} \sim n^{th}$ people need to be checked after $G_0 - G_j$ check points and the $r^{th} \sim (r+i)^{th}$ people has overlaped timetable.

Then for the Greedy algorithm, after G_{j+1} check point, we need to do Greedy $(r+i+1)^{th} \sim n^{th}$)

for optimal solution, after OPT_{j+1}, it will be optimal $(r+i+1-x)^{th} \sim n^{th}$)

which means the the optimal solution leaves some people which can be checked

$\Leftrightarrow$ we can replace OPT by Greedy, which will not affect the latter solution

$\therefore OPT_{j+1} = G_{j+1}$, contradiction

P4. Input: (Array, Card

Note: Judge( ) means use the equivalence tester

abandon means we don't consider those cards in the following test of this algorithm

Algorithm:

F (Array Card[], Int n)

C1 = card[on $\frac{n}{2}$-1];

C2 = Card[$\frac{n}{2}$ on n];

if n==2

k = Judge(C1, C2) # judge whether this two cards are same or not. if same, return 2 else: return 0

if k==2, mark C1, C2 are the same, return 2

else abandon C1, C2, return 0

endif

endif

k1 = F[C1, $\frac{n}{2}$]

k2 = F[C2, $\frac{n}{2}$]

if k1 == k2 , k=Judge(C1, C2) # only test once, because elements of C1, C2 are the same, respectively.

if k==2, make C1, C2 the same, return k1+k2; else abandon C1, C2, return 0, endif

endif

if k1 or k2 == 0, abandon C1, C2, return 0

else, k = Judge(C1, C2)

if k==2, mark C1, C2 the same, return k1+k2

else abandon the smaller one (C1, C2), return max(k1, k2)

endif

endif

end

This algorithm above can check which cards has the largest number then we can use this card to do another (n-1) test to check whether there exist $\frac{n}{2}$+1 this card. (compare with all other cards)

The Judge invocations are $O(\log_2 n \cdot (\frac{1}{2}n+1) \cdot 2) + O(n-1)$

$= O(n \log_2 n)$ for the worst case.

P5.

a) Line 6 means that it divides A into B and C, where B stores the a_i which has 1 in K^{th} bit, C stores the a_i which has 0 in K^{th} bit

Line 12 to 16 are the base case, means that if the 1^{th} bit of A are all 0 or all 1, then we always can find a number X_1 makes $X_1 \oplus A_{1^{th}\,bit} = 0$

∴ If B or C is empty we'll return 0

else return 1, obviously

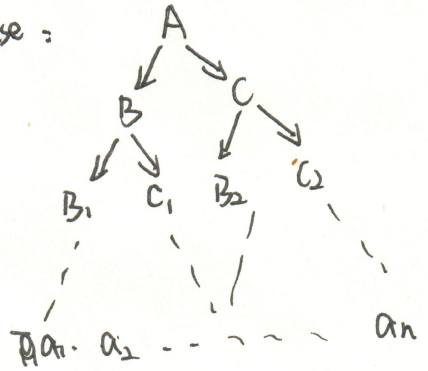
Line 18 to 27 are the recursion step. which means that:

If B and C are not empty, then the maximum XOR result must be 1 at K^{th} bit, this is why Line 26 add 1.

If B or C is empty, which means that there must exist X_K makes $X_K \oplus A_{K^{th}\,bit} = 0$ therefore we only need to consider the latter bits this is why we do recursion in Line 18, 21, and only compute $\min\{A \text{ xor } B, A \text{ xor } C\}$ in Line 25

b) The time complexity is $O(K \cdot N)$

because:



In each layer, since A=B+C therefore every layer has n components in all $(a_1, a_2 \cdots a_n)$

And in each function, we have only one loop, therefore in each layer, we will access all the a_j $\Leftrightarrow O(n)$

and the depth of this tree is K obviously (from K to 1)

∴ $O(n) \cdot O(K) = O(KN)$

P6. Algorithm.

```
OPT(A,B,C,i,j,k)  # i=j=k=0 at beginning
    if A_i ≠ C_k and B_j ≠ C_k
    endif  return k=0
    if A_i = C_k and B_j ≠ C_k:
        k = OPT(A,B,C,i+1,j,k+1)
    endif
    if A_i ≠ C_k and B_j = C_k:
        k = OPT(A,B,C,i,j+1,k+1)
    endif
    if A_i = C_k and B_j = C_k:
        k = max{ OPT(A,B,C,i+1,j,k+1), OPT(A,B,C,i,j+1,k+1) }
    endif
    return k
end
Let k = OPT(A,B,C,0,0,0)
    then if k = min
            return true
         else
            return false.
         endif
```

Note, since the range of i,j,k should be $[0, m-1]$, $[0, n-1]$ and $[0, min-1]$ respectively

So the algorithm before will error due to the range,

Since this error is not vital to the algorithm, I do not fix them in the algorithm, but it needs to add some limitation to realize this algorithm for the simplicity

P7. Algorithm. . This problem is a RMQ problem, we use ST table to solve it.

① pre-processing the range, limit the input (l, r) into [1, n] ; (constant time)

② build ST-table.

define: $f[i][j]$ denotes the max value start from i^{th}, end with $(i+2^j-1)^{th}$

then every $f[i][j]$ can be compute by:

$$f[i][j] = \max(f[i][j-1], f[i+2^{j-1}][j-1])$$

* use this function to build a table

which stores the value of $f[0][1]$, $f[1][1]$, $f[2][1]$ ....

$f[0][2]$, $f[1][2]$ ....

⋮

$f[0][\log n]$, $f[1][\log n]$ ....

the time complexity is $O(\log n] + O(\log(n-1)) + \cdots + O(\log 1)$

$= O(n \log n)$

③ search:

$L = r - l + 1$ is the length of this interval let $k = \log L$ down rounding

divide L into two parts:

answer = $\max\{ f[l][k], f[r-2^k+1][k] \}$

is the right answer, time complexity is $O(p)$ from all questions

∴ total complexity $O(n \log n + q)$