ZIO 2027 Editorial

Samik Goyal

Accompanying OCaml and Python implementations of these solutions can be found in the appendices. No AI was used to write this editorial, though AI assisted in writing the code.

Answer Key

P1

A) 18
B) 46
C) 152

P2

A) 3
B) 4
C) 10

P3

A) 8
B) 40320
C) 60480

P4

A) 8
B) 36
C) 4320

Problem 1

Observation 1. If we have some set S of people that go in the lift together, then the amount of time the lift spends on taking them from ground floor and coming back is $2 \cdot F_{max}$ where F_{max} is the max value of F_i over all the people $i \in S$ in the group.

Observation 2. Given a solution, we can order the individual trips in any way.

Corollary 1. Given a set of people, there is always an optimal solution where the person with the max F_i goes in the first trip.

Observation 3. Given a set of people, the first trip will have the C people with the highest F_i. If less than C people are left, everyone is taken.

This can be proven by using an exchange argument. If you don't take the C people with the highest F_i in an optimal solution then by swapping you can never have a worse solution.

Final Greedy Solution. We can apply *Observation 3* recursively on a set of people to yield a greedy solution. Let us sort the people in reverse order of F_i. Then the first trip takes the first C people. Second trip takes the next C and so on.

Data Observation 1. Notice that even though the problem statement does not specify that F_i will be non-decreasing, the data has that property. We can use this to calculate the answer without sorting.

Problem 2

Observation 1. Each switch will be pressed at most once.

This can be seen from the fact that pressing the same button twice is the same as not pressing it at all.

Observation 2. Each switch press is independent. That is, the state of lamps after a series of switch presses is based only on which switches were pressed and the number of times they were pressed, and not on the order in which they were pressed.

Corollary 1. There exists an optimal solution with the following structure: we go to each lamp from 1 to N in that order, and then either press the large switch once or do not. Let us call the state the lamps are in now as the intermediate. At the end, we press the small switch on the lamps still left on.

Observation 3. A large switch of lamp i only affects lamps j with $j \geq i$.

Corollary 2. When using the structure specified in *Corollary 1*, once we have decided whether or not to press the large switch of lamp i, the intermediate state of lamp i is decided.

Observation 4. The intermediate state of lamp i depends on whether the total number of large switches j that were pressed with $j \leq i$ was odd or even.

Final Solution. We go to each lamp from 1 to N in that order. Then we either press the large switch or we don't. After this if the lamp is still left switched on, we use the small switch. Now we model this with a DP state.

DP State. Define $DPodd_i$ to be the amount of minimum switch presses if we have already made decisions (large and small switch) for lamps 1 to i, provided that the total number of large switch presses is odd. Define $DPeven_i$ similarly.

DP Transitions. The transitions are the following. They can be proven using some case work.

$$DPodd_i = \begin{cases} min(DPeven_{i-1} + 2, DPodd_{i-1} + 1) & \text{if } S_i = 0 \\ min(DPeven_{i-1} + 1, DPodd_{i-1}) & \text{if } S_i = 1 \end{cases}$$

$$DPeven_i = \begin{cases} min(DPeven_{i-1}, DPodd_{i-1} + 1) & \text{if } S_i = 0 \\ min(DPeven_{i-1} + 1, DPodd_{i-1} + 2) & \text{if } S_i = 1 \end{cases}$$

DP Base Case. The base cases are the following:

$$DPodd_{base} = \infty$$
$$DPeven_{base} = 0$$

Problem 3

Observation 1. $f_{A_i} \leq i$ for all i.

This can be trivially seen as if f_{A_i} was greater than i then there is no case where car A_i would have parked in i.

Observation 2. For all i and j such that $j < i$ and $A_j > A_i$, $f_{A_i} > j$.

This can be seen as if $f_{A_i} \leq j$, then car A_i would have parked in spot j as j would've been free when it came. j would have been free as $A_j > A_i$ and cars come in order of their number.

Observation 3. All f_i can be chosen independently as long as the final spaces where the cars park remain the same.

Final Solution. We have created an upper and lower bound for all f_{A_i}. The non-inclusive lower bound for f_{A_i} would be the largest index smaller than i with $A_j > A_i$. If such an index doesn't exist, it will be 0. The upper bound will be i itself. f_{A_i} can take any values in this range so we simply take the product of size of ranges across all f_{A_i} for our solution.

Problem 4

Note. My solution to this problem is very mathematical in nature. However, it is quite elegant, and easy to compute on paper.

Observation 1. The first traversal over any edge would be in the downwards direction (call this entering a node). That is, away from the root which is 1. The second would go towards the root (call this exiting a node). Third would be away, and so on.

Observation 2. Each edge will be traversed at most twice.

We can prove this easily using contradiction. Let there exist a tour where some edge going from some node S to its parent P is traversed thrice or more. Focus on the first three occurrences. This means our tour will be of the form $\underbrace{\ldots, P}_{A}, \underbrace{S, \ldots, S}_{B}, \underbrace{P, \ldots, P}_{C}, \underbrace{S, \ldots}_{D}$. A and C cover sections outside the subtree of S while B covers the subtree. Now consider the tour $\underbrace{\ldots, P}_{A}\underbrace{\ldots, P}_{C}, \underbrace{S, \ldots, S}_{B}\underbrace{\ldots}_{D}$. It covers the same nodes with 2 fewer steps. Hence we have found a better solution.

Corollary 1. Once we enter a node, we tour over its full subtree before exiting.

Observation 3. We will always exit a node unless the tour ends at a node which is in its subtree.

Observation 4. Our total length of tour will be $2 \cdot (N-1) - Depth_v$ where v is the node where we end.

Corollary 2. Our tour will end at a node with the maximum depth. This will be a leaf.

Observation 5. Now, consider shortest closed tours (ones which end at root). Finding our answer is equivalent to finding the total number of closed tours where the last visited leaf was one with maximum depth.

Corollary 3. Since all closed tours can be reversed, our answer is equivalent to finding the total number of closed tours where the *first* visited leaf was one with maximum depth.

Definition 1. Let $child_i$ be the number of children of node i.

Observation 6. Now to calculate the total number of closed tours. We can view this as, at each node we must figure out an ordering of the children to enumerate them in. We do this recursively for every node. This makes our total number of closed tours:

$$\prod_i child_i!$$

Observation 7. The probability of the first leaf visited in a random closed tour being v is $\prod_{i \text{ is a strict ancestor of } v} \frac{1}{child_i}$.

Final Solution. We calculate the total number of closed tours from *Observation 6* and the sum of probabilities (as defined in *Observation 7*) over all leaves with maximum depth. Then we multiply them together to give the final answer.

A OCaml Code and Tests

Problem 1

Code. The following code implements the greedy strategy in OCaml.

```
let rec helper_solve c l = match l with
| [] -> 0
| head::tail -> 2 * head + helper_solve c (List.drop c l);;

let solve c l = helper_solve c (List.rev (List.sort compare l));;
```

Subtasks. Here are the test cases in OCaml.

```
solve 2 [1; 3; 4; 6];;
solve 3 [1; 2; 4; 4; 5; 7; 8; 9; 12];;
solve 4 [2; 3; 4; 5; 6; 8; 9; 11; 13; 14; 16; 17; 19; 21; 22; 25; 27; 30];;
```

Problem 2

Code. The following code implements the DP in OCaml.

```
let rec helper_solve dpodd dpeven = function
| [] -> min dpodd dpeven
| head::tail when head = 0 -> helper_solve ( min (dpeven+2) (dpodd+1) ) ( min dpeven (
    dpodd+1) ) tail
| head::tail when head = 1 -> helper_solve ( min (dpeven+1) dpodd ) ( min (dpeven+1) (
    dpodd+2) ) tail
| _ -> failwith "Invalid input";;

let solve l = helper_solve 1000000000 0 l;;
```

Subtasks. Here are the test cases in OCaml.

```
solve [1; 1; 1; 0; 0; 1];;
solve [1; 0; 1; 1; 0; 1; 0; 1; 1; 1];;
solve [0; 1; 1; 0; 1; 0; 1; 0; 1; 1; 0; 0; 0; 1; 1; 0; 1; 0; 1; 1; 0; 1; 1; 0];;
```

Problem 3

Code. The following code implements the solution in OCaml. Note that it reverses the list to make the logic easier to code.

```
let rec helper_find_prefix_less_than limit = function
| head::tail when head<=limit -> 1 + helper_find_prefix_less_than limit tail
| _ -> 0;;

let rec helper_solve l = match l with
| [] -> 1
| head::tail -> (helper_find_prefix_less_than head l) * (helper_solve tail);;

let solve l = helper_solve (List.rev l);;
```

Subtasks. Here are the test cases in OCaml.

```
solve [3; 1; 2; 4];;
solve [1; 2; 3; 4; 5; 6; 7; 8];;
solve [4; 1; 2; 7; 3; 15; 5; 9; 6; 8; 11; 10; 13; 12; 14];;
```

Problem 4

Code. The following code implements the solution in OCaml. It is a lot more complex than the other problems.

```
type tree = Node of int * tree list;;

let rec helper_build x adj =
  let rec get_children = function
    | [] -> []
    | (curr, par)::tail when par = x -> helper_build curr adj :: get_children tail
    | (curr, par)::tail -> get_children tail
  in
  Node (x, get_children adj);;

let helper_generate_tree p = helper_build 1 (List.mapi (fun i par -> (i+1, par)) p);;

let helper_merge (p, maxdep) (pdash, childmaxdep) =
  let newmaxdep = childmaxdep+1 in
    if newmaxdep > maxdep then (pdash, newmaxdep)
    else if newmaxdep < maxdep then (p, maxdep)
    else (p +. pdash, maxdep);;

let rec helper_solve = function
  | Node (_, []) -> (1.0, 0)
  | Node (_, l) ->
    let n = float_of_int (List.length l) in
      let p, maxdep = List.fold_left helper_merge (0.0, 0) (List.map helper_solve l) in
        (p /. n, maxdep);;

let rec fact = function
| 0 -> 1
| n -> n * fact (n-1);;

let rec helper_all_tours (Node(v, l)) =
  List.fold_left ( *. ) (float_of_int (fact (List.length l))) (List.map
      helper_all_tours l);;

let solve p =
  let prob, _ = helper_solve (helper_generate_tree p) in
    int_of_float (prob *. helper_all_tours (helper_generate_tree p) +. 0.5);;
```

Subtasks. Here are the test cases in OCaml.

```
solve [-1; 1; 1; 1; 2; 2; 4];;
solve [-1; 1; 2; 2; 3; 2; 2; 7; 6; 6; 1];;
solve [-1; 1; 1; 1; 2; 2; 2; 2; 3; 3; 4; 5; 5; 5; 9; 9; 9; 12; 15; 11; 20];;
```

B Python Code and Tests

Problem 1

Code. The following code implements the greedy strategy in Python.

```
def solve(c, l):
    l = sorted(l, reverse=True)
    return sum(2 * x for x in l[::c])
```

Subtasks. Here are the test cases in Python.

```
solve(2, [1, 3, 4, 6])
solve(3, [1, 2, 4, 4, 5, 7, 8, 9, 12])
solve(4, [2, 3, 4, 5, 6, 8, 9, 11, 13, 14, 16, 17, 19, 21, 22, 25, 27, 30])
```

Problem 2

Code. The following code implements the DP in Python.

```
def solve(l):
    dp_odd, dp_even = 10**9, 0
    for s in l:
        if s == 0:
            dp_odd, dp_even = min(dp_even + 2, dp_odd + 1), min(dp_even, dp_odd + 1)
        elif s == 1:
            dp_odd, dp_even = min(dp_even + 1, dp_odd), min(dp_even + 1, dp_odd + 2)
        else:
            raise ValueError("Invalid␣input")
    return min(dp_odd, dp_even)
```

Subtasks. Here are the test cases in Python.

```
solve([1, 1, 1, 0, 0, 1])
solve([1, 0, 1, 1, 0, 1, 0, 1, 1, 1])
solve([0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 0])
```

Problem 3

Code. The following code implements the solution in Python.

```
def solve(l):
    ans = 1
    for i, val in enumerate(l):
        count = 1
        j = i - 1
        while j >= 0 and l[j] <= val:
            count += 1
            j -= 1
        ans *= count
    return ans
```

Subtasks. Here are the test cases in Python.

```
solve([3, 1, 2, 4])
solve([1, 2, 3, 4, 5, 6, 7, 8])
solve([4, 1, 2, 7, 3, 15, 5, 9, 6, 8, 11, 10, 13, 12, 14])
```

Problem 4

Code. The following code implements the solution in Python. It is simpler than the OCaml version.

```
import math

def solve(p):
    adj = [[] for _ in range(len(p) + 1)]
    for i, par in enumerate(p, start=1):
```

```
        if par != -1:
            adj[par].append(i)

    def dfs(u):
        children = adj[u]
        if not children:
            return 1.0, 0
        max_dep = 0
        prob_sum = 0.0
        for v in children:
            p_child, dep_child = dfs(v)
            new_dep = dep_child + 1
            if new_dep > max_dep:
                max_dep = new_dep
                prob_sum = p_child
            elif new_dep == max_dep:
                prob_sum += p_child
        return prob_sum / len(children), max_dep

    prob, _ = dfs(1)
    all_tours = math.prod(math.factorial(len(adj[u])) for u in range(1, len(p) + 1))
    return round(prob * all_tours)
```

Subtasks. Here are the test cases in Python.

```
solve([-1, 1, 1, 1, 2, 2, 4])
solve([-1, 1, 2, 2, 3, 2, 2, 7, 6, 6, 1])
solve([-1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 4, 5, 5, 5, 9, 9, 9, 12, 15, 11, 20])
```