# Optimization of Cloud Cluster Cost

time limit per test: 5 seconds
total time limit: 2 minutes
memory limit per test: 512 megabytes
input: standard input
output: standard output

## Background

The execution of containerized applications in managed container orchestration services offered by cloud providers is becoming mainstream. A customer of such service creates a **cluster** consisting of one or more **nodes** which are used for running customer's containers. Containers are submitted for execution in the form of **pods** by the customer and are assigned to cluster nodes by the **cluster scheduler**. Cluster nodes usually correspond to virtual machines (VM), so possible node types (e.g. amount of available CPUs and memory per node) are limited by a set of VM instance types (called **flavors**) supported by the cloud provider. Each flavor has price per second, and the **cluster cost** is calculated based on the price of the used nodes and their running time.

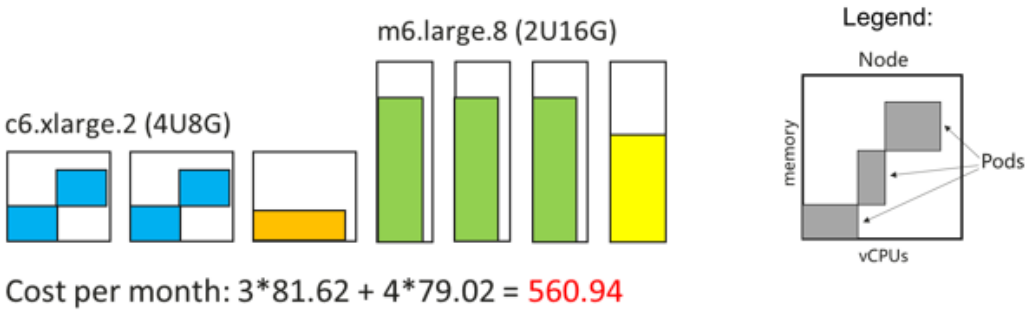
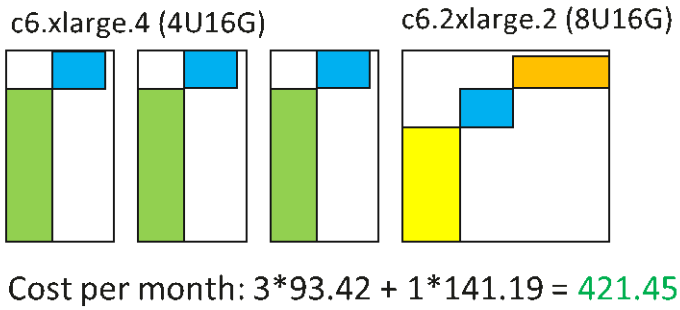



Fig 1. Example of cluster with 7 nodes running 9 pods.

Currently customers must manually decide which flavors and how many to use as the cluster nodes. The choice of cluster nodes has direct impact on the customer's costs, and suboptimal decisions may result in paying for unused resources and substantial cost overruns. In example below the cluster costs for running the same set of pods can be reduced by 25% by using different node flavors.




However, this example considers only some fixed set of pods. In practice, the set of running pods is changing in time – new pods are created and old pods are deleted based on current customer's demands. For example, more pods are created during the rush hours, while less pods are running during the night. Some pods may be submitted only time after time, e.g. to run some data processing jobs. In such cases, the cost-efficient cluster configuration for one time moment may not

be efficient for other time moments. For example, different number of nodes or even node flavors may be needed for different periods in order to save costs.

Manually tuning the cluster configuration to achieve costs savings puts too much burden on the customer. The goal of this challenge is to develop an **automated cluster management solution** which will remove this burden by making decisions about which flavors to use, when and how many on behalf of the customer. To be more specific, the goal is to devise the following **algorithms** comprising this solution such as to **minimize the total cost** of running customer cluster for some time period while satisfying all customer's requests. The first part of solution is the **pod scheduling algorithm**, which is used to decide on which node to place a pod. The second part of solution is the **node scaling algorithm** which is used to decide which nodes to add to the cluster to accommodate the unscheduled pods. The main challenge is that in production these algorithms must work in **online** setting by processing customer requests one by one, without knowing which requests will come next and when.

## Description

Your task is to develop the main logic of a service for managing a customer's cluster. Initially the cluster is empty - there are no nodes and no running pods. The service is given the set of possible node flavors, for each flavor its resource capacity (CPU and memory) and price per second are specified.

The service must process two types of customer requests:

1. **Create** pods request (specifies the number of pods to be created, and their resource requests for CPU and memory)
2. **Delete** pods request (specifies a set of previously created pods to be deleted)

Requests arrive online, and each request must be processed completely before processing the next one as follows.

**Create** pods request must be processed by immediately scheduling all pods, i.e. each pod from the request must be placed on some cluster node. When a pod is placed it consumes the requested resources on the node. The total amount of consumed resources by all pods on a node must not exceed the node capacities both for CPU and memory. If some pods cannot be scheduled due to lack of resources, new nodes must be created until all pods are successfully scheduled. The sizes of nodes must be chosen only from the specified set of flavors.

**Delete** pods request must be processed by immediately releasing all resources consumed by the specified pods on respective nodes. As soon as some node becomes idle (not runs any pod) it is automatically deleted from the cluster. Make sure your solution does not attempt to reuse such nodes after deletion.

Your solution must implement the logic of described service by interacting with a grading program. The format of the interaction is described in the separate section below.

The solution is graded based on the **total cluster cost** calculated as follows. The cost of each cluster node is calculated by multiplying the node flavor's price and node running time (time between the node creation and its deletion). The total cluster cost is simply the sum of the costs of all nodes created by the solution during processing of requests from the input file. The lower the achieved cluster cost the better.

# Interaction

The interaction with your program is organized via standard input and output streams.

Upon startup, your program is provided with VM flavor data in the following format.

The first line contains one integer $F$ ($1 \leq F \leq 50$) , denoting the number of flavors.

The next $F$ lines , each line contains two integers and one float: *available CPU, available memory*, and *price per second* of the corresponding flavor. *CPU* is given as an integer number of abstract CPU units. The exact nature of *CPU* units is irrelevant to the problem. *Memory* is given as an integer number of megabytes.

It is guaranteed that all *CPU* values in the input are between $100$ and $51200$. All *memory* values are between $128$ and $2000000$. All *Price per second* is a floating-point numbers between $0.005$ and $20$ given with at most $4$ digits after decimal point.

Then the program starts to receive and process a sequence of requests.

The total number of requests is between $2$ and $20001$.

Each request begins with a line containing request timestamp (integer number of seconds since some time point, between $0$ and $10000000$), a string that describes request type (**CREATE** for pod creation request, **DELETE** for pod deletion request, **END** for input end), and a positive integer $S$ -- the number of pods in this request. It is guaranteed that timestamps are strictly increasing throughout the requests, and for creation requests $S \leq 100$.

1. If the request type is pod **creation**, the next $S$ lines describe the corresponding pods. Each of these lines contains three integers.

   (1) The first integer is *pod id*. It is guaranteed that all pods in one testcase are numbered with consecutive integers starting from $1$ in order of their appearance.

   (2) The second integer is required *CPU*, an integer number of *CPU* units.

   (3) The third integer is required memory, an integer number of *megabytes*.
2. If the request type is pod **deletion**, the next line contains $S$ distinct integers: ids of deleted pods. It is guaranteed that these pods exist at the time of the deletion request.
3. If the request type is input **end**, then $S = 0$ and your program must terminate with exit code $0$ upon receiving this request. Otherwise it may be judged incorrectly.

It is guaranteed that all pods will be deleted before the end of the input, and no more than $10000$ pods are created during one interaction.

For each pod **creation** request your program has to output two lines, otherwise it won't receive subsequent requests:

1. The first line contains one integer $C (0 \leq C \leq 100)$ denoting the number of flavors used for the new nodes. then contains $C$ integers $id_i (1 \leq id_i \leq F)$, denoting chosen flavors.
2. The second line contains $S$ integers, denoting the indexes of the nodes used to accommodate the corresponding pods from the request, in the same order. Note that the order of these numbers must be consistent with the order of pods in the input, and the nodes must be indexed from $1$ according to the order of their appearance in your output.

**After printing, do not forget to flush the output.** To do this, use `fflush(stdout)` or `cout.flush()` in C++,`System.out.flush()` in Java, or `stdout.flush()` in Python.

## Example

| Input | Output |
| --- | --- |
| 1<br>200 512 0.5<br>0 CREATE 4<br>1 100 128<br>2 100 128<br>3 100 128<br>4 200 256<br>1 DELETE 1<br>4<br>10 CREATE 1<br>5 100 128<br>11 DELETE 4<br>5 1 2 3<br>12 END 0 | 3 1 1 1<br>1 1 2 3<br>0<br>2 |

In the example above there are single flavor and four requests. The first request creates $4$ pods at timestamp $0$, the second request deletes pod number $4$ at timestamp $1$, the third request creates another pod at timestamp $10$, and the last request deletes all remaining pods. The solution creates $3$ nodes to accommodate pods from the first request: pods $1$ and $2$ are placed on node $1$, pod $3$ is placed on node $2$, and pod $4$ is placed on node $3$. On the second request the solution deletes pod $4$, freeing resources on node $3$. Since node $3$ becomes empty, it is instantly deleted. On the third request the solution places pod $5$ on node $2$ and does not create any new nodes. On the last request the solution deletes all pods and nodes. The solution used $3$ nodes, two of which were active for $11$ seconds, and the third was active for $1$ second. Therefore the total cost is $2 \times 0.5 \times 11 + 0.5 \times 1 = 11.5$.

## Evaluation

Your solutions will be evaluated by running the program on multiple test cases. For each test case a separate interaction in performed and the solution score is computed as $1000 \times (1 + \frac{baseline - cost}{baseline})$ .where *cost* is the cluster cost achieved by the solution and *baseline* is the cluster cost achieved by a simple baseline solution, which is provided to the participants. The overall score is computed as the sum of scores for each test case. The winner solution is the one achieved the highest overall score.

If your solution **violates time** or **memory limit** on some test case or **the output** is **invalid**, the score on this particular test case is **0**.