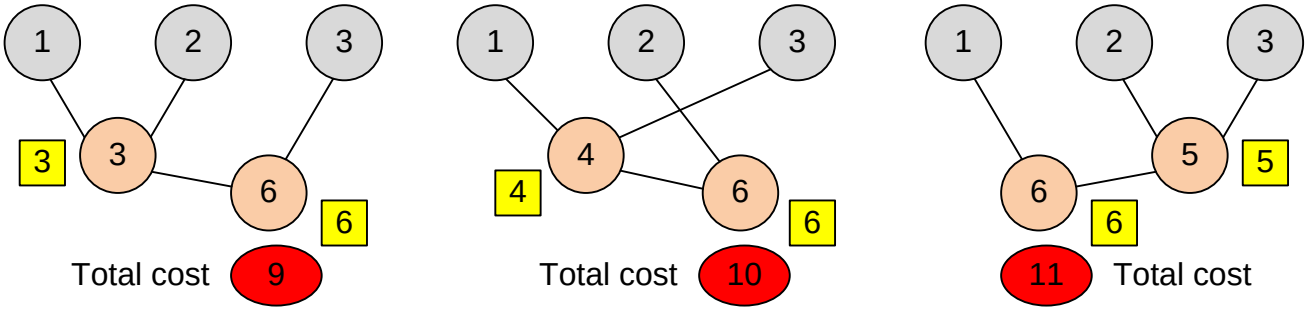
1228. Add All

The cost of adding two numbers equals to their sum. For example to add 1 and 10 cost 11. The cost of addition 1 and 2 is 3. We can add numbers in several ways:

- 1 + 2 = 3 (cost = 3), 3 + 3 = 6 (cost = 6), Total = 9
- 1 + 3 = 4 (cost = 4), 2 + 4 = 6 (cost = 6), Total = 10
- 2 + 3 = 5 (cost = 5), 1 + 5 = 6 (cost = 6), Total = 11

We hope you understood the task. You must add all numbers so that the total cost of summation will be the smallest.



Input. First line contains positive integer n ($2 \le n \le 10^5$). Second line contains n nonnegative integers, each no more than 10^5.

Output. Print the minimum total cost of summation.

Sample input	**Sample output**
4 1 2 3 4	19

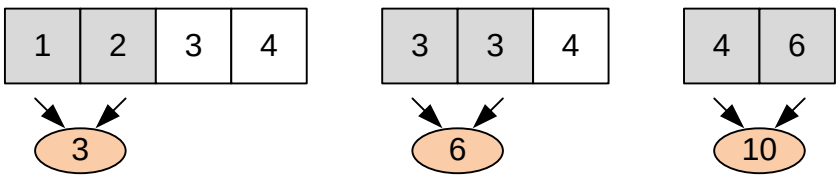
Algorithm analysis

Add the smallest two numbers each time. Then the total cost of summation for all n integers will be the minimum. Since numbers can be repeated, will store them in a multiset.

Example

To minimize the cost of addition, add the numbers in the following order:

1. Add 1 and 2, sum is 3. Cost of addition is 3.
2. Add 3 and 3, sum is 6. Cost of addition is 6.
3. Add 4 and 6, sum is 10. Cost of addition is 10.



Total cost of summation is 3 + 6 + 10 = 19.

Algorithm realization

Store the input numbers in the multiset *s* (numbers can be repeated). The two smallest numbers are always at the beginning of the multiset.

```
multiset<long long> s;
```

Read the amount of numbers *n*. Read the sequence of terms and push them into the multiset s.

```
scanf("%lld",&n);
s.clear();
for(i = 0; i < n; i++)
  scanf("%lld",&num),s.insert(num);
```

Accumulate the cost of additions in the variable *res*. Until there is only one number left (the size of the multiset *s* is greater than 1), add the two smallest numbers and insert their sum into *s*. The cost of adding numbers *a* and *b* is *a* + *b*.

```
res = 0;
while(s.size() > 1)
{
  a = *s.begin(); s.erase(s.begin());
  b = *s.begin(); s.erase(s.begin());
  s.insert(a + b);
  res += a + b;
}
```

When there is only one number left in the multiset, print the answer – the value of the *res* variable.

```
printf("%lld\n",res);
```

Algorithm realization – priority queue

Declare a priority queue *pq*, at the beginning of which there is a smallest element.

```
priority_queue<long long, vector<long long>, greater<long long> > pq;
```

Read the amount of numbers *n*. Read a sequence of terms and insert them into priority queue *pq*.

```
scanf("%lld",&n);
for(res = i = 0; i < n; i++)
  scanf("%lld",&num), pq.push(num);
```

Accumulate the cost of additions in the variable *res*. Until there is one number left (the size of the queue *pq* is greater than 1), add the two smallest numbers and insert their sum into *pq*. The cost of adding numbers *a* and *b* is *a* + *b*.

```
while(pq.size() > 1)
{
  a = pq.top(); pq.pop();
  b = pq.top(); pq.pop();
```

```
    pq.push(a + b);
    res += a + b;
}
```

When there is only one number left in the queue, print the answer – the value of the variable *res*.

```
printf("%lld\n",res);
```

7444. Graph game

Given graph, with 2*n* vertices and *m* edges. On every vertex and edge written an integer number. Serik and Zhomart were bored and invented the game on graph. The rules of this game are the following:

- Serik starts the game and they alternate turns.
- There are exactly *n* turns for each player.
- In every turn Player must choose a non-chosen vertex.
- The score of the player is the sum of numbers written in his chosen vertices, plus the sum of numbers written in edge, where both vertices of the edge are chosen by him.
- Every player tries to maximize the difference between his and opponent's score.
- Of course, Serik and Zhomart are very smart.

Input. The first line contains integers n, m ($1 \le n, m \le 10^5$).

The second line contains integers $a_1, a_2, \ldots, a_{2n}$ ($1 \le a_i \le 1000$) – numbers on vertices.

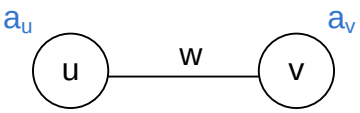
Next m lines contain three integer numbers u, v, w ($1 \le u, v \le n$, $1 \le w \le 1000$) – vertices u and v are connected, w is number on this edge. Only restriction for graph is: no loop edge.

Output. Print the difference between Serik's score and Zhomart's score.

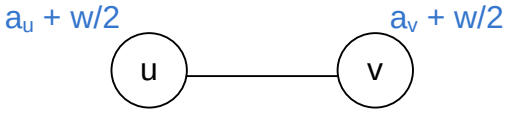
Sample input	Sample output
3 3 2 3 2 2 3 1 6 1 3 4 3 2 1 2 1	1

Algorithm analysis

Let the original graph contains an edge (u, v) of weight w, and the numbers a_u and a_v are written on the vertices:



Construct a new graph in which each such edge will be replaced by



We transferred the edge weight to the vertex weights. If Serik and Zhomart start to play on the new graph, then the difference between the scores for the optimal game will be the same as on the original graph. Indeed, if vertex u is chosen by one of the guys, and vertex v by the other, then the difference between the number of received points will remain the same:

$$(a_v + w / 2) - (a_u + w / 2) = a_v - a_u$$

If both vertices are chosen by one of the game participants, then he will receive

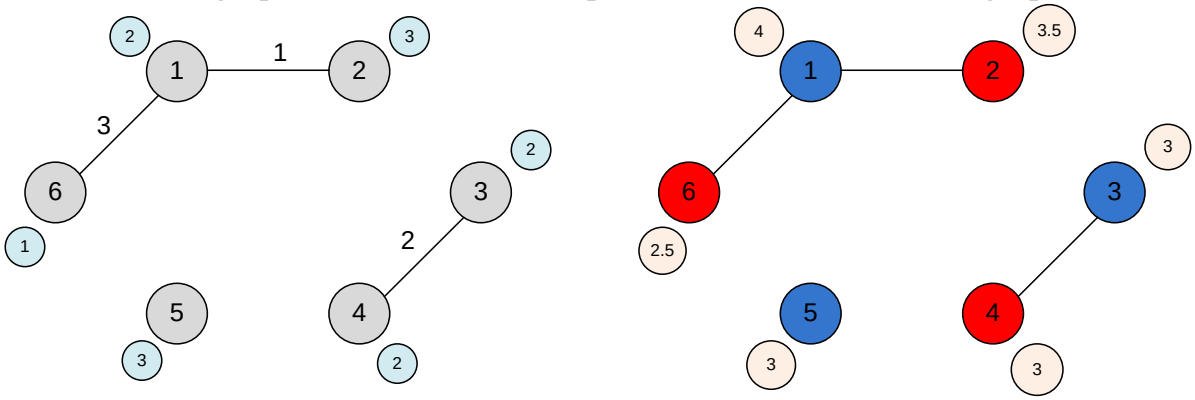
$$(a_v + w / 2) + (a_u + w / 2) = a_v + a_u + w$$

points. That is, he will receive the numbers assigned to the vertices on the original graph plus the weight of the edge.

The optimal play on the new graph is greedy. Each of the participants on his turn must choose a vertex with the maximum value assigned to it.

Example

Consider the graph shown in the example. Let's construct a new graph for it.



The first player will choose the vertices: 1, 3, 5 with weight 4 + 3 + 3 = 10.
The second player will choose the vertices: 2, 4, 6 with weight 3.5 + 3 + 2.5 = 9.
The difference between Serik's score and Zhomart's score is 10 – 9 = 1.

Algorithm realization

Store the numbers at the vertices of the graph in the array *a*.

```
#define MAX 200001
double a[MAX];
```

Read the input data.

```
scanf("%d %d", &n, &m);
```

```
n += n;
for (i = 1; i <= n; i++)
  scanf("%lf", &a[i]);
```

Construct a new graph.

```
for (i = 1; i <= m; i++)
{
  scanf("%d %d %d", &u, &v, &w);
  a[u] += w / 2.0;
  a[v] += w / 2.0;
}
```

Sort the array *a* in descending order.

```
sort(a + 1, a + n + 1, greater<double>());
```

Simulate the game – at each move the participant chooses the vertex with the maximum value.

```
double res = 0;
for (i = 1; i <= n; i++)
  if (i & 1) res += a[i];
  else res -= a[i];
```

Print the answer.

```
printf("%lld\n", (long long)res);
```

1591. Shoemaker problem

Shoemaker has *n* jobs (orders from customers) which he must make. Shoemaker can work on only one job in each day. For each *i*-th job, it is known the integer T_i, the time in days it takes the shoemaker to finish the job. For each day before finishing the *i*-th job, shoemaker must pay a fine of S_i cents. Your task is to help the shoemaker, writing a program to find the sequence of jobs with minimal total fine.

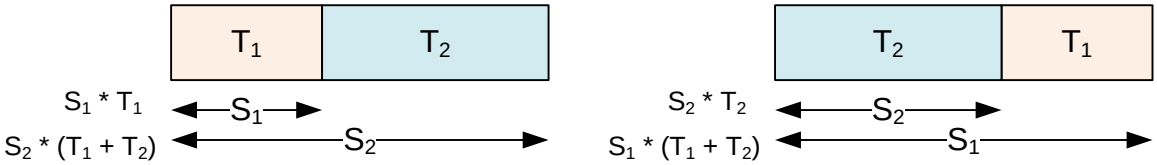
Input. Consists of multiple test cases. The first line of each test case contains the number of jobs *n* ($1 \le n \le 1000$), after which *n* lines contain the values of T_i ($1 \le T_i \le 1000$) and S_i ($1 \le S_i \le 10000$).

Output. For each test case print in a separate line the sequence of jobs with minimal fine. If multiple solutions are possible, print the lexicographically first.

Sample input	**Sample output**
4 3 4 1 1000 2 2	2 1 3 4

Algorithm analysis

Consider two jobs with characteristics T_1, S_1 and T_2, S_2.



If the first job is performed first, and then the second one, the fine is

$$S_1 * T_1 + S_2 * (T_1 + T_2)$$

If the second job is performed first, and then the first one, the fine is

$$S_2 * T_2 + S_1 * (T_1 + T_2)$$

Consider the condition when the fine for performing the jobs in order 1, 2 is better than when performing the jobs in order 2, 1:

$$S_1 * T_1 + S_2 * (T_1 + T_2) < S_2 * T_2 + S_1 * (T_1 + T_2)$$

Let's open the brackets and simplify the expression:

$$S_2 * T_1 < S_1 * T_2$$

Or the same as

$$\frac{T_1}{S_1} < \frac{T_2}{S_2}.$$

Now let we have *n* jobs. If there are *i*-th and *j*-th jobs for which $T_i / S_i > T_j / S_j$, then by swapping them in the sequence of execution, we will reduce the total amount of the fine. Thus, to minimize the fine, the jobs should be sorted by non-decreasing ratio of the time of their execution to the amount of the fine.

In case of equality of the ratio ($T_i / S_i = T_j / S_j$), the jobs should be sorted in ascending order of their numbers.

Example

Sort the jobs according to the non-decreasing ratio of their execution time to the amount of the fine:

job number	2	1	3	4
T_i	1	3	2	5
S_i	1000	4	2	5

$$\frac{1}{1000} \le \frac{3}{4} \le \frac{2}{2} \le \frac{5}{5}$$

We obtain, respectively, the optimal order of performance of the jobs indicated in the sample. The third and the fourth jobs have the same ratio (2/2 = 5/5), so we arrange them in ascending order of job numbers.

Algorithm realization

Information about jobs is stored in the array *jobs*, which elements are vectors of length 3. After reading the data, jobs[*i*][0] contains the execution time of the *i*-th job T_i, jobs[*i*][1] contains the value of the penalty S_i, and jobs[*i*][2] contains job number *i*.

```
vector<int> j(3,0);
vector<vector<int> > jobs;
```

Sorting function. Comparison $\frac{a[0]}{b[0]} < \frac{a[1]}{b[1]}$ is equivalent to a[0] * b[1] < b[0] * a[1]. If the ratios a[0] / b[0] and a[1] / b[1] are the same, then job with a lower number should follow earlier. Therefore, in this case, it is necessary to compare the numbers of jobs that are stored in a[2] and b[2].

```
int lt(vector<int> a, vector<int> b)
{
  if (a[0] * b[1] == b[0] * a[1]) return a[2] < b[2];
  return a[0] * b[1] < b[0] * a[1];
}
```

The main part of the program. Read the input data. Fill the array *jobs*.

```
while(scanf("%d",&n) == 1)
{
  jobs.clear();
  for(i = 1; i <= n; i++)
  {
    scanf("%d %d",&j[0],&j[1]); j[2] = i;
    jobs.push_back(j);
  }
```

Sort the jobs according to the comparator ***lt***.

```
  sort(jobs.begin(),jobs.end(),lt);
```

Print the result as required in the problem statement.

```
  for(i = 0; i < n; i++)
    printf("%d ",jobs[i][2]);
  printf("\n");
}
```

1592. Bridge

n people come to a river in the night. There is a narrow bridge, but it can hold only two people at a time. They have one torch and, because it's night, the torch has to be used when crossing the bridge. The movement across the bridge without a torch is prohibited.

Each person has a different crossing speed; the speed of a group is determined by the speed of the slower member. Your job is to determine a strategy that gets all n people across the bridge in the minimum time.

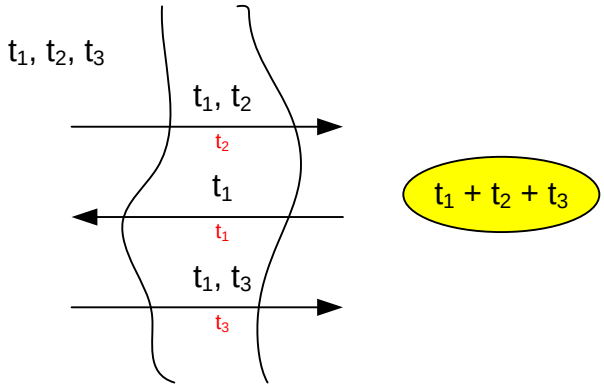
Input. Consists of multiple test cases. The first line of each test case contains the number of people n ($n \leq 10^3$), and the second line gives the sequence of n numbers – the crossing times for each of the people. Nobody takes more than 10^4 seconds to cross the bridge.

Output. For each test case print the next information. The first line must contain the total number of seconds required for all n people to cross the bridge. The following lines give a strategy for achieving this time. Each line contains either one or two integers, indicating which person or people form the next group to cross. Each person is indicated by the crossing time specified in the input. Although many people may have the same crossing time the ambiguity is of no consequence. Note that the crossings alternate directions, as it is necessary to return the flashlight so that more may cross. If more than one strategy yields the minimal time, any one will do.

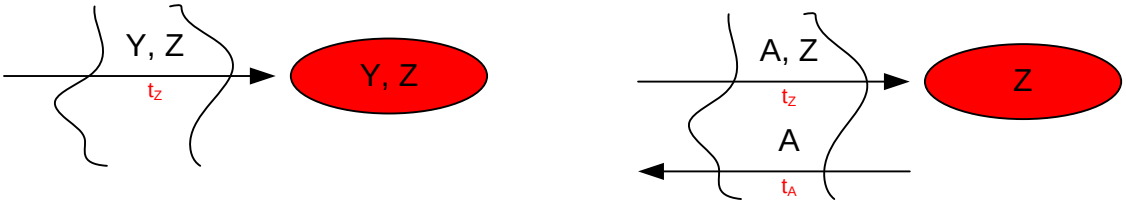
Sample input	**Sample output**
4 1 2 5 10 3 1 2 3	17 1 2 1 5 10 2 1 2 6 1 2 1 1 3

Algorithm analysis

Sort the time it takes people to cross the river in ascending order. Let t_i be the time of crossing the river by the i - th person ($t_1 \leq t_2 \leq \ldots \leq t_n$). Consider how one, two, or three people should cross the bridge. For $n = 1$ and $n = 2$, the optimal speed of crossing the river, respectively, is t_1 and $t_2 = \max(t_1, t_2)$ (the speed of movement of two people is equal to the speed of the slow one). In the case of three people ($n = 3$), the first and second go to the other side, the fastest (first) comes back with a lantern and transfers the third. Thus, the optimal time to cross the river is $t_1 + t_2 + t_3$.

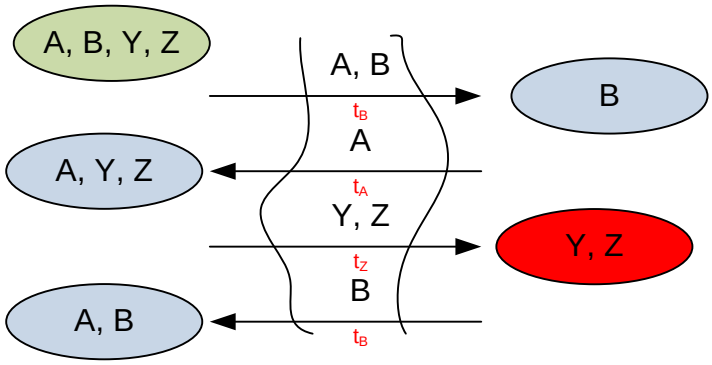


Consider the case when $n > 3$. Let $A \le B \le \ldots \le Y \le Z$ be the people sorted by time of crossing the bridge in increasing order (A is the fastest, Z is the slowest). Let J be the person with whom Z moves. If J stays on the other side and never returns back over the bridge, then it is optimal to choose it equal to Y. If J returns, then it is optimal to choose it as the fastest, that is, A. Thus Z can cross the bridge either with Y or A.

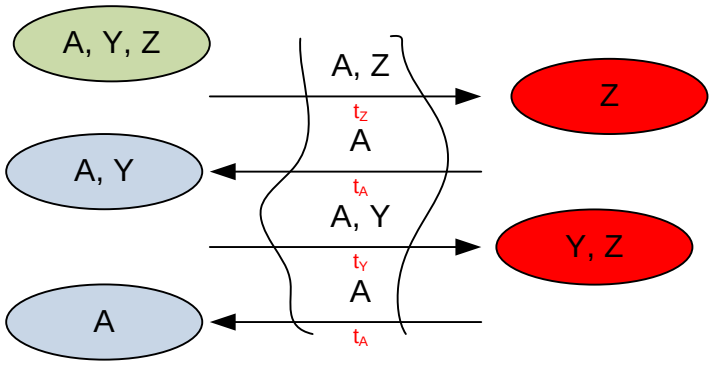


Compute the passage times of the two slowest people (Y and Z) according to these two strategies.

1. Z goes with Y. But then before that there must be somebody who will return the lantern, for example K. This K also had to be taken to the other side in order to return the lantern, and give it to Y and Z. Let it be L. Thus, K and L must return. To minimize the time, the two fastest should be selected as K and L, that is, A and B. The passage time for Y and Z is $t_A + 2t_B + t_Z$.



2. Z crosses the bridge together with A, then A returns. Then A crosses the bridge with Y and A returns. To go to another side of the river for Y and Z takes $2t_A + t_Y + t_Z$ time.



In both cases, only two slowest people cross the bridge. The strategy (first or second) is chosen depending which of the values ($t_A + 2t_B + t_Z$ or $2t_A + t_Y + t_Z$) is less. If initially n people should cross the bridge, then $n - 2$ people must do it recursively.

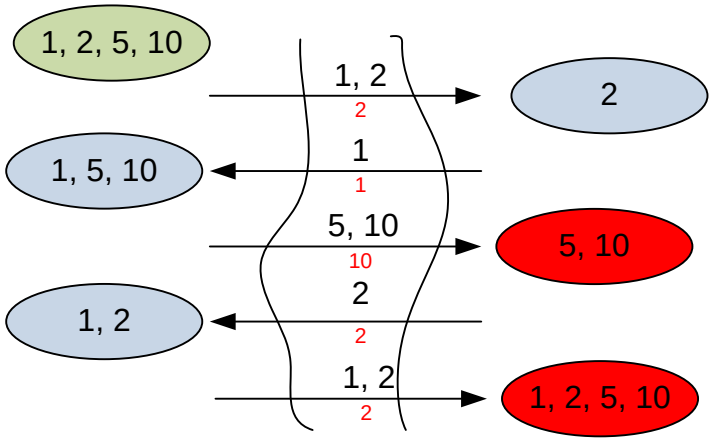
Example

Sort the times of crossing the bridge: 1, 2, 5, 10. Here $t_A = 1$, $t_B = 2$, $t_Y = 5$, $t_Z = 10$. The passage time of the two slowest people according to the first and second strategies are respectively equal

- $t_A + 2t_B + t_Z = 1 + 2 * 2 + 10 = 15$;
- $2t_A + t_Y + t_Z = 2 * 1 + 5 + 10 = 17$;

Since the first value is less, then Z should go with Y. Z with Y cross the bridge in time 15, after which it remains to go for A and B to the other bank. This is done in time $\max\{t_A, t_B\} = 2$.

The total time to cross the bridge is 15 + 2 = 17.



Algorithm realization

Array m stores the time of people to cross the river.

```
int m[1001];
```

Function go(*n*, *visible*) returns the optimal time in which *n* people can cross the river. The variable *visible* = 1, if the moving strategy itself should be printed, and *visible* = 0 otherwise.

```
int go(int n,int visible)
{
  int First, Second, Best;
```

One person crosses the river.

```
  if (n == 1)
  {
    if (visible) printf("%d\n",m[0]);
    return m[0];
  } else
```

Two people cross the river.

```
  if (n == 2)
  {
    if (visible) printf("%d %d\n",m[0],m[1]);
    return m[1];
  } else
```

Three people cross the river.

```
  if (n == 3)
  {
    if (visible)
    {
      printf("%d %d\n",m[0],m[1]);
      printf("%d\n",m[0]);
      printf("%d %d\n",m[0],m[2]);
    }
    return m[0] + m[1] + m[2];
  };
```

Compute the optimal time *First* and *Second* for the first and the second strategies described above.

```
  First = m[0] + 2 * m[1] + m[n-1];
  Second = 2 * m[0] + m[n-2] + m[n-1];
  Best = (First < Second) ? First : Second;
  if (visible)
  {
    if (Best == First)
    {
      printf("%d %d\n",m[0],m[1]);
      printf("%d\n",m[0]);
      printf("%d %d\n",m[n-2],m[n-1]);
      printf("%d\n",m[1]);
    } else
    {
      printf("%d %d\n",m[0],m[n-2]);
      printf("%d\n",m[0]);
      printf("%d %d\n",m[0],m[n-1]);
      printf("%d\n",m[0]);
    }
  }
```

Recursively compute the optimal strategy for the remaining $n - 2$ people.

```
  return Best + go(n-2,visible);
}
```

The main part of the program. Read the number of test cases, read the time it takes for people to cross the bridge into array m.

```
while(scanf("%d",&n) == 1)
{
  for(i = 0; i < n; i++) scanf("%d",&m[i]);
```

Sort the times of crossing the river in ascending order.

```
  sort(m,m+n);
```

Run function ***go*** with parameter *visible* = 0, that returns the optimal river crossing time. Print it, and then run function ***go*** again with parameter *visible* = 1, that prints the sequence of movements.

```
  res = go(n,0);
  printf("%d\n",res);
  res = go(n,1);
}
```

1599. Dynamic frog

With the increased use of pesticides, the local streams and rivers have become so contaminated that it has become almost impossible for the aquatic animals to survive.

Frog Fred is on the left bank of such a river. *n* rocks are arranged in a straight line from the left bank to the right bank. The distance between the left and the right bank is *d* meters. There are rocks of two sizes. The bigger ones can withstand any weight but the smaller ones start to drown as soon as any mass is placed on it. Fred has to go to the right bank where he has to collect a gift and return to the left bank where his home is situated.

He can land on every small rock at most one time, but can use the bigger ones as many times as he likes. He can never touch the polluted water as it is extremely contaminated.

Can you plan the itinerary so that the maximum distance of a single leap is minimized?

Input. The first line is the number of test cases *t* ($t < 100$). Each case starts with a line containing two integers *n* ($0 \le n \le 100$) and *d* ($1 \le d \le 10^9$). The next line gives the description of the n stones. Each stone is defined by s-m. *s* indicates the type Big (B) or Small (S) and *m* ($0 < m < d$) determines the distance of that stone from the left bank. The stones will be given in increasing order of *m*.

Output. For each test case print the case number followed by the minimized maximum leap.

Sample input 1	**Sample output 1**

```
3
1 10
B-5
1 10
S-5
2 10
B-3 S-6
```

```
Case 1: 5
Case 2: 10
Case 3: 7
```

Sample input 2

```
1
6 50
S-2 B-14 S-20 S-26 B-38 S-43
```

Sample output 2

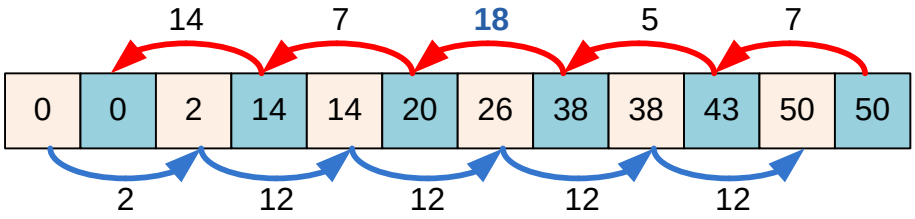
```
Case 1: 18
```

Algorithm analysis

Obviously, on the way back, the frog can use all the stones on its way. We need to develop a strategy for moving the frog from the left bank to the right. We'll assume that the left and right banks are large stones, and initially the frog is on the leftmost stone. Now we will replace each large stone with two small ones located in the same place. This can be done since it is obvious that the frog will use any large stone no more than twice. Since now we have a sequence of only small stones, we will formulate an algorithm for the frog's movement: when moving from the left to the right bank, it must jump over one stone every time – this is the principle of the greedy approach.

Example

Cnsider the second test case. The crossing contains n = 6 stones, the distance between the banks is d = 50. The left and right banks are represented by large stones. The array and the movement of the frog along it is as follows.



Algorithm realization

Read the input data.

```
scanf("%d\n",&tests);
for(t = 1; t <= tests; t++)
{
  scanf("%d %d\n",&n,&d);
  memset(m,-1,sizeof(m));
```

The left bank is one large stone. Replace it with two small ones.

```
  m[0] = m[1] = 0;
```

Read the information about stones and store in the m array. Put each large stone into the array twice, each small stone put only once.

```
for(ptr = 2, i = 0; i < n; i++)
{
  do {scanf("%c",&letter);} while (letter == ' ');
  scanf("-%d",&s);
  if (letter == 'B') {m[ptr] = m[ptr+1] = s; ptr += 2;}
  else {m[ptr] = s; ptr++;}
}
```

Represent the right bank with one large stone. Replace it with two small ones.

```
m[ptr] = m[ptr+1] = d;
ptr++;

scanf("\n");
```

Move from the leftmost stone to the rightmost one, jumping over one. We are looking for the maximum differences between the *i*-th and the (*i* + 2)-nd stones.

```
for(dist = 0, i = 2; i < ptr; i += 2)
  if (m[i] - m[i-2] > dist) dist = m[i] - m[i-2];
```

Decrease the value of *i* by 1. Now we move from right to left along neighboring stones that have odd numbers (stones with even numbers drowned when the frog moved to the right bank).

```
for(i--; i >= 2; i -= 2)
  if (m[i] - m[i-2] > dist) dist = m[i] - m[i-2];
```

Print the answer.

```
  printf("Case %d: %d\n",t,dist);
}
```

1593. Elegant permuted sum

You will be given *n* integers $\{a_1, a_2, \ldots, a_n\}$. Find a permutation of these *n* integers so that summation of the absolute differences between adjacent elements is maximized. We will call this value the *elegant permuted sum*.

Consider the sequence {4, 2, 1, 5}. The permutation {2, 5, 1, 4} yields the maximum summation. For this permutation *sum* = |2 – 5| + |5 – 1| + |1 – 4| = 3 + 4 + 3 = 10. Of all the 24 permutations, you won't get any summation whose value exceeds 10.

Input. The first line is the number of test cases *t* ($t < 100$). Each case consists of a line that starts with *n* ($1 < n < 51$) followed by *n* non-negative integers. None of the elements of the given permutation will exceed 1000.

Output. For each test case print the case number followed by the elegant permuted sum.

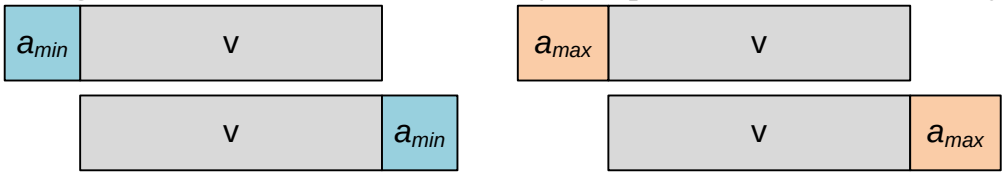
Sample input	Sample output
3 4 4 2 1 5 4 1 1 1 1 2 10 1	Case 1: 10 Case 2: 0 Case 3: 9

Algorithm analysis

Sort the numbers of the input sequence *a*. Create a new array *v*, where we'll construct the required permutation. Initially put into it the minimum and maximum elements of the sequence *a* (and, accordingly, remove these elements from *a*). We'll compute the elegant sum in the variable *s*. Initialize *s* = | v[0] – v[1] |.

As long as *a* is not empty, choose greedily the best choice among the following four possibilities:

1. The smallest element of the current array *a* is placed at the start of array v.
2. The smallest element of the current array *a* is placed at the end of array v.
3. The largest element of the current array *a* is placed at the start of array v.
4. The largest element of the current array *a* is placed at the end of array v.



For each case, recompute the new value of *s*. We make the choice for which the new value of *s* will be the largest. For each test, print the value of *s* as the answer.

Since *a* and *v* are dynamically updated, use deques as containers.

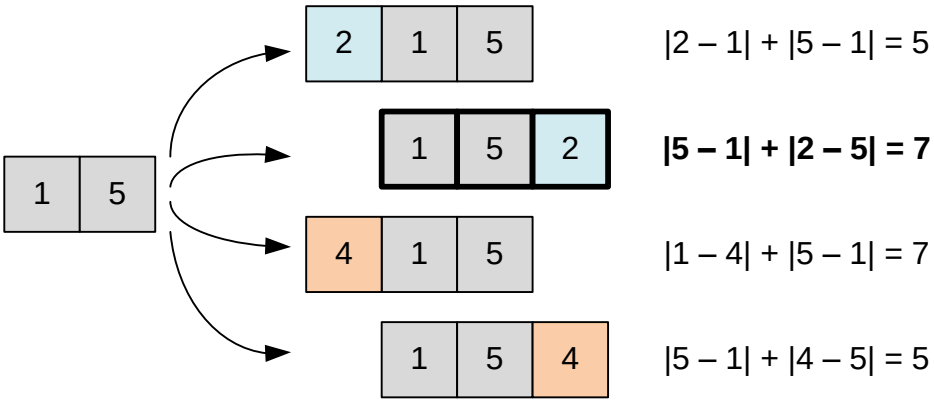
Example

Consider how the algorithm works for the first test case. Sort the array:

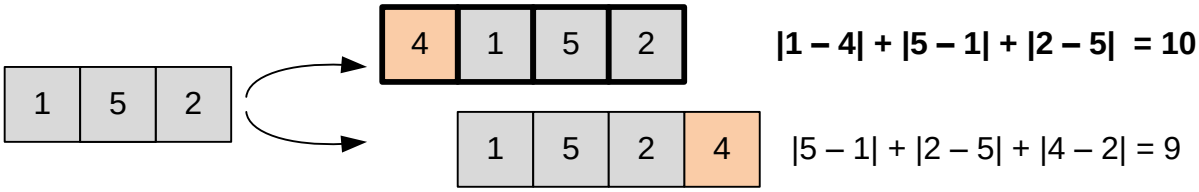
$$a = \{1, 2, 4, 5\}$$

Step 1. Push the smallest and the largest elements into array v: v = {1, 5}. Remove these elements from *a*, whereupon *a* = {2, 4}.

Append the smallest and the largest elements of array *a* to the right and to the left of array v. The largest value of the sum is reached, for example, on the array {1, 5, 2}.



Step 2. $v = \{1, 5, 2\}$, $a = \{4\}$. There is one element left in the *a* array. Append it to the right and to the left of array v. Recalculate the sums.



The resulting sum is 10, it is obtained, for example, for permutation

$\{4, 1, 5, 2\}$

Algorithm realization

Declare the deques.

```
deque<int> a, v;
```

Read the number of test cases *tests*.

```
scanf("%d", &tests);
for (i = 1; i <= tests; i++)
{
```

Start processing the next test xase. Clear the contents of arrays.

```
  scanf("%d", &n);
  a.clear(); v.clear();
```

Read the input array *a*.

```
  for (j = 0; j < n; j++)
  {
    scanf("%d", &val);
    a.push_back(val);
  }
```

Sort the input array.

```
  sort(a.begin(), a.end());
```

Push the minimum and the maximum elements of the sequence *a* into array v.

```
  v.push_back(a.back());
  v.push_front(a.front());
```

Initially set $s = | v[1] – v[0] |$.

```
  s = abs(v.back() - v.front());
```

Delete these two elements from array *a*.

```
  a.pop_back();
  a.pop_front();
```

While array *a* is not empty, we consider 4 cases and make the optimal choice among them using the greedy method.

```
  while (!a.empty())
  {
```

Declare an integer array mx of four elements.

```
    mx[0] = abs(v.front() - a.front());
    mx[1] = abs(v.back() - a.front());
    mx[2] = abs(v.front() - a.back());
    mx[3] = abs(v.back() - a.back());
    rmax = *max_element(mx, mx + 4);

    if (rmax == mx[0])
    {
      v.push_front(a.front());
      a.pop_front();
    } else
    if (rmax == mx[1])
    {
      v.push_back(a.front());
      a.pop_front();
    } else
    if (rmax == mx[2])
    {
      v.push_front(a.back());
      a.pop_back();
    } else
    {
      v.push_back(a.back());
      a.pop_back();
    }
    s += rmax;
  }
```

Print the answer.

```
  printf("Case %d: %d\n", i, s);
}
```