Greedy algorithms

Algorithms for optimization problems typically go through a sequence of steps, with a set of choices at each step. A **greedy algorithm** always makes the choice that looks best at the moment. That is, it makes a *locally optimal* choice in the hope that this choice will lead to a *globally optimal* solution.

E-OLYMP [6198. Minimum sum](#) Two arrays of positive integers are given $a_{1..n}$ and $b_{1..n}$. Find the permutation $i_1, i_2, \ldots, i_n$ of numbers 1, 2, ..., n, for which the sum

$$a_1 * b_{i1} + \ldots + a_n * b_{in}$$

is minimal. Each number must be included in permutation only once.

► Sort array a in ascending order and array b in descending order. Then the required minimum sum is

$$\sum_{i=1}^{n} a_i \cdot b_i$$

Consider how to get the maximum sum for the input sample.

a_i	2	3	4	7	10	
b_i	11	9	6	6	5	
$a_i * b_i$	22	27	24	42	50	165

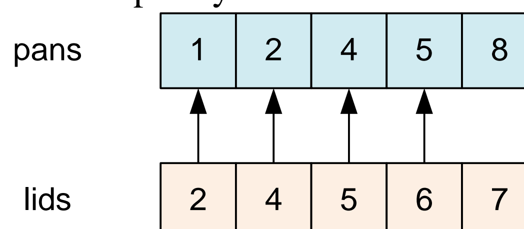
E-OLYMP [7335. Saucepans and lids](#) Great disaster happened this morning in the cafe where you used to have your snack in university. Larysa Ivanivna, the cleaner, while sweeping the floor dropped one of cupboards and all kitchenware stored there has scattered over the floor. Luckily, this cupboard contained only saucepans with lids. However, some items bent or broke, so has been disposed.

Now the schoolmaster wants to calculate losses and understand how many new saucepans and lids should be bought. First, she needs to find out how many remaining saucepans can be covered with remaining lids.

Saucepans and lids are round shaped. A lid can cover a saucepan if and only if radius of the lid is not less than radius of the saucepan.

► Sort the radii of the lids and the radii of the saucepans in ascending order. For the smallest saucepan, find the smallest lid that can cover it. Next, for the second smallest pan, find the smallest suitable lid for it, and so on. The answer is the number of saucepans that can be covered.

Let's find the largest number of pans you can cover with lids for the given sample.



Read the input data.

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

Sort the radii of the pans and lids.

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

Using the greedy method, look each time for the smallest lid that can cover the smallest pan.

```
for(i = j = 0; i < n, j < m; j++)
  if (pan[i] <= lid[j]) i++;
```

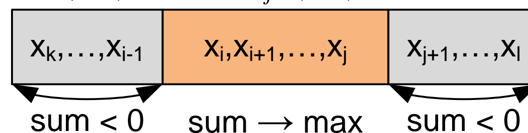
Print the number of covered pans.

```
printf("%d\n",i);
```

E-OLYMP [2585. Profit](#) The cows have opened a new business, and Farmer John wants to see how well they are doing. The business has been running for n ($1 \le n \le 100000$) days, and every day i the cows recorded their net profit P_i ($-1000 \le P_i \le 1000$).

Farmer John wants to find the largest total profit that the cows have made during any consecutive time period. (Note that a consecutive time period can range in length from one day through n days.) Help him by writing a program to calculate the largest sum of consecutive profits.

► In the problem we need to find a subsequence of consecutive numbers that will have the maximum possible sum among all possible such subsequences. If the maximum sum is attained on subsequence $x_i, x_{i+1}, \ldots, x_j$, then for any k, $1 \le k < i$ and l, $j < l \le n$, the sum of elements $x_k, \ldots, x_{i-1}$ and $x_{j+1}, \ldots, x_l$ will be negative.



Kadane's algorithm. Move through the array from left to right and accumulate the current partial sum in the variable s. If at some moment s turns out to be negative, then we assign $s = 0$. The maximum of all values of the variable s during the passage through the array will be the answer to the problem.

Consider a sequence X given below. Construct the partial sums. The current value of the partial sum is set to zero when the current sum becomes less than zero and we start counting the sum from the next number. The maximum value among all partial sums is 6, which is the answer. The required subsequence is 4, -2, 4.

X	5	-3	1	-7	4	-2	4	-1	-8	2
s	5	2	3	-4 0	4	2	6	5	-3 0	2
				s = 0					s = 0	

Read the length of the sequence *n*. Reset the value of the maximum profit *max* and the current partial sum *s*.

```
scanf("%d",&n);
s = 0; max = 0;
```

Read *n* integers. Each input integer *Number* is added to the current sum *s* and the current value of the profit *max* is recalculated. If at some step the value of *s* becomes negative, then we set it to zero and continue the process.

```
for(i = 0; i < n; i++)
{
  scanf("%d",&Number);
  s += Number;
  if (s > max) max = s;
  if (s < 0) s = 0;
}
```

Print the answer.

```
printf("%d\n",max);
```

E-OLYMP **138. Cash machine** The cash machine contains the sufficient number of banknotes with denominations 10, 20, 50, 100, 200 and 500 hryvnias. Find the minimum number of bills to give an amount of *n* hryvnias or print -1 if its impossible.

► Take the note with maximum denomination *c* and use it to give the amount *n* as long as possible. The maximum number of such notes that can be given is *n* / *c*. The rest *n* % *c* hryvnia will be given with other denominations.

The sum of 770 hryvnia can be given in the next way: 500 + 200 + 50 + 20.

E-OLYMP **8538. Calculator** Ilya's calculator performs two actions: multiplies the current number by three and adds one to it. The calculator now shows the number 1. Help Ilya to determine the smallest number of operations, after which he will get the number *n*.

► Move from number *n* to 1 in greedy way:

- if *n* is divisible by 3, divided it by 3;
- otherwise subtract one from *n*;

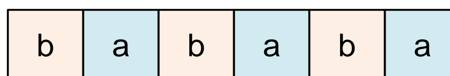
Let *n* = 21. You can get 1 for the minimum number of operations (for four) as follows:

$$21 \to 7 \to 6 \to 2 \to 1$$

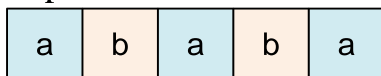
E-OLYMP [9002. Chocolate lover] Aziz is very fond of eating chocolate. But since chocolate is very bad for teeth, his father won't let him eat a lot of chocolate. This time he managed to convince his father and get permission to eat one chocolate every day. Aziz loves two types of chocolate. One weighs a grams, the other b grams. Aziz's father allowed him to eat one chocolate every day for n days, but with condition that he should not eat the same chocolate for two days in a row. Now Aziz is worried about only one question. How to make sure that within n days he could eat the maximum amount (in grams) of chocolate.

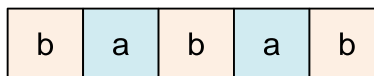
Help him with this.

► If the number of days n is even, then Aziz can eat chocolate either in the order a b a b ... a b, or in the order b a b ... a b a. However, in both cases, he will eat $(a + b) * n / 2$ grams of chocolate.



For odd n, Aziz can consume chocolate in one of the following orders: a b a b ... a, or b a b ... a b. In the first case he will eat $n / 2 * b + (n + 1) / 2 * a$ chocolate, in the second case he will eat $n / 2 * a + (n + 1) / 2 * b$ chocolate. It remains to choose in which of these options the mass of chocolate is greater.





Indeed, the sequence a b a b ... a of length n (n is odd) contains $\lfloor n/2 \rfloor$ letters a and $\lceil n/2 \rceil$ letters b.

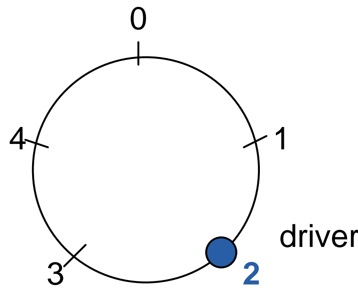
E-OLYMP [4211. Gas stations] Along the circumferential highway of length l there are n gas stations are located. If the driver wants to refuel at some point in the highway, he can drive up to any gas station, where he was happy to serve. Of course, if gasoline suddenly appears quite at the end, the driver, of course, go to the nearest gas station, even if he have to turn back.

However, occasionally there are unlucky drivers who suddenly right on the highway petrol ends. Determine the maximum possible distance to the nearest gas station, which require drivers to overcome this distance.

► When the driver runs out of gas, he can move either forward or backward on the highway. If the distance between some neighboring gas stations is d, and the driver is on the highway somewhere between them, then in the worst case (if the gasoline runs out just in the middle) he will have to cover the distance $d / 2$. It remains to find the greatest distance between all neighboring gas stations and divide it by 2.

Let l_1 and l_n be respectively, the positions of the first and last gas stations. Since the highway is circular and its length is l, the distance between the first and last gas stations is $l - l_n + l_1$.

Consider the sample from the problem statement. The greatest distance is reached between the gas stations in positions 1 and 3. It equals to $d = 2$. In the worst case, the driver have to walk the distance $d / 2 = 1$ to the nearest gas station.



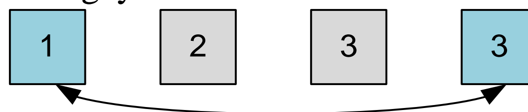
E-OLYMP 609. Water Recently Sergey went to the well for water, but did not return. He took n cans with him, each of which he filled completely with water. Now Sergey wants to deliver them to his country house. This is where the problem lies. At one time Sergey can carry no more than 2 cans because he has only two hands. Moreover, he can carry no more than k liters of water.

Now Sergey stands at the well and thinks about the minimum number of times he can take all the water home, and whether he can to do it at all. Help him solve this problem.

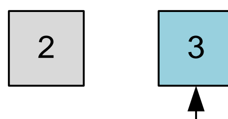
► If the volume of some canister is more than k, then it is impossible to take all the water home, print "Impossible".

Sort the canisters in ascending order of volume. Let's try to carry the largest canister together with the smallest one. If this is not possible, then the largest canister should be carried home alone. If possible, then remove the largest and the smallest canister from consideration, and solve the problem for the remaining canisters in the same way.

Consider an example from the problem statement. Volumes of available cans are: 1, 2, 3, 3. Sergey can carry no more than $k = 4$ liters of water at a time. The sum of volumes of the smallest and the largest canisters is $1 + 3 = 4$, which is no more than k. Therefore, for the first time Sergey will transfer these two cans.



Canisters 2 and 3 remain. The sum of the volumes of the smallest and the largest canisters is $2 + 3 = 5$, which is more than k. Therefore, at the second time, the largest canister should be carried home alone.



For the third time, Sergey will take home the last canister with volume of 2 liters.

E-OLYMP 6249. Planting trees Farmer Jon has recently bought n tree seedlings that he wants to plant in his yard. It takes 1 day for Jon to plant a seedling, and for each tree Jon knows exactly in how many days after planting it grows to full maturity. Jon would also like to throw a party for his farmer friends, but in order to impress them he

would like to organize the party only after all the trees have grown. More precisely, the party can be organized at earliest on the next day after the last tree has grown up.

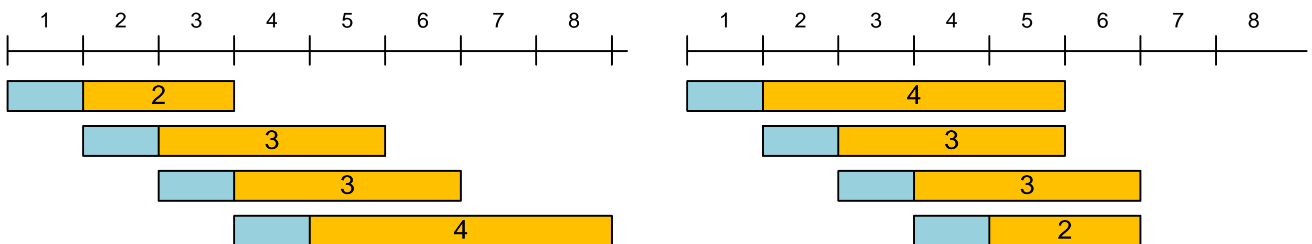
Help Jon to find out when is the earliest day when the party can take place. Jon can choose the order of planting the trees as he likes, so he wants to plant the trees in such a way that the party will be as soon as possible.

► It is beneficial to plant the tree that grows the longest first. Sort the growth times of trees in descending order – in this order we'll plant them.

Let John plant a tree on the i-th day that will grow for t_i days. This means that the tree will grow from the $(i + 1)$-th day to the $(i + t_i)$-th. If the i-th tree is the last to grow, then the party can be held on the $(i + t_i + 1)$-th day. It remains to find the maximum among these values, which will be the answer:

$$\max_{1 \le i \le n}(i + t_i + 1)$$

On the left we will consider the case if we'll plant the trees in the order 2, 3, 3, 4. On the right, consider the case when trees are planted in a decreasing (optimal) order of their growth time. The day the tree was planted is marked in blue.



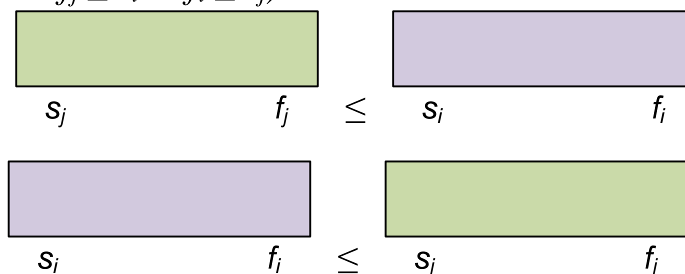
With the optimal planting of four trees, a party can be held on the day number $\max_{1 \le i \le 4}(i + t_i + 1)$, which equals to

$$\max(1 + 4 + 1, 2 + 3 + 1, 3 + 3 + 1, 4 + 2 + 1) = \max(6, 6, 7, 7) = 7$$

An activity-selection problem

Suppose we have a set S = $\{a_1, a_2, \ldots, a_n\}$ of n activities that wish to use a resource, such as a lecture hall, which can be used by only one activity at a time. Each activity a_i has a *start time* s_i and a *finish time* f_i $(0 \le s_i < f_i)$. If selected, activity a_i takes place during the half-open time interval $[s_i, f_i)$.

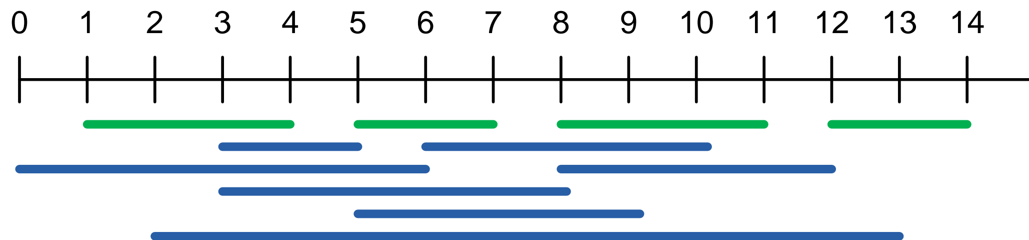
Activities a_i and a_j are **compatible** if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not *overlap* (a_i and a_j are compatible if $f_j \le s_i$ or $f_i \le s_j$).



The **activity-selection problem** is to select a maximum-size subset of mutually compatible activities.

For example, consider the following set S of activities, which we have sorted in *monotonically increasing order* of finish time:

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14



For this example, $\{a_1, a_4, a_8, a_{11}\}$ is the largest subset of mutually compatible activities.

To solve the problem, we must:

- sort intervals in *monotonically increasing order* of finish time;
- greedily select the intervals: iterate over them from left to right and select those that do not intersect with already selected;

Let A be the maximum set of chosen compatible activities.
Let n be the number of activities (intervals).
The variable i indexes the most recent addition to A, corresponding to the activity a_i. f_i is always the maximum finish time of any activity in A.

The **for** loop considers each activity a_m in turn and adds a_m to A if it is compatible with all previously selected activities To see if activity a_m is compatible with every activity currently in A, it suffices to check that its start time s_m is not earlier than the finish time f_i of the activity most recently added to A. If activity a_m is compatible, then add activity a_m to A and set i to m.

<u>GREEDY-ACTIVITY-SELECTOR(s, f)</u>
A = $\{a_1\}$
i = 1
for m = 2 to n
  do if $s_m \geq f_i$
    then A = A U $\{a_m\}$
        $i = m$
return A

E-OLYMP **66. The Director's Visitors** n time intervals [T_{1i}; T_{2i}) are given. Find the maximum number of non-intersecting intervals.

► Convert time to minutes. Create an array of segments – time intervals [T_{1i}; T_{2i}). Sort them in ascending order of *finish* time.

Next, greedily select visitors: iterate over the intervals from left to right and select those intervals that do not intersect with already selected.

Declare the class ***segment***, that will store the time interval [*start*; *fin*).

```
class segment
{
public:
  int start, fin;
  segment(int start, int fin) : start(start), fin(fin) {}
};
```

Store the set of input intervals in the vector v.

```
vector&lt;segment&gt; v;
```

Function ***f*** to sort the segments. The intervals are sorted in ascending order of their finish time.

```
int f(segment a, segment b)
{
  return a.fin <= b.fin;
}
```

The main part of the program. Read the input data.

```
scanf("%d", &n);
while (n--)
{
  scanf("%d:%d %d:%d", &h1, &m1, &h2, &m2);
```

Save the time in intervals in minutes.

```
  v.push_back(segment(h1 * 60 + m1, h2 * 60 + m2));
}
```

Sort the time intervals.

```
sort(v.begin(), v.end(), f);
```

We accept a visitor who has a zero interval (intervals are numbered from zero). Let the current visitor accepted by director, corresponds to the time interval *cur*. Since the director will always accept a visitor number zero, set *cur* = 0. In the variable *res* we count the number of accepted visitors. Since director accepts visitor number 0, initialize *res* = 1.

```
cur = 0; res = 1;
```

Iterate the intervals, starting from $i = 1$.

```
for (i = 1; i < v.size(); i++)
{
```

Look for an interval, the start of which is not less than the finish time of the current one. We accept a visitor from this interval and set this interval as current.

```
  if (v[i].start >= v[cur].fin)
  {
    cur = i;
    res++;
  }
}
```

Print the number of accepted visitors.

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

E-OLYMP **4786. Segments** n segments $[l_i; r_i)$ are given. Find the maximum number of non-intersecting intervals.

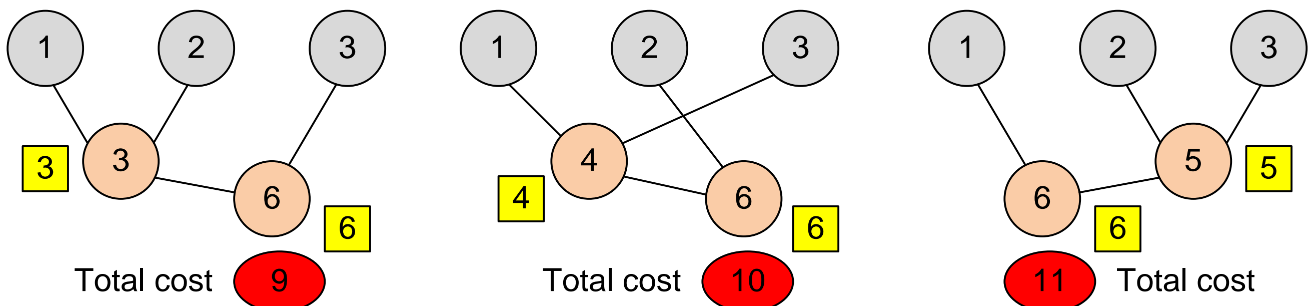
► Use GREEDY-ACTIVITY-SELECTOR algorithm.

Middle level problems

E-OLYMP **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

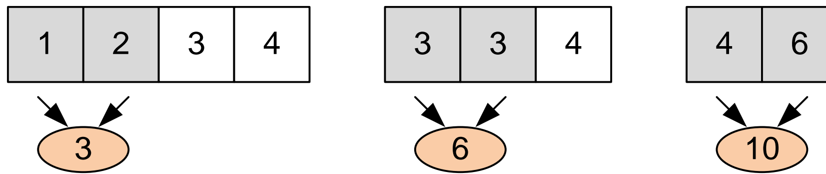
We hope you understood the task. You must add all numbers so that the total cost of summation will be the smallest.



► 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*.

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

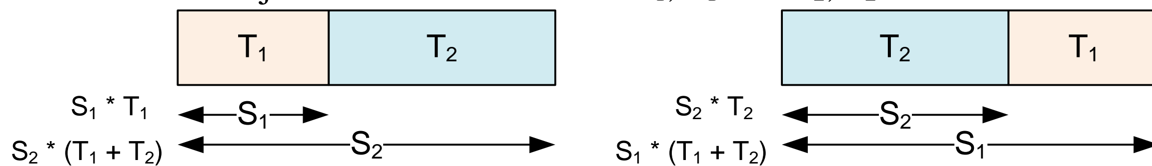
1. Add 1 and 2, sum is 3. Cost of addition is 3.
2. Add 3 and 3, sum is 6. Cost of addition is 6.
3. Add 4 and 6, sum is 10. Cost of addition is 10.



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

E-OLYMP 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.

► 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.

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.

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");
}
```

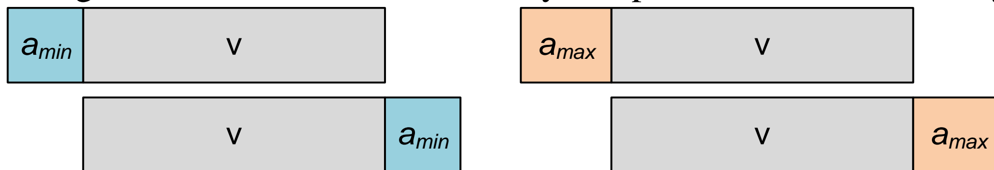
E-OLYMP [1593. Elegant permuted sum] You will be given n integers $\{a_1, a_2, \ldots, a_n\}$. Find a permutation of these n integers so that summation of the absolute differences between adjacent elements is maximized. We will call this value the *elegant permuted sum*.

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.

► 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.

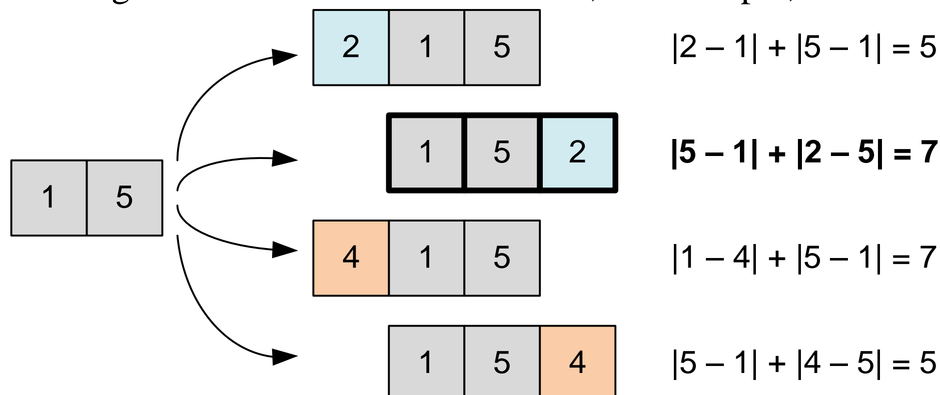
Since *a* and *v* are dynamically updated, use deques as containers.

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

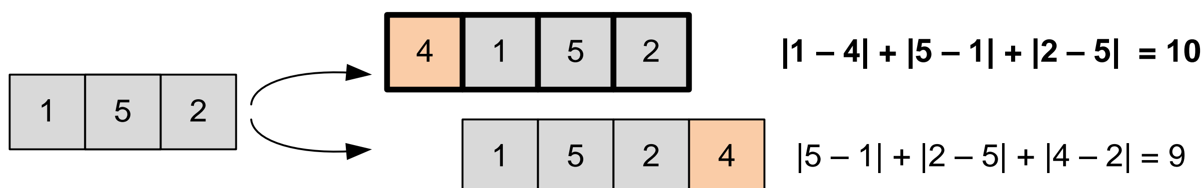
$$a = \{1, 2, 4, 5\}$$

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}

Knapsack

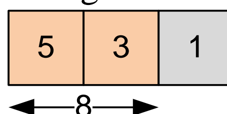
E-OLYMP [2103. Knapsack](#) Vasya is going to hike with fellow programmers and decided to take a responsible approach to the choice of what he will take with him. Vasya has *n* things that he could take with him in his knapsack. Every thing weighs 1 kilogram. Things have different "usefulness" for Vasya.

The hiking is going to be very long, so Vasya would like to carry a knapsack of weight no more than *w* kilo.

Help him to determine the total "usefulness" of things in his knapsack if the weight of backpack can be no more than *w* kilo.

► Let's use the greedy approach. Sort things in order of non-increasing usefulness. Since the weight of each item is 1 kilogram, Vasya should take min(w, n) of the most useful items.

Consider the sample from the problem statement. Sort three given items in descending order. And take the two with the greatest usefulness.



E-OLYMP [4831. Knapsack](#) Find the maximum weight of gold that can be carried out in a knapsack of capacity *s*, if there are *n* gold ingots with specified weights.

► Create an array *m*, in which we set m[i] to 1, if we can obtain the weight i with the available ingots. Initially set m[0] = 1.

Let the array *m* have already been filled in the required way for some set of ingots. The next ingot of weight *w* arrives. Then one should set to 1 all such m[i] ($w \le i \le s$) for which m[$i - w$] = 1. The answer is the largest weight, not bigger than *s*, that you can carry in your backpack.

Consider the filling of the cells of array m with the arrival of the next ingot. The weights of the ingots are indicated on the left.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
init	1																				
5	1					1															
7	1					1		1					1								
12	1					1		1					1					1		1	

Declare the array.

```
#define MAX 10010
int m[MAX];
```

Read the input data.

```
scanf("%d",&s);
```

Initialization of array m.

```
memset(m,0,sizeof(m));
m[0] = 1;
```

Process the the next ingot of weight w. We go through the array m from right to left and set to 1 all such m[i] ($w \le i \le s$) for which m[$i - w$] = 1.

```
while(scanf("%d",&w) == 1)
{
  for(i = s; i >= w; i--)
    if (m[i - w] == 1) m[i] = 1;
}
```

Look for the largest weight no more than s, that can be carried in the backpack, and print it.

```
for(i = s;; i--)
  if (m[i] > 0) break;
printf("%d\n",i);
```