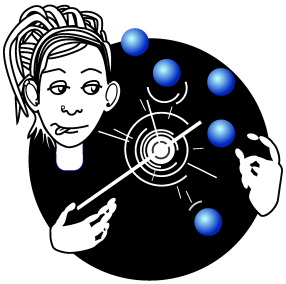
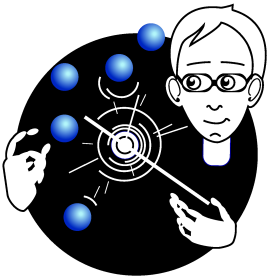
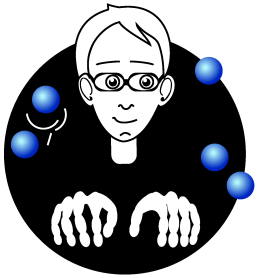
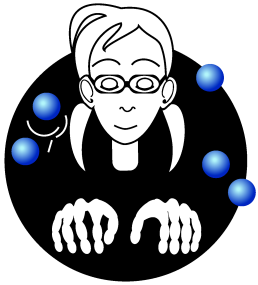
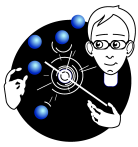



SOI 2016

The Swiss Olympiad in Informatics



How to write a perfect solution

Before we present the tasks themselves to you, we would like to give you some hints, how to write a good solution in theoretical track.

Unless stated otherwise in the task description, your solution should contain:

- *Source code of the program* in Pascal, C, C++ or Java. Your program will not be tested automatically, but reviewed manually by the organizers. Hence your source code should contain a lot of comments and you should write it as simple and elegant as possible. You do not need to spend much time with handling input and output in a fancy way (it is not forbidden to do so, but you will not score any more points; on the other hand, by doing so you could make your solution less comprehensible to the organizers).
- *Description of your solution.* This does not mean just to translate the source code to your mother tongue, but it should contain:
  - Description of the basic idea of your solution, without any implementation details such as names of variables, procedures, etc.
  - Description of the data structures used, e.g. what we need to store in the memory during the computation and how we will do it.
  - Description of the algorithm. Here you can use described data structures and mention important procedures, functions and variables.
- *Reasoning about correctness.* Unless stated otherwise, you do not need to present a rigorous proof of correctness of you algorithm, but you should provide solid arguments, why does your solution work. You should argue about all aspects of your solution that are not immediately obvious. If it is not obvious, you should include here also arguments why does your program always terminate.
- *Analysis of the time and space complexity of the algorithm.* Analysis of the time complexity gives an estimate on run time of the program, depending on the size of the input. This time is proportional to the

number of elementary operations of the program, such as assignment of a single variable, comparison of two variables, arithmetical operations, etc. We are usually interested in the run time in worst case, e.g. the maximal possible run time of the algorithm. (Sometimes we analyse an average run time). Analysis of the space complexity is analogical – it gives an estimate on the size of used memory, depending on the size of the input.

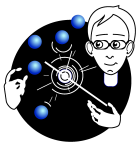
In the complexity analysis, we are not interested in exact size of used memory or exact running time (these are not only difficult to measure, but varies significantly between various computers). Instead of that, we focus on their rate of growth, expressed in the O-notation. We say that function $f(n)$ is $O(g(n))$ if it is possible to estimate $f(n)$ by some multiple of[1] $g(n)$.

For example, let program runs at most $T(n) = 0.5n^3 + 5n \log n + 83$ microseconds for an input of size n. Since $0.5n^3$ grows much faster than[2] $5n \log n$, we can say that the time complexity of the program is $O(n^3)$. (Which we can read also as: "The running time of the algorithm is in worst case asymptotically n^3.") As you can see, the multiplicative constant 0.5 is not important in the complexity analysis. Other thing you should notice is that if $T(n)$ is $O(n^3)$, it is also $O(n^4)$, $O(n^7)$,.... In the complexity analysis, we always try to give the smallest upper bound for, e.g. the slowest growing function possible.

To analyse time and memory complexity in this way is usually quite easy. For example, if a program contains two nested loops, each executing n times, its time complexity is $O(n^2)$. As an another example, look at this programs:

[1] Mathematically precise definition:
$f(n) = O(g(n)) \Leftrightarrow \exists n_0 \in \mathbb{N}, c \in \mathbb{R}^+; \forall n > n_0; f(n) \leq c \cdot g(n)$

[2] Formally, $\lim_{n\to\infty} \frac{0.5n^3}{5n \log n} = \infty$.



C:

```
int i,j,c;
int A[n];

void main() {
  c = 0;
  for (i=1; i<=n; i++)
    for (j=i+1; j<=n; j++)
      if (A[i]<A[j]) c++;
  printf("%d\n", c);
}
```

Pascal:

```
var i,j,c:integer;
    A:array[1..n] of integer;

begin
  c := 0;
  for i:=1 to n-1 do
    for j:=i+1 to n do
      if A[i]<A[j] then inc(c);
  writeln(c);
end.
```

The first loop will be executed $n-1$ times. The second loop is executed $n-i$ times for each execution of the first loop. So the comparison $A[i] < A[j]$ will be executed $(n-1)+(n-2)+\ldots+1 = \frac{n(n-1)}{2}$ times. So the time complexity of this program is $O(n^2)$. As for the memory complexity, the program uses 3 integer variables and one integer array of length n. Hence the memory complexity of the program is $O(n)$.

An Example

TASK: You are given a sequence of n positive integer numbers. Write a program that decides if sum of some consecutive numbers is equal to k.

SOLUTION: The simplest solution is to consider all sequences of consecutive numbers, compute their sum and compare it with k:

```
var n,k:integer;                    { Input values }
    C:array[1..MaxN] of integer;    { Input sequence }
    i,j,z,sum:integer;

begin
  for i:=1 to N do                  { For each subsequence }
    for j:=i to N do
      begin
```

```
        sum:=0;
        for z:=i to j do sum:=sum+C[z];  { Compute its sum }
        if sum=k then
          begin
            writeln('Subsequence exists'); halt
          end;
      end;
  writeln('Subsequence does not exist');
end.
```

This algorithm is easy to implement and it is easy to see that it is correct.[3] But what is its time complexity? It is easy to see that the most frequently executed operation is `sum:=sum+C[z]`, and it is executed for each member of each subsequence. Since there are $O(n^2)$ subsequences and each of them has length $O(n)$, the total time complexity of the algorithm is[4] $O(n^3)$. Can we do better?

It is easy to see that we do not need to recompute the sum of the considered sequence over and over. The sum $C[i] + \ldots + C[j-1] + C[j]$ can be obtained just by adding value $C[j]$ to the previously computed sum $C[i] + \ldots + C[j-1]$. This idea leads us to the following modification of the program:

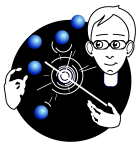
```
begin
  for i:=1 to N do
    sum := 0;
    for j:=i to N do
      begin
        sum := sum + C[j];    { Now sum = C[i]+C[i+1]+...+C[j] }
        if sum=k then
          begin
            writeln('Subsequence exists'); halt
          end;
      end;
```

[3] Since it tests all possible consecutive subsequences, computes sum of each such subsequence and compares it to k.

[4] Moreover, this complexity bound is tight: if we compute the number of operations more precisely, we obtain

$$\sum_{i=1}^{n} \sum_{j=i}^{n} j - i = \frac{n^3 - n}{6}$$

.



```
  writeln('Subsequence does not exist');
end.
```

This algorithm is more efficient – it requires only $O(n^2)$ operations.[5]

Next, we can do a small optimization: As soon as the variable `sum` exceeds the value k the program can stop adding another numbers to it and proceed to next loop of the outermost cycle (i.e. to the next value of i):

```
begin
  for i:=1 to N do
    sum := 0; j:=i;
    while (sum<k) and (j<=n) do   { End the loop as soon }
      begin                       { as possible }
        sum := sum + C[j];
        if sum=k then
          begin
            writeln('Subsequence exists'); halt
          end;
        j := j+1
      end;
  writeln('Subsequence does not exist');
end.
```

Unfortunately, the time complexity of this algorithm is the same as the previous one: if k is large enough, our optimization does not save anything. But now we can see another way of making the algorithm faster: After finishing the while loop the previous algorithm throws away computed value `sum` $= C[i]+C[i+1]+\ldots+C[j]$ and computes $C[i+1]+C[i+2]\ldots$ again. But since the while loop terminates as soon as possible, we know that `sum` $= C[i]+\ldots+C[j] > k$ and that $C[i]+\ldots+C[j-1] < k$. Hence also $C[i+1]+\ldots+C[j-1] < k$. So in the next iteration of the outermost loop, it is sufficient to start with the subsequence $C[i+1]+\ldots+C[j]$, whose sum can be obtained just by subtracting value $C[i]$ from variable `sum`:

```
var n,k:integer;                  { Input values }
    C:array[1..MaxN] of integer;  { Input sequence }
    i,j,sum:integer;
```

[5] This is again the best possible bound on complexity. The analysis is similar as in the example in the introduction to complexity analysis.

```
begin
  sum:=0; i:=1; j:=1;
  while (sum<>k) and (i<=n) do    { Check if sum=k }
    begin                         { after decrementing }
      while (sum<k) and (j<=n) do { Enlarge the subsequence }
        begin
          sum := sum + C[j];
          if sum=k then
            begin
              writeln('Subsequence exists'); halt
            end;
          j := j+1
        end;
      sum := sum-C[i];            { Shrink the subsequence }
      i := i+1;
    end;
  if sum=k then writeln('Subsequence exists')
           else writeln('Subsequence does not exist');
end.
```

What is the time complexity of this algorithm? The outer while loop executes at most n times, since variable i is incremented in each execution and it is never decremented. The inner while loop can execute $O(n)$ times for each execution of the outer loop. But together it can execute only n times, since the variable j is incremented in each execution and is never decremented. Hence the overall time complexity of the algorithm is $O(n)$. The memory complexity is $O(n)$, since the algorithm uses a constant number of integer variables and an array of integers of length n. The correctness of the algorithm follows directly from its description: we started with an obviously correct, but slow algorithm ($O(n^3)$ time), optimized it to an optimal algorithm and we showed that each of these optimizations was correct.