Interfaces provide a workaround for multiple-inheritance

- Defined similarly to classes,
- we define no members, we just delare them.
- They do not contain attributes (only methods).
- Syntactically – like with inheritance,
- multiple interfaces get separated by commas.

Dynamic programming

- Has nothing in common with dynamic allocation!
- Refers to a way of designing algorithms trying to avoid the recursion.
- At the beginning we design an algorithm using recursion...
- ... which is unnecessarily inefficient.
- Thus we may try to improve its complexity using cache.
- Finally we realize that we do not need the recursion.

Typical problems we solved by recursion

- Lecturer goes to S11,
- number of valid bracketings (using *n* pairs of brackets),
- number of decompositions of a number into sums of nonincreasing numbers (Young tableauxs),
- Pascal's triangle,
- longest increasing subsequence,
- matrix-bracketing (for purpose of multiplication),
- knapsack,
- ...

Lecturer goes to S11, again

and this time faster

- Lecturer comes from Ground floor to 1st floor and at the staircase he always either steps at the next step or steps over 1 step. In how many ways he can pass the staircase?
- Fibonacci numbers, we solved the problem using recursion:
- $f(n) := f(n-1) + f(n-2)$ (and starting conditions for $n = 1$ and 2.
- Solution is inefficient as we are computing repeatedly the same values.

Implementation
using recursion

```
...
static long fibonacci(int n)
{    if(n==1) return 1;
if(n==2) return 2;
return fibonacci(n-1)+fibonacci(n-2);
}
static void Main(){
     Console.WriteLine(fibonacci(Convert.ToInt32(
            Console.ReadLine())));
}
...
```

Problem: Computing still the same values
and how to avoid doing that

- Problem is clear, how to avoid doing it?
- We introduce a cache for results,
- always we take a look whether the result is in cache and when not, we perform a recursive call.

Example
using cache

```
static long[] c;
static long fib(int n)
{   if(c[n]==0)
    { if(n==1) c[n]=1;
        else if(n==2) c[n]=2;
            else c[n]=fib(n-1)+fib(n-2);
    }
    return c[n];
}
static void Main(){
     int i=Convert.ToInt32(Console.ReadLine());
     c=new long[i+1];
     Console.WriteLine(fib(i));
}
```

One step further

Do we need the recursion?

- If we take a closer look, it appears we do not need recursion.
- We replace it with a cycle...
- ... as we are calculating the values in an increasing ordering.
- We just have to fill-in the array (result is on its last position).

Example

using cycle and an array

```
static long[] c;
static void Main(){
     int i=Convert.ToInt32(Console.ReadLine());
     c=new long[i+1];
     c[1]=1;
     c[2]=2;
     for(int j=3;j<=i;j++)
          c[j]=c[j-1]+c[j-2];
     Console.WriteLine(c[i]);
}
```

Optimization

Do we need the array?

- We always use just last two elements of the array.
- Why should we waste the memory then?
- Let us "cache" last two values in variables.

Example

the final one

```
static void Main(){
     long
a=1,b=2,c,j,i=Convert.ToInt32(Console.ReadLine());
     switch(i)
     { case 1: b=1; break;
        case 2: b=2; break;
        default: for(j=3;j<=i;j++)
               {    c=a+b; a=b; b=c;}
     break;}
     Console.WriteLine(b);
}
```

Number of valid bracketings

- Considering *n* pairs of brackets, in how many ways we may organize them to obtain a valid bracketing (e.g.: $(()), ()())$.
- We were solving using recursion on position of a bracket in the expression.
- The algorithms was not too efficient as it was still computing the same values:
- `bracket(opening,closing,total)` always adds the same value for a given triple.
- And yet `total` is a constant.

Example

good old recursive algorithm

```
static long counter=0;
static void bracket(int o,int c,int t)
{    if((o==c) &&(o==t)) counter++;
     if(o>c) bracket(o,c+1,t);
     if(o<t) bracket(o+1,c,t);
}
static void Main()
{    bracket(0,0,Convert.ToInt32(Console.ReadLine()));
     Console.WriteLine(counter); }
```

What now?

Still the same

- The program still computes the same value. How to solve the problem now?
- In the same way, just...
- ... the array shall be two-dimensional (as there are two parameters)...
- ... and we have to workaround the fact that the function does not return the result!

Example

better recursive algorithm

```
static long[,] ca;    static long counter=0;
static void bracket(int o,int c,int t)
{    long tmp=counter;
     if(ca[o,c]==0)
     {    if((o==c) &&(o==t)) counter++;
          if(o>c) bracket(o,c+1,t);
          if(o<t) bracket(o+1,c,t);
          ca[o,c]=counter-tmp;
     }else counter+=ca[o,c];}
static void Main()
{    int i=Convert.ToInt32(Console.ReadLine());
     ca=new long[i+1,i+1];
     bracket(0,0,i);
     Console.WriteLine(counter);}
```

Do we need the recursion?

Still the same question, again the same answer...

- We saved on the recursion, but do we really need it?
- No, it suffices to fill-in the cache.
- How to fill the cache in?
- We may make a self-explanatory step aside using "debugging"-version.

Example

the *debugging* version

```
static long[,] ca;
static long counter=0;
static void bracket(int o,int c,int t)
{    long tmp=counter;
     if(ca[o,c]==0)
     {    if((o==c) &&(o==t)) counter++;
          if(o>c) bracket(o,c+1,t);
          if(o<t) bracket(o+1,c,t);
          ca[o,c]=counter-tmp;
          Console.WriteLine("Filling {0},{1} with
{2}",o,c,counter-tmp);
     }else counter+=ca[o,c];
}
```

Example

without recursion

```
static long[,] ca;
static void Main()
{    int i=Convert.ToInt32(Console.ReadLine()),a,b;
     ca=new long[i+1,i+1];
     for(a=i;a>=0;a--)
          for(b=i;b>=a;b--)
          {    if((i==b)&&(a==b))
               ca[b,a]=1;
               if(b>a) ca[b,a]=ca[b,a+1];
               if(b<i) ca[b,a]+=ca[b+1,a];
          }
     Console.WriteLine(ca[0,0]);
}
```

Decomposing a number

into sum of nonincreasing numbers

- Given a non-negative number, we want to know in how many ways we may decompose it into a sum of nonincreasing positive numbers.
- Example: 4: 4, 3+1, 2+2, 2+1+1, 1+1+1+1.
- We have solved also using a recursion (on position in the sum using maximum permitted value and the remainder).
- The algorithm was still often (and unnecessarily) decomposing values 1, 2, 3...

Example

recusive algorithm

```
static long counter=0;
static void decomp(int what, int max){
     int i,j;
     if(what==0) counter++;
     j=((max>what)?what:max);
     {    for(i=j;i>0;i--)
          decomp(what-i,i);
     }
}
static void Main(){
     int i=Convert.ToInt32(Console.ReadLine());
     decomp(i,i);
     Console.WriteLine(counter);
}
```

Again we make a cache

and again in the same way

- We create a cache parametrized by the arguments of the function,
- again we remember how many endings are added (like in bracketing),
- after doing that we may again get rid of the recursion...
- ... but this time we keep it as a homework.

Example

with a cache

```
static long[,] ca;
static long counter=0;
static void decomp(int what, int max){
    int i,j;long tmp;
    if(ca[what,max]==0)
    {   tmp=counter;
        if(what==0) counter++;
        j=((max>what)?what:max);
        {   for(i=j;i>0;i--)
            decomp(what-i,i);
         }
         ca[what,max]=counter-tmp;
     } else counter+=ca[what,max];
}
```

Pascal's triangle

contains combinatorial numbers describing how many *k*-element combinations are the using *n* elements

- We won't use explicit formula $\binom{n}{k} = \frac{n!}{k! \times (n-k)!}$,
- we use a recurrent formula saying: $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$.
- We want to determine just particular $\binom{n}{k.}$
- This formula can be implemented using recursion...
- but still such an algorithm sucks... still because of the same reason...
- thus we introduce a cache and realize that we do not need the recursion.
- This approach to the problem is called the **dynamic programming**.

Example
without cache

```
static long combin(int n, int k)
{    if((k==0)||(n==k))
          return 1;
     else return combin(n-1,k)+combin(n-1,k-1);
}
static void Main(){
     int n=Convert.ToInt32(Console.ReadLine()),
     k=Convert.ToInt32(Console.ReadLine());
     Console.WriteLine(combin(n,k));
}
```

Example

with a cache

```
static long [,]ca;
static long combin(int n, int k)
{    if(ca[n,k]==0)
     {    if((k==0)||(n==k))
               ca[n,k]=1;
          else ca[n,k]=combin(n-1,k)+combin(n-1,k-1);
     }
     return ca[n,k]; }
static void Main(){
     int n=Convert.ToInt32(Console.ReadLine()),
     k=Convert.ToInt32(Console.ReadLine());
     ca=new long[n+1,k+1];
     Console.WriteLine(combin(n,k));}
```

Dynamic programming

general atributes of the method I/II

- Considering a problem that can be solved with a recursion we realize that the recursion is asynchronous [it does not depend on anything except its parameters].
- Then it holds that such a recursion brings always the same result (for given arguments).
- To make use of this fact, it is necessary that the algorithm computes the values (for fixed arguments) repeatedly (and many times).
- So we use a cache, compute the value only once and read the cache since next time.
- And finally we should get rid of the recursion and replace it using several cycles.
- Complexity of the algorithm then usually decreases rapidly.

Dynamic programming
general atributes of the method I/II

- It is also possible to observe dynamic programming as a method "an sich", but then it is unclear where the algorithms appeared (and what's the common property of algorithms from this family).
- Note that dynamic programming behaves with recursion as, e.g., divide et impera did. But compared to divide et impera the division is not fixed (we are trying all possible divisions instead).
- So far we've been just plaing, more serious problems are comming: longest increasing subsequence, matrix multiplication (bracketing), knapsack, string metric,...

Longest increasing subsequence

- The recursive algorithm shall ask (for each element) how long is the longest increasing subsequence ending in this element.
- We will be still asking about subsequence consisting of the first element (at most).
- Thus we start caching and for each element we store length of longest increasing subsequence ending in this element.
- Then we sweep through the array and each "last"element gets attached after some already explored element.