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.

Matrix multiplication

- We should multiply a row of matrices (with compatible ranks).
- Matrix multiplication is not commutative but it is associative.
- Good bracketing can save us work. Thus how to bracket matrices for the purpose of multiplication?
- Recursively: One multiplication is the last one (we cannot say that one is the first one(!)).
- Thus we branch on all possibilities.
- And we will be continuously exploring pairs or individual matrices (unnecessarily).
- Thus we start caching the optimal bracketing for the sequence $M_i \dots M_j$.

Shortest path in a graph

what we know and what not

- On Discrete Mathematics you saw at least one algorithm. What was a crucial assumption?
- Dijkstra's algorithm: non-negative edge-values, Bellman-Ford's algorithm: no negative cycle.
- Shortest path in generally weighted graph hides the longest path with non-negative assignment ($\Rightarrow$ NP-hard).
- What if we want to find shortest paths between all vertex-pairs?

Floyd-Warshall's algorithm

based on dynamic programming

- The algorithm asks (for all pairs) about the shortest path (for an individual pair) using at most $n - 1$ edges.
- The recursion goes on length of paths.
- Recursion is parametrized by 3 values (from, to, length).
- Cache will be 3dimensional.

Floyd-Warshall's algorithm

how to fill the cache?

- A cache-element (with coords x, y, z) says the shortest x, y-path with at most z edges.
- The dynamic program's outer cycle shall be indexed by z, inner cycles by x, y.
- The most internal cycle takes all vertices w and tries whether a path with length $z + 1$ of form $x, ...w, y$ is shorter than the best so far known path (of length z or through different w'.
- Finding a path of length $z + 1$ means finding path of length z and an edge.
- The latter is in modified adjacency-matrix, the latter already in the cache.

Code snippets

are tying your life more strongly to C#!

- There are situations we have to solve very often when writing programs (in C#).
- We often create for-cycle, we use to write `Console.WriteLine`...,
- To avoid work demanding on typewriting, code snippets are present in Visual Studio.
- They get invoked by pressing <TAB> twice.
- `for`, `foreach`, `cw`, `do`, `else`, `forr`, `if`, `sim`, `svm`, `switch`, `while`, `try`, `tryf`... (altogether at least 38).
- Use them on your discretion – restricts typewriting, restricts you on a particular language (environment).

Exceptions

we know exceptions as enemies, now we make friends of them

- What to do if something goes wrong?
- Stop the program (and say what went wrong): Not too good.
- Design particular return-values saying that something went wrong: Restrict the range of a function.
- Ignore the problem (the ostrich-algorithm): Causes more harm than use.
- Send the information that something went wrong: OK, but how?
- That is why the Exceptions were designed.

Exception

how to domesticate them

- We know that division by zero, incompatible typecast, null-pointer dereference caused an exception.
- Idea is similar to compiler-directives in Pascal, just we have tu specify where the "directive"should take effect and we may invoke the exceptions, too. Also syntactically it looks differently.

Exceptions

- We know that the exceptions (so far) made the program finish.
- But we can "catch" the exceptions (and reflect the fact that something went wrong).
- We may also "throw" the exceptions.
- The exception is passing through the call-stack up to the block that "catches" this exception.
- There may be different types of exceptions and not all blocks catch all the exceptions.
- If we find no block catching recent exception, the program terminates (we leave even the method `Main`.

Exceptions

rules for exceptions

- Syntax and semantics:
- Keyword `try` introduces a block with possible exception-occurence.
- `catch` starts a block with the exception-handler (follows `try` block).
- There may be more `catch` blocks depending on types of exceptions we expect.
- `finally` starts a block that should be performed after the exception is handled (it may take effect even with the default handler that usually stops the execution).
- `throw` throws the exception. Syntactically it works like keyword `return`.

Exceptions

example

```
void safedivision(int a, int b)
{    try{
          Console.WriteLine(a/b);
     }
     catch(System.DivideByZeroException e)
     {    Console.WriteLine("Impossible!");}
}
```

Own exception

How to throw our own exception?

```
class me:System.Exception{}
...
void safedivision(int a, int b)
{    try{
          if(b==0) throw new me();
          return (a/b);
     }
     catch(System.Exception e)
     {    Console.WriteLine("Aiee, an exception is
here..."); }
     finally
     {    Console.WriteLine("Anyway...");}
}
```

Exceptions – remarks

- There may be several consecutive `catch` blocks.
- First matching block is executed (first block describing a compatible data-type).
- In C# it is necessary to define son-typed handlers before parent-typed handlers:
- `catch(System.Exception e){...}`
  `catch(System.DivideByZeroException e){...}`
  ... we will be unable to compile this source!

Exceptions – remarks

- How we should NOT write programs:

```
bool already=false;
while(!already)
{    try{function_that_sometimes_crash();
already=true;}
     catch(System.Exception e)
    {    Console.WriteLine("once more,
please...");}
}
```

- Exception is a good servant but a bad master!

Generic data-types

how to define them

- We know that there exist generic data-types (e.g., `List`). But how to define them?
- Remark: These data-types are light-version of templates in C++, those can be indexed by anything (e.g., by number).
- They can be used if we want to create several instances with different underlying data-type.
- It is also a replacement of preprocessor-macros in C.

Generics II

- When using generics, we proceeded as with normal data-type, just we added a parameter into angle-brackets [chevrons].
- When defining generic data-type we do almost the same, i.e., we just name the parameter and behave with that as with a data-type.
- There does not have to be only one parameter, more parameters are separated by commas.
- `public class gen_cl <T> {public T variable;}`

Generic class example

```
public class my_list <T>
{    public T data;
     public my_list<T> next;
}
...
     my_list<int> x=new my_list<int>();
     x.data=new my_list<int>();
     //This would not work:
     // x.next=new my_list<double>();
```

Generic methods

not only classes may be generic

- In the function-body the parameter behaves as when defining generic class.
- This time we put chevrons containing the data-types between function-name and parameters.
- `void gen_met<T,U>(T a, U b){ Console.WriteLine("Parameters are {0},{1}.",Convert.ToString(a),Convert.ToString(b));}`

Generic method example

```
static void swapit<T>(ref T a, ref T b)
{    T tmp=a;
     a=b;
     b=tmp;
}
static void Main()
{    int a=1,b=2;
     swapit<int>(ref a, ref b);
     Console.WriteLine("a is {0}, b is {1}",a,b);
}
```

Restricting data-types

in newer versions of .NET Framework

- We may restrict the parameter using keyword `where`.
- When defining the class, we put `where` behind the chevrons,
- when defining the methods, it stands after the header.
- `class gen<T> where T:IComparable{...}`
  `//T implements function CompareTo`
- `void gener<T>(out T a) where T:new(){a=new T();}`

Operator overloading

is the same as function overloading

- We know that a function is defined by its name and argument-structure.
- Several functions with the same name may exist.
- Also there are, e.g., many types of numbers: integers, longint, double, rational numbers...
- ... and we want to add, subtract, multiply or divide them...
- without calling obscure functions like `add_two_rationals`.

Complex numbers

and operations on them, a. k. a. operator overloading

- We want to create a class representing complex numbers,...
- whose elements can be added like `c=a+b;`
- thus we overload operator $+$.
- When overloading an operator, it looks like function overloading just the name of the function is fixed (e.g., "`operator +`") `and number of arguments, too (follows from grammar of C#).`
- We may overload an operator in a class identical with at least one of its parameters (we cannot overload an operator in completely different class).
- And the functions must be `static`!

Example

Gaussian integers

```
class compl
{    public int re, im;
     public compl(int re, int im)
     {    this.re=re; this.im=im;'}
     public static compl operator +(compl a,compl b)
     {    return new compl(a.re+b.re,a.im+b.im);}
     public static compl operator *(compl a,compl b)
     {    return new compl(a.re*b.im-a.im*b.im,
               a.re*b.im+a.im*b.re);
     }
}
```

Example – continued

To make the class `compl` demonstrable, we override her a method `ToString`, too:

```
public override string ToString()
{    return ""+re+"+ "+im+"i";}
```

And let's go:

```
kompl a=new compl(1,0), b=new compl(0,1),c;
c=a+b;
Console.WriteLine(c);
Console.WriteLine(a*b);
```

Overloadable operators

We may overload the operators:
unary `!`, `~`, `++`, `--`
binary `+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`, `<<`, `>>`
WE CANNOT overload mainly &&, ||, [], (*type*)*x*, +=, −=...

That's all for today...

...thank you for your attention.