Foundation of Information Processing

Design of algorithms

Considerations about what kind of design approaches (strategies) there are to make algorithms

Consideration 1:

What do recursion and iteration mean?

How modules are related to them?

When to apply recursion, when iteration?

Consideration 2:

What would be other means to solve a task algorithmically stepwise?

Consideration 3:

Is there a general purpose strategy of problem solving which could work in all cases?

Would some of the problem solving strategies be better than others?

How to select the best strategy for the task?

Design of algorithms is a many-sided process

Making algorithms requires *creativity*,
but there are *basic strategies*
to design and to implement algorithms.

The most typical ones are *recursion* and *iteration*.

Algorithmic problem solving

”The sooner you start coding your program,
the longer it is going to take.”
(Prof. Henry Ledgard, University of Toledo, Ohio, MIT graduate)

About principles of the design of algorithms:

- Presenting algorithms by stepwise refinement:
  - Sorting of items as an example.
- Modularity:
  - Modules, parameters, procedures, functions.
  - Structure of a program.
- Recursion and iteration.
- Overview of the main design principles.

Source: J. Boberg, Johdatus tietojenkäsittelytieteeseen, Turun yliopisto, 2010 (in Finnish).

Stepwise refinement: sorting as an example

Task: sort the names in the list in the alphabetical order.
Solution: bubble sort (comparisons of the adjacent names).
Design principle: stepwise refinement.

The initial solution (to be refined):

WHILE the list has not been sorted yet DO
    Go through the list by comparing two adjacent names and
    swap the names if they are in the wrong order.
ENDWHILE

Example: the swapped ones underlined; the number of swaps per full round in ()

1.1	*1.2*	*1.3*	*1.4*	*1.5*	*1.6*	*1.7 (4)*	*2 (3)*	*3 (2)*	*4 (0)*
John	John	John	John	John	John	John	Fred	Bertha	Bertha
Kate	Kate	Fred	Fred	Fred	Fred	Fred	Bertha	Fred	Fred
Fred	Fred	Kate	Bertha	Bertha	Bertha	Bertha	John	Jane	Jane
Bertha	Betha	Bertha	Kate	Kate	Kate	Kate	Jane	John	John
Sam	Sam	Sam	Sam	Sam	Jane	Jane	Kate	Kate	Kate
Jane	Jane	Jane	Jane	Jane	Sam	Mary	Mary	Mary	Mary
Mary	Mary	Mary	Mary	Mary	Mary	Sam	Sam	Sam	Sam

Stepwise refinement as a design method

Let us refine by adding details such as the name to be checked and the repetition structure to compare two adjacent names:

```
WHILE the list has not been sorted yet DO
    Set the name to be checked := the first name in the list
   REPEAT
      IF the name to be checked is after the next name
      in the list according to the alphabetical order THEN
         (* the names are in the wrong order*)
         Swap the names in the list
      ENDIF
      Set the name to be checked := the next name in the list
   UNTIL the list is at the end
ENDWHILE
```

Stepwise refinement (cont.)

Let us refine more by indexing the variable *name* in the list and by setting the number of the names:

```
WHILE the list has not been sorted yet DO
    i := 1 (* start with the first name of the list *)
    REPEAT
        IF name[i] is according to the alphabetical order
        after name[i+1] THEN
            Swap these names in the list
        ENDIF
        i := i + 1 (* move to the next name *)
    UNTIL i = N (* the end of the list *)
ENDWHILE
```

Stepwise refinement (cont.)

In the **”final” version,** the variable *changes* checks whether the list is in the order after all the adjacent names have been compared.

```
REPEAT
    changes := 0 (* no swaps done *)
    i := 1 (* start with the first name of the list *)
    REPEAT
        IF name[i] > name[i+1] THEN
            Swap these names in the list
            changes := changes + 1 (* the swap done *)
        ENDIF
        i := i + 1 (* move to the next name *)
    UNTIL i = N (* the end of the list *)
UNTIL changes = 0 (* no swaps done so the list in the order *)
```

Modularity and programs: modules, parameters, procedures and functions

- In the design of modules, the following matters should be considered:
  - **Task-specificity:** one module solves one (sub)task, not several ones.
  - **Naturality:** subtasks should be as natural parts of the algorithm as possible for the clarity of the algorithm and the general usability of the applied modules.
  - **Suitable level of abstraction:** a module should consist of submodules at the same level of abstraction.
- **Usability** is very important:
  - **Portability**: a module can be used anywhere where the same task appears, besides for the use it is defined originally.
  - **Replaceability**: a module can be replaced by another module (for example, by a more efficient module) in all cases where it appears, and it still solves the same task without any changes in details needed.
- Next, let us do revision and deepen our knowledge about the matters learned earlier.

Functions as modules

- A **function** is a module where there are
  - m input parameters and
  - n output parameters (many times one only).
- Definition:

  MODULE name **(fp1, fp2,..., fpN) RETURNS** value
  body
  ENDMODULE

- The parameters *fp1, fp2,..., fpN* are the *formal* parameters.
- In the body of the module there is at least one structure

  RETURN expression

  which returns the value of the function (the result).
- In our pseudo language, *value* in RETURNS can be a symbolic notation about the nature of the output since only the RETURN structure(s) defines the variable of the output.

Example: a function of factorial

```
MODULE factorial(n) RETURNS n!
    k := 1
    WHILE n > 1 DO
        k := k * n
        n := n – 1
    ENDWHILE
    RETURN k
ENDMODULE
```

- In our pseudo language, n! is a symbolic notation about the nature of the output so the factorial.
- n is the *formal* input parameter.
- k is the *formal* output parameter, defined by RETURN.
- By calling factorial(5),
  - the input 5 is the *actual* input parameter and
  - the *actual* output parameter (result) is 120.

Procedures as modules

- A **procedure** is a module where there are
  - m input parameters and
  - **no result** to be returned.
- Definition:

MODULE name **(fp1, fp2,..., fpN)**
  body
ENDMODULE

- In the body of the module there is no RETURN structure since the procedure **does not return the value**.
- Example: the variable *side* is the *formal* parameter which defines the length of the side of the square.
- By calling square(10), 10 is the *actual* input parameter.
- A programmer must define the unit of the input parameter (mm/cm/m?).

Example:

```
MODULE square(side)
   Put the pen on paper
   REPEAT 4 TIMES
      draw(side)
      turn(90)
   ENDREPEAT
   Lift the pen
ENDMODULE
```

The main module and other modules

- **Program** ≈ a programmed algorithm: a set of modules where one of them is the *main module* which starts the program.

```
MODULE main()
    body
ENDMODULE
```

```
MODULE main()
    x := factorial(5)
    print(x)
ENDMODULE
```

- **Module:** an “independent” and preferably general-purpose entity, consisting of statements that other modules can call.
- In the example the main module *main* calls the module *factorial* which computes the factorial of a given number.
- The factorial is defined as n!=n*(n-1)*(n-2)…*2*1 where n is a given input (here 5).
- The other module in the example is *print*.

Recursion as a design principle

- **Recursion** is dividing a problem into parts (reduction) where at least one subproblem is *similar* than the original problem.
- This enables efficient solutions, but the principle is not suitable to those problems which cannot be divided into similar subproblem(s) than the original problem.
- In practice, the **module calls itself** either directly or indirectly.
- Example: how to compute factorial recursively?

  n! = n * n-1 * … * 2 * 1

```
MODULE factorial(n) RETURNS n!
  IF n=0 THEN
    RETURN 1
  ELSE
    RETURN n*factorial(n-1)
  ENDIF
ENDMODULE
```

```
factorial(4)
= 4 * factorial(3)
= 4 * ( 3 * factorial(2) )
= 4 * ( 3 * ( 2 * factorial(1) ) )
= 4 * ( 3 * ( 2 * ( 1 * factorial(0) ) ) )
= 4 * ( 3 * ( 2 * ( 1 * 1 ) ) )
= 4 * ( 3 * ( 2 * 1 ) )
= 4 * ( 3 * 2 )
= 4 * 6
= 24
```

Iteration as a design principle

- **Iteration** is **repeating** the same set of statements (instructions) to proceed towards the solution.
- Better intermediate results are obtained until the solution is accurate or accurate enough (approximate solution).
- Example: how to compute factorial iteratively?

  n! = n * n-1 * … * 2 * 1

```
MODULE factorial(n) RETURNS n!
  k:=1
  WHILE n>1 DO
    k:=k*n
    n:=n–1
  ENDWHILE
  RETURN k
ENDMODULE
```

factorial(4): how k changes?
1 (in the beginning)
1*4=4 (k:=k*n, the 1st round of the loop)
4*3=12 (k:=k*n, the 2nd round)
12*2=24 (k:=k*n, the 3rd round)
24 (the result to be returned)

Design principles of algorithms

- Brute force: experiment with all possible solutions.
- Greedy algorithms: select the best local option.
  - Packing backpacks with items, pay with the least number of coins.
- Dynamic programming: use intermediate results.
  - Compute the Fibonacci number with a given value, find optimal routes.
- Divide and conquer: divide into subtasks.
  - Robust list sorting.
- Search space methods: find the best solution in the state space.
  - Backtracking, and branch and bound.
  - Relocate pieces of furniture, chess.
- Problem conversion: change the type of the problem to another type.
- Randomized/approximate methods:
  - Compute the approximate value of the square root.
  - Packing backpacks with items approximately.

Brute force:
experimenting with ”all” possible solutions

- Simple, direct, or obvious solutions are being used to solve the problem.
- For example, experimenting with **all the possible options**, and then selecting the best one.
- The method can be applied if the problem can be solved algorithmically.
- In many case, the solution is not so robust.
- Example: pack items of different sizes into backpacks of the same size, minimizing the empty space in the backpacks.
    1. Generate all the possible packing orders of the items.
    2. Try each of these orders.
    3. Select the order which needs the least number of the backpacks.
- This works only if there are not so many items. Why? How many different orders of packing can be generated?

Greedy algorithms

- In each step of the algorithm, the **best option** is selected among the available solutions (local optimum).
- The final result is not necessarily the best possible solution.
- Example: the previous problem of packing backpacks using the greedy approach
  1. Sort the items according to the decreasing order of the size.
  2. Pack items in this order until there is no room in a backpack or there are no items to be packed.
  3. If the current backpack is full and there are still items left, take the new backpack and go to Step 2.
- Sorting in the order is simpler than generating all the possible orders. However, the solution is not necessarily optimal.
- Other similar problems: generalized packing (both items and backpacks are of different sizes), pay with the least number of coins, route planning (e.g., from Lappeenranta to Helsinki), etc.

Dynamic programming

- Divide the task into parts and **save** the **results of subtasks** in the table of results (**intermediate results**) so you do not have to solve them again.
- For example (see next page), Fibonacci numbers can be computed fully recursively, or by saving the intermediate results for later computations.
- Save the intermediate results during computations so you do not have to recompute these results. Update them as needed according to the problem.
- Dynamic programming is very suitable to graph problems. A graph consists of nodes which are connected to each other directly or via one/more nodes.
- For example, the shortest paths in routing problems, e.g., using a map (a graph) of connected cities (nodes). The minimum distance from City A to B?

The minimum distance of each route from the start node to the end node is saved in the table. The minimum distances from the start node are being computed to all other nodes while going through the graph of connected nodes. Thus, each minimum is updated if the smaller distance appear. The solution is a mixture of the greedy algorithm and dynamic programming.

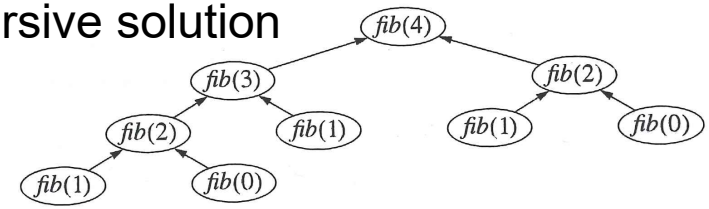
Fibonacci numbers: Dynamic programming


```
fib(n: INTEGER): INTEGER is
    do
        if n <= 1 then
            Result := n
        else
            Result := fib(n–1) + fib(n–2)
        end
    end
```

0 1 1 2 3 5 8

This is simple, elegant and obviously correct. Unfortunately, it has exponential time complexity (Exercise 2.7). Drawing a tree diagram of all the recursive calls the algorithm makes shows clearly what is slowing it down:

Recursive solution



The algorithm solves small instances many times over, and this effect becomes worse as n increases. Arrows are drawn pointing upwards rather than downwards for a reason that will become clear shortly.

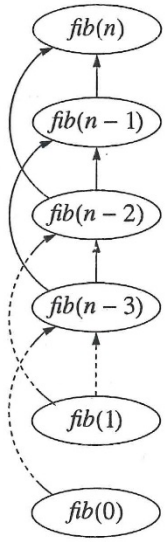
Whenever a divide-and-conquer algorithm solves small instances repeatedly in this way, *dynamic programming* may be used to eliminate the redundant work. The solutions are stored in a table, and an instance is solved from scratch only when it is encountered for the first time. Thereafter, whenever that solution is needed, it is simply retrieved from the table. For the Fibonacci problem, this yields

```
fib(n: INTEGER): INTEGER is
    do
        if table.item(n) = empty then
            if n <= 1 then
                table.put(n, n)
            else
                table.put(fib(n–1) + fib(n–2), n)
            end
        end;
        Result := table.item(n)
    end
```

Intermediate results in the table

where *table.item*(0..n) are initialized to the value *empty*, for example -1.

Even the test for *empty* can be eliminated by solving the instances in an order that guarantees that when a given instance's turn arrives, all the instances whose solution it needs have already been solved. To find this order, let the instances be vertices in a directed graph, and join x to y by an arrow if the solution to instance x is used when solving y. For the Fibonacci numbers problem, this gives



The correct ordering is then a *topological ordering* of the vertices (Section 11.4); in this case, *fib*(0), *fib*(1), *fib*(2), … , *fib*(n) or *fib*(1), *fib*(0), *fib*(2), … , *fib*(n).

It often happens that the ordering can be determined in advance and embedded into the algorithm. This can be done for the Fibonacci numbers problem:

```
fib(n: INTEGER): INTEGER is
    local
        i: INTEGER;
    do
        table.put(0, 0);
        table.put(1, 1);
        from i := 2 until i > n loop
            table.put(table.item(i–1) + table.item(i–2), i);
            i := i + 1
        end;
        Result := table.item(n)
    end
```

This algorithm has $O(n)$ complexity. The elimination of recursion is a further practical benefit: dynamic programming is a useful code optimization technique even when it does not reduce the asymptotic time complexity (Exercise 9.5).

The longest common subsequence

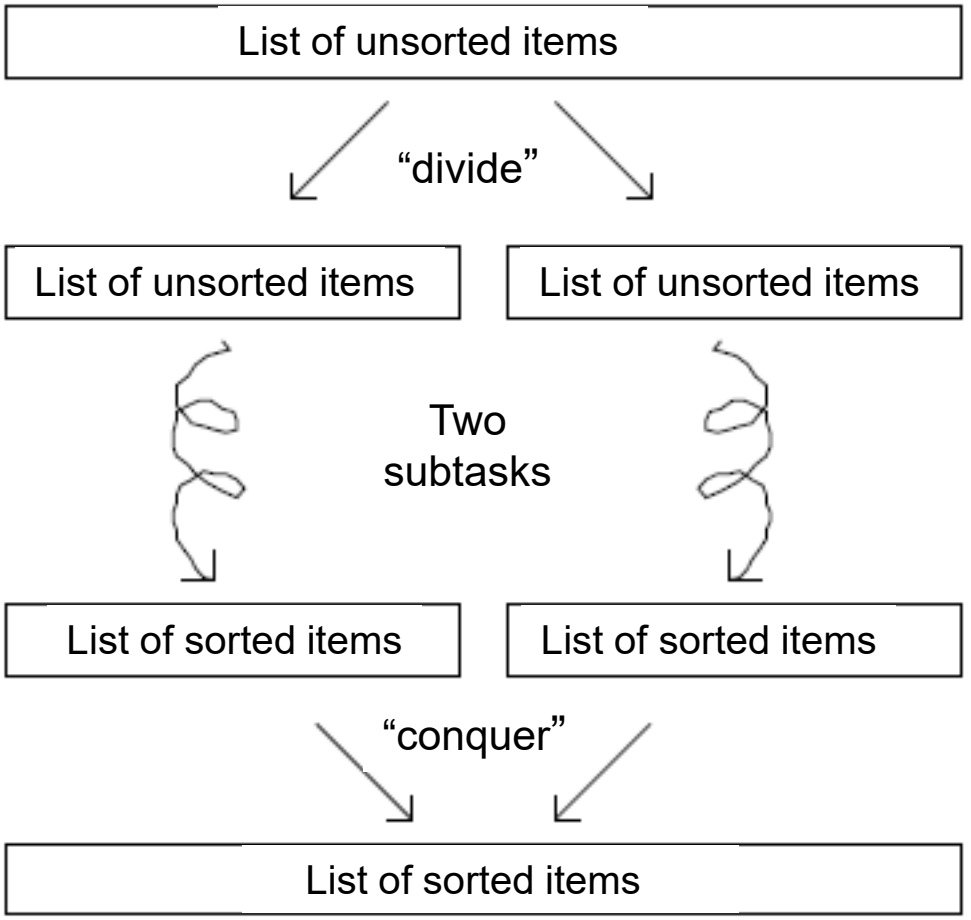
The following problem is typical of the way dynamic programming arises. Let x and y be two sequences of characters, for example x = *abdebcbb* and y = *adacbcb*.

E. Horowitz et al. Fundamentals of Data Structures in C. Computer Science Press, 1993.

Divide and conquer

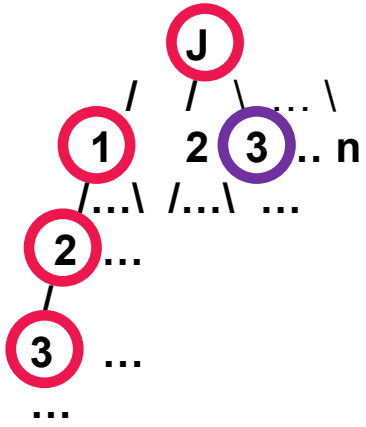
1. Divide the task into the subtasks.
2. Solve the subtasks individually.
3. Compose the whole solution from the solutions of the subtasks.

- The subtasks must be independent.
- The problem can be solved recursively if the subtasks are similar.
- Example: Sort the list using merge sorting where the unsorted list is being recursively divided into the smaller lists until the list contains one item only. Then the lists are merged (the sorted list).
- The goal is to obtain a more efficient algorithm. This is to be considered in the following lectures.



Search space methods

- Sometimes the solution cannot be found straightforwardly based on made choices. For example, local optimization can lead to a dead end.
- If there is the dead end where no solution is possible then:
  - Return to the previous selection situation (**backtracking**).
  - Select another option where to continue further.
- For example, relocate pieces of furniture in a room.
- Present as a **space tree** where options branch forward.
  - For example, all possible locations of pieces of furniture (1,2,3, ..., n), starting from the root node (J).



- The next node is usually selected by branching (**branch and bound**) based on the goodness criterion (greedy algorithm).
  - A piece of furniture 3 (in violet) is selected if it contains the best goodness value.
- This is not necessarily easy since these criteria are usually just estimates.
- For example: what move to make in chess in order to play optimally?

Problem conversion

- The type of the problem is changed to another type so it would be more efficient/possible to solve.
- Example: compute the product of x and y using logarithms, assuming that the operator of multiplication $*$ is *not available*

$$x \rightarrow \log x$$
$$y \rightarrow \log y$$
$$\log x + \log y$$
$$log^{-1}(\log x + \log y) = x * y$$

- Example: *efficient* digital image processing using Fourier transforms to sharpen/change contrast/etc. of a digital image.
  - Instead of spatial filtering operations on millions of image pixels, the Fourier transforms $\mathfrak{F}$ of the whole image i(x,y) and the filter h(x,y) are applied where the inverse Fourier transform of the product of these two transforms I(u,v) and H(u,v) is the filtered image g(x,y).

$$\mathrm{g(x,y)} = \mathfrak{F}^{-1}(I(u,v)H(u,v))$$

Randomized/approximate methods

- To be used when
  - the exact solution is *not needed*, or
  - the exact solution is *computationally impossible* to obtain.
- Selections during execution of an algorithm are based on randomized/approximate estimates.
- Backpacking using the greedy algorithm is an approximate algorithm.
- Computing the approximation of the square root using an iterative equation is a randomized algorithm.
  - The initial guess in needed.
- As compared to the optimal solution, it is good to know:
  - What is the error of the solution at maximum?
  - How much faster is the solution?

Summary

- There are several useful principles for the design of algorithms.
- The basic ones are recursion and iteration.
- Usually, the mixture of the principles are needed:
  - For example, a greedy algorithm & dynamic programming, or a search space method & a greedy algorithm.
- To make an efficient algorithm, select a suitable strategy according to the nature of the task to be solved and to fulfill requirements.
- If the exact solution is *not needed*, or the exact solution is *computationally impossible* to obtain, then instead of the optimal solution

  => randomized/approximate algorithms.