Block 4 (Montag 19.2.2024)

5 Info exercises

The exercises take place on-site, in room W1 0-008. There are computers with Linux. You can also use your own laptops with Linux if they are prepared to have Shell, Editor, Compiler, gnuplot, gdb, (preferentially) valgrind. It is not recommended to use powerful and sophisticated development environments, because they create a lot of overhead for just a small benefit.
You can participate as single persons or in two-person groups, two-person groups are recommended.
The exercises are "in-time": The exercise sheets and the material are provided right before start in StudIP. The exercises are completed during the course time and graded at the end of each exercise.
General scheme: the exercises are typically based on partially written programs. You have to complete the programs, only sometimes to write full programs. Then you have often to perform simulations, and sometimes analyse or plot the data.
Towards the end of an exercise, you have to present in front of the screen your own code to the teacher, as well as the results of the simulations. All group members have to contribute to the presentation. The grading will result in up to 10 points, all group members receive the same number of points.
Typically, the simulations rely on a correct code. Thus, in the *worst case*, you might ask the teacher to look into your code and he/she try to find mistakes, which means the teacher also works on the exercise, without guarantee of success. Naturally, this leads to the decrease of the number of points, even if all tasks are completed in the end! Clearly, you are always allowed to ask general questions about programming, how to look for mistakes etc, without reduction of the number of points.
There are 8 exercises, each lasting about 4h, from 14:15 until (ca.) 17:45, depending how long the grading takes.

6 Algorithms

VIDEO: `video04a_complexity_r`

6.1 Easy, hard and unsolvable problems

$t(x)$= run time of a program $\mathcal{P}$ with input x. In theoretical Computer Science: run time is evaluated for model computers (e.g. *Turing Machines*).

$n = |x|$= "size" of input (e.g. number of bits needed to code the problem). *Time complexity* of a program $\mathcal{P}$ = slowest (*worst case*) run time as function of n:

$$T(n) = \max_{x:|x|=n} t(x) \tag{1}$$

Example: Run time

$T(n)$	$T(10)$	$T(100)$	$T(1000)$
$f(n)$	23000	46000	69000
$g(n)$	1000	10000	100000
$h(n)$	100	10000	1000000

□

[Selbsttest]

Which program is (asymptotically) the slowest?

O notation
$T(n) \in O(g(n))$: $\exists c > 0, n_0 \geq 0$ with $T(n) \leq cg(n)\ \forall n > n_0$. "$T(n)$ is at most of order $g(n)$."

Typical time complexities:
Polynomial problems (class P) are considered as "easy", exponential problems as "hard". Some some important problems: no polynomial algorithm is *known*. So far, there is no proof that there are no polynomial algorithms. *NP-hard* problems, run actually in polynomial time on a *non-deterministic*

Table 1: Growth of functions dependent on the input size n.

$T(n)$	$T(10)$	$T(100)$	$T(1000)$
n	10	100	1000
$n \log n$	10	200	3000
n^2	10^2	10^4	10^6
n^3	10^3	10^6	10^9
2^n	1024	1.3×10^{30}	1.1×10^{301}
$n!$	3.6×10^6	10^{158}	4×10^{2567}

(model) computer.

Decision problems: Only output "yes"/"no". Example: Is the ground state energy of a system $\leq E_0$ (E_0: given parameter value)?

For some problems one can prove that they are *undecidable*, i.e., there is *no general* algorithm, which answers "yes" or "no" for all possible inputs=problem instances. But such problems are often provable, i.e. one can prove one of the possible answers, but not both.

- Halting problem: Does a given (arbitrary) program stop after a finite time for a given (arbitrary) input? (If it stops it is easy to prove: let it run and wait)
- Correctness problem: Does a given (arbitrary) program created the desired output for *all* possible inputs (If not, it is "easy" to prove: let it run for an input where the output is not correct)

There are (academic) functions which are even *not computable*. Proof by diagonalization, works like Cantor's proof that there are non-rational numbers.
VIDEO: `video04b_rekursion_r`

6.2 Recursion and Iteration

Principle of *Recursion*: Subroutines calls itself. Natural approach for recursive function definitions.

Example: factorial $n!$:

$$n! = \begin{cases} 1 & \text{if } n = 1 \\ n \times (n-1)! & \text{else} \end{cases} \tag{2}$$

Simple translation to C function

```
double factorial(int n)
{
  if(n==1)
    return(1.0);
  else
    return(n*factorial(n-1));
}
```

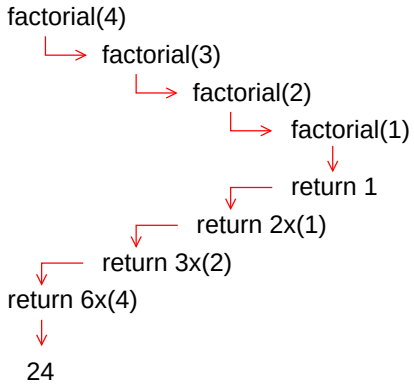

Figure 1: Hierarchy of recursive calls for the calculation of factorial(4).

Analysis of run time of `factorial()` using *recurrence equations*:

$$\tilde{T}(n) = \begin{cases} C & \text{for } n = 1 \\ C + \tilde{T}(n-1) & \text{for } n > 1 \end{cases} \tag{3}$$

Solve via differential equation: $\frac{d\tilde{T}}{dn} = \frac{\tilde{T}(n)-\tilde{T}(n-1)}{n-(n-1)} = C \rightarrow \tilde{T}(n) = CN + K$, with $\tilde{T}(1) = C \rightarrow$
Solution: $\tilde{T}(n) = Cn$, i.e. the run time is linear in n (fun fact: but exponentially in the length of input (=number of bits), because $n = 2^{\log_2 n}$).

[Selbsttest]

Propose an iterative calculation of the factorial by a loop. How is the time complexity?

VIDEO: `video04c_dvidide_and_conquer_r`

6.3 Divide-and-conquer (Divide and conquer)

[Selbsttest]

Think for yourself for 5 minutes about an algorithm (just the basic idea) which sorts the elements $\{a_0, \ldots, a_{n-1}\}$ "in increasing order".
Next, discuss your solution for five minutes with your bench neighbor.
What asymptotic run time $T(n)$ do you expect for your algorithm?

ATTENTION: Do not read on before you have thought about the problem!

Here: Principle (relies on recursion):

1. reduce problem to some smaller sub problems
2. solve all sub problems
3. construct the solution of the problem by using the solutions of the sub problems.

Example: Sorting of a set of elements $\{a_0, \ldots, a_{n-1}\}$ by *Mergesort*.
Basic idea: divide set into two sets of equal size, sort both sets and merge them such that the result is sorted.

```
/** sorts 'n' integer numbers in the array 'a' in ascending **/
/** order using the mergsort algorithm                      **/
void my_mergesort(int n, int *a)
{
  int *b,*c;                                    /* two arrays */
  int n1, n2;                      /* sizes of the two arrays */
  int t, t1, t2;                            /* (loop) counters */

  if(n<=1)                           /* at most one element ? */
    return;                                   /* nothing to do */
  n1 = n/2; n2 = n-n1;                /* calculate half sizes */

  /* array a is distributed to b,c. Note: one could do it   */
  /* using one array alone, but yields less clear algorithm */
  b = (int *) malloc(n1*sizeof(int));
  c = (int *) malloc(n2*sizeof(int));
  for(t=0; t<n1; t++)                              /* copy data */
      b[t] = a[t];
  for(t=0; t<n2; t++)
      c[t] = a[t+n1];

  my_mergesort(n1, b);                /* sort two smaller arrays */
  my_mergesort(n2, c);


  t1 = 0; t2 = 0;     /* assemble solution from subsolutions */
  for(t=0; t<n; t++)
    if( ((t1<n1)&&(t2<n2)&&(b[t1]<c[t2]))||
        (t2==n2))
      a[t] = b[t1++];
    else
      a[t] = c[t2++];

  free(b); free(c);
}
```

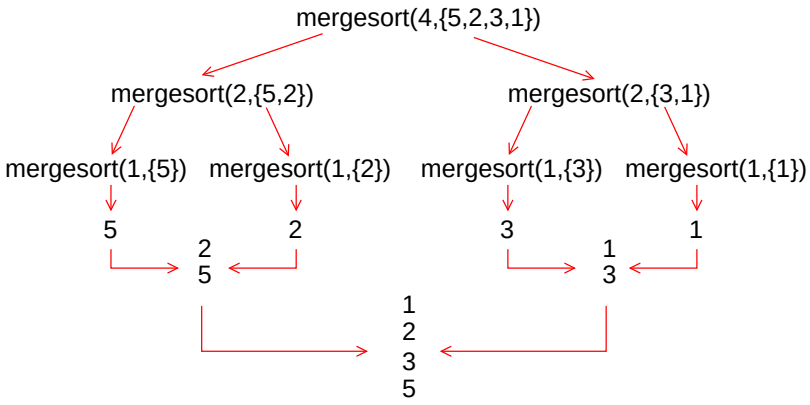

Figure 2: Aufruf von `mergesort`$(4, \{5, 2, 3, 1\})$.

Run time : division of set and reassemble: $O(n)$; recursive calls: $T(n/2)$.
Recurrence:

$$T(n) = \begin{cases} C & (n = 1) \\ Cn + 2T(n/2) & (n > 1) \end{cases} \tag{4}$$

Solution for large n: $T(n) = \frac{C}{\log 2} n \log n$. Proof by insertion

$$\begin{aligned} T(2n) &= C2n + 2T(2n/2) \\ &= C2n + 2(\frac{C}{\log 2} n \log n) \\ &= \frac{C}{\log 2} 2n \log 2 + \frac{C}{\log 2} 2n \log n \\ &= \frac{C}{\log 2} 2n \log(2n) \end{aligned}$$

VIDEO: `video04d_dynamic_programming_r`

6.4 Dynamic Programming

Fibonacci numbers:

$$\text{fib}(n) = \begin{cases} 1 & (n = 1) \\ 1 & (n = 2) \\ \text{fib}(n-1) + \text{fib}(n-2) & (n > 2) \end{cases} \tag{5}$$

E.g. $\text{fib}(4) = \text{fib}(3) + \text{fib}(2) = (\text{fib}(2) + \text{fib}(1)) + \text{fib}(2) = 3$,
$\text{fib}(5) = \text{fib}(4) + \text{fib}(3) = 3 + 2 = 5$.
Grows very quickly: $\text{fib}(10) = 55$, $\text{fib}(20) = 6765$, $\text{fib}(30) = 83204$, $\text{fib}(40) > 10^8$

Number of calls for a recursive implantation grows also exponentially with n, even faster than the function itself!

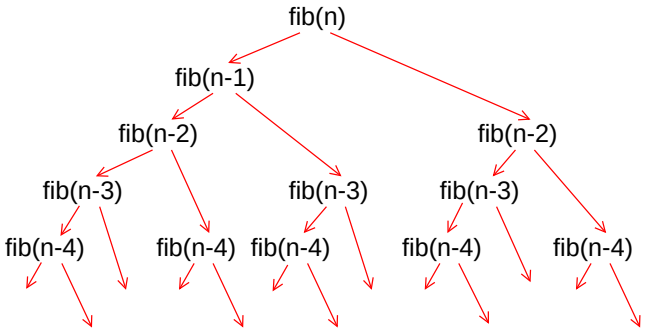

Figure 3: Hierarchie der Aufrufe für fib(n).

[Selbsttest]

How can you find a faster algorithm?

ATTENTION: Do not read on before you have thought intensively about the problem

Better: dynamic programming. Principle: calculate solutions for smaller problems, store them, and use them iteratively (not recursively) to obtain solutions for larger problems.

```
/** calculates Fibonacci number of 'n' dynamically **/
double fib_dynamic(int n)
{
  double *fib, result;
  int t;
  if(n<=2)                                    /* simple case ? */
    return(1);                          /* return result directly */
  fib = (double *) malloc(n*sizeof(double));
  fib[1] = 1.0;                       /* initialise known results */
  fib[2] = 1.0;

  for(t=3; t<n; t++)          /* calculate intermediate results */
    fib[t]=fib[t-1]+fib[t-2];

  result = fib[n-1]+fib[n-2];
  free(fib);
  return(result);
}
```

[Selbsttest]

What is the run time of the algorithm?
Competition: what is the largest value n, such that fib(n) results in a a finite output value: each one has to give an estimation.

Even faster: Formula

$$\text{fib}(n) = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right) \tag{6}$$