A practical guide to computer simulation II

Alexander K. Hartmann, University of Göttingen

May 21, 2003

4 Algorithms

Some background on theoretical computer science: better understanding what is possible and what not, what is fast and what is slow.
Literature: [1, 2, 3]

Some general programming techniques helping to speed up programs.
Literature: [4, 5, 6]

4.1 Easy/hard/undecidable problems problems

$t_{\mathcal{P}}(x)$= running time of program $\mathcal{P}$ with input x. In theoretical computer science: Running times studied on model computers (e.g. *Turing machines*).
$n = |x|$= "size" of input (e.g. number of bits to code problem). *Time complexity* of program $\mathcal{P}$ = worst case running time as a function of n.

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

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 of order at most $g(n)$."

Typical time complexities:

Table 1: Growth of functions as a function of 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}

Polynomial problems (class P) are called "easy", exponential problems are called "hard". For some problems no polynomial algorithm is known, but no proof so far that there is no polynomial algorithm (NP-hard problems, the run polynomially on a *nondeterministic* (model) computer).

Some problems can be proven to be *undecidable*, i.e. *no* general algorithm the problem answering "yes" or "no" in all possible cases

- halting problem: will a given program halt with a given input?
- correctness problem: does a program produce the desired output for *all* inputs?

There are some functions which cannot be computed at all: proof using diagonalization.

4.2 Recursion and Iteration

Principle of *recursion*: Subroutine call itself to solve problem. Natural for recursive definitions.
Example: factorial $n!$:

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

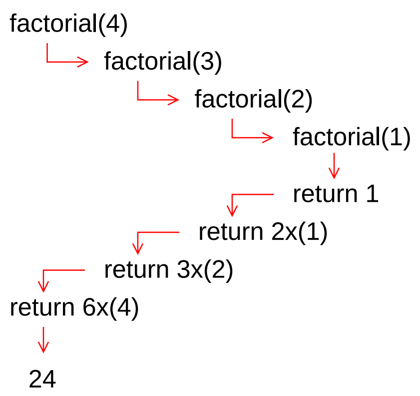

Figure 1: Hierarchy of recursive calls for calculation of factorial(4).

Analysis of running time using *recurrence equation*.

$$\tilde{T}(n) = \begin{cases} C & \text{for } n = 1 \\ C + \tilde{T}(n-1) & \text{for } n > 1 \end{cases} \tag{2}$$

Solution: $\tilde{T}(n) = Cn$, i.e. running linear in n but exponentially in the length of the input (=number of bits)!
For all recursive programs: equivalent iterative program:

```
double factorial2(int n)
{
  double result;
```

```
  result=1;
  while(n>0)
  {
    result *= n--;
  }
  return(result);
}
```

Here: $O(n)$ running time obvious.

4.3 Divide-and-conquer

Principle:

1. reduce problem to some smaller problems
2. solve smaller problems
3. assemble solution of large problem from solutions of small problems

Example: Sort set of elements $\{A_1, \ldots, A_n\}$ using *mergesort*.
Basic idea: part set into two sets of equal size, sort both and finally merge them.

algorithm mergesort(n,$\{A_1, \ldots, A_n\}$)
begin
    if $n = 1$ **then**
        return
    for $i := 1, 2, \ldots, n/2$ **do**   **comment** divide set
    begin
        $B_i := A_i$;
        $C_i := A_{i+n/2}$;
    end
    mergesort($n/2, \{B_1, \ldots, B_{n/2}\}$);   **comment** sort subsets
    mergesort($n/2, \{C_1, \ldots, C_{n/2}\}$);
    $x := 1$; $y := 1$;   **comment** largest elements of sequences
    for $i := 1, \ldots, n$ **do**   **comment** merge sorted subsets
        if $x \leq n/2$ AND $B_x < C_y$ **then**
            $A_i := B_x$; $x := x + 1$;
        else
            $A_i := C_y$; $y := y + 1$;
end

Running time: Division of set, merge: $O(n)$, recursive calls $T(n/2)$. Recurrence:

$$T(n) = \begin{cases} C & (n = 1) \\ Cn + 2T(n/2) & (n > 1) \end{cases} \tag{3}$$

Solution for large n: $T(n) = C/log2n \log n$

4.4 Dynamic programming

Fibonacci numbers:

$$\mathrm{fib}(n) = \begin{cases} 1 & (n=1) \\ 1 & (n=2) \\ \mathrm{fib}(n-1)+\mathrm{fib}(n-2) & (n>2) \end{cases} \tag{4}$$

Functions + number of calls in a recursive implementation grows exponentially with n.

Better: *dynamic programming.* Principle: calculate solutions for small problems, solve them and use results to solver larger problems iteratively.

```
double fib_dynamic(int n)
{
  double *fib, result;
  int t;
  if(n<=2)
    return(1);
  fib = (double *) malloc(n*sizeof(double));
  fib[1] = 1.0;
  fib[2] = 1.0;

  for(t=3; t<n; t++)
    fib[t]=fib[t-1]+fib[t-2];

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

Running time $O(n)$.

4.5 Backtracking

If no direct way to obtain solutions: try different possibilities.

Backtracking: previously made decisions turns out to be infeasible: take decision back and try different way.

Example: N queens problem

N queens are to be placed on an $N \times N$ chess board in such a way that no queen checks against any other queen.

This means that in each row, in each column and in each diagonal at most one queen is placed. □

Idea of algorithm: place per column (Q_i, $Ii = 0, \ldots, N-1$) one queen in a systematic way. If no solution is found, track back.

algorithm queens$(n, N, \{Q_i\})$
begin
    if $n = -1$ **then**
        print array $Q_0, \ldots, Q_{N-1}$; **comment** problem solved
    for $i := 0, \ldots, N-1$ **do**
    begin
        $Q_n := i$; **comment** place queen in column n and row i
        if Queen n does not check against any other **then**
            queens$(n-1, N, \{Q_i\})$;
    end
    $Q_n := 0$;
    return
end

Initially: $Q_i = 0$ for all i and call queens$(N, N, \{Q_i\})$.

Execution for $N = 4$:

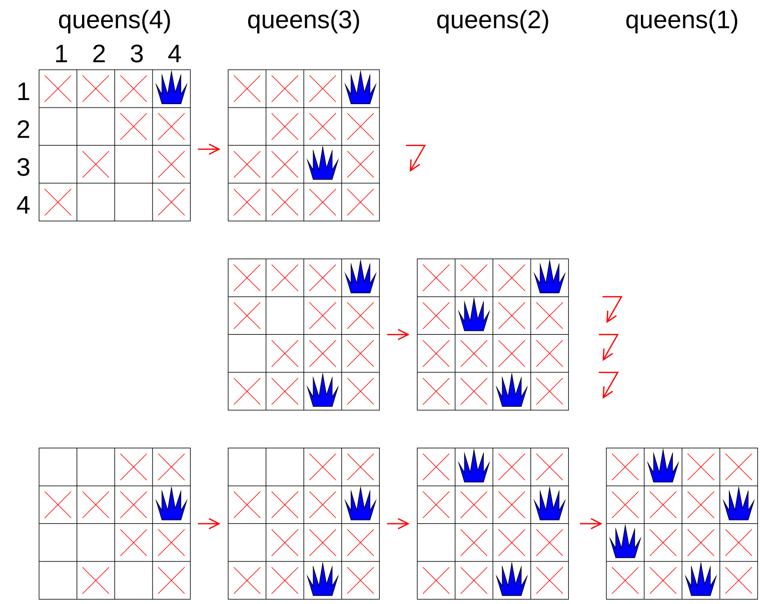

Figure 2: How the algorithm solves the 4-queens problem.

4.6 Excercise

Write a program for the N-queen problem. The input parameter should be N. The program should print *all* solutions it finds. **Hint**: In addition to an array for the columns Q_i, use additional arrays for all rows and diagonals, which are set to 1 if a queen is placed in this row/diagonal. Use a 2d array `board[][]` for printing a solution.

References

[1] P. Papadimitriou, *Computational Complexity*, (Addison-Wesley, Reading, MA, 1994)

[2] M Sipser, *Introduction to the Theory of Computation*, (PWS, Boston, MA, 1997)

[3] U. Schöning, Theoretische Informatik kurzgefasst, (Spektrum Akademischer Verlag, Heidelberg,Berlin 1999)

[4] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *The design and analysis of computer algorithms*, (Addison-Wesley, Reading (MA) 1974)

[5] R. Sedgewick, *Algorithms in C*, (Addison-Wesley, Reading (MA) 1990)

[6] T.H. Cormen, S. Clifford, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, (MIT Press 2001)