ALGORITHMS. Seminar 6: Variants of binary search. Applications of merging. Quick selection. Applications of the greedy technique.

Problem 1 (S+L) Let $a[1..n]$ be an increasingly sorted array and v a value. Write an algorithm of complexity $\mathcal{O}(\lg n)$ which finds the position where the value v can be inserted in $a[1..n]$ such that it remains increasingly sorted.

Solution. The search process can be described as follows:

```
search(real a[1..k], v)
li ← 1; ls ← k
if (v ≤ a[li]) return(li) endif
if (v ≥ a[ls]) return(ls + 1) endif
while (ls > li)
    m ← (li + ls)/2
    if (a[m] = v) then return(m + 1) endif
    if (v < a[m]) then ls ← m − 1
        else li ← m + 1
    endif
endwhile
if (v ≤ a[li]) then return(li)
if (v > a[ls]) then return(ls + 1)
```

The correctness can be proved by using as invariant property the statement $a[li-1] \leq v \leq a[ls+1]$ (under the formal assumption that $a[0] = -\infty$ and $a[k+1] = \infty$). The fact that the complexity order is $\mathcal{O}(\lg n)$ can be proved as in the case of binary search. Another variant of the algorithm is:

```
search(real a[1..k], v)
li ← 1; ls ← k
while (li ≤ ls)
    if (v ≤ a[li]) return(li) endif
    if (v ≥ a[ls]) return(ls + 1) endif
    m ← (li + ls)/2
    if (a[m] = v) then return(m + 1) endif
    if (v < a[m]) then ls ← m − 1
        else li ← m + 1
    endif
endwhile
return(li)
```

In this case the statement $a[li-1] \leq v \leq a[ls+1]$ is also a loop invariant and at the end of the loop (when $li = ls + 1$) one have that $a[li-1] \leq v \leq a[li]$. Thus the value which should be returned is li.

By using in insertion sort this algorithm for searching the insertion position in one obtains:

```
binary_insertion_sort(real a[1..n])
for i ← 2, n do
    aux ← a[i]
    poz ←search(a[1..i − 1],v)
    for j ← i − 1, poz, −1 do a[j + 1] ← a[j] endfor
    a[poz] ← aux
endfor
return a[1..n]
```

By taking into account only the comparisons the insertion sort combined with binary search belongs to $\mathcal{O}(n \lg n)$. If we take into account the number of assignments involving elements of a then the algorithm will belong to $\mathcal{O}(n^2)$.

Problem 2 (S) *Ternary search.* The binary search technique can be extended by dividing an array $a[li..ls]$ in three subarrays $a[li..m1]$, $a[m1+1..m2]$, $a[m2+1..ls]$, where $m1 = \lfloor (2li+ls)/3 \rfloor$ and $m2 = \lfloor (li+2ls)/3 \rfloor$. The search algorithm becomes:

```
ternary_search (integer a[1..n], v)
li ← 1; ls ← n;
while (li <= ls)
    m1 ← (2li + ls)DIV3; m2 ← (li + 2ls)DIV3
    if(x[m1] = v) then return(m1) endif
    if(x[m2] = v) then return(m2)endif
    if(v < x[m1]) then ls ← m1 − 1
    else if (v < x[m2])then li ← m1 + 1;ls ← m2 − 1;
        else li ← m2 + 1 endif
    endif endwhile
return(-1)
```

If $T(n)$ denotes the number of comparisons then the following recurrence relation is satisfied (in the particular case when $n = 3^k$):

$$T(n) = \begin{cases} 0 & n = 0 \\ T(n/3) + c & n > 0 \end{cases}$$

where $c \in \{1, 2, 3, 4\}$ is the number of comparisons made in one loop. By applying the backward substitution for the particular case when $n = 3^k$ one obtains that $T(n) = c \log_3 n$ thus $T(n) \in \mathcal{O}(\log n)$, which is the same complexity order as in the case of binary search (only the multiplicative constants are different).

Problem 3 (S+L) *The bisection method.* Let $f : [a, b] \to R$ be a continuous function having the following properties: (i) $f(a)f(b) < 0$; (ii) there exists a unique x^* having the property $f(x^*) = 0$. Find an approximation of x^* with accuracy $\epsilon > 0$.

Solution. Finding an approximation of x^* with accuracy ϵ means to identify an interval of length ϵ which contains x^* (or an interval of length 2ϵ if we consider as approximation of x^* the middle point of the interval). One can apply the same strategy as in the case of binary search by taking into account the fact that x^* belongs to the interval for which the function f has values of opposite signs at the extremities.

```
bisection(real a,b,epsilon)
li ← a; ls ← b
repeat
    m ← (li + ls)DIV2
    if f(m) = 0 then return m endif
    if f(m) ∗ f(li) < 0 then ls ← m
        else li ← m
    endif
until |ls − li| < 2ε
return (li + ls)DIV2
```

The complexity of the algorithm is related to the size of the interval $[a, b]$ and to the desired accuracy, ϵ. By using the notation $n = (b - a)/\epsilon$ one obtains that the algorithm belongs to $\mathcal{O}(\lg n)$.

Problem 4 (S+L) Let $a[1..m]$ and $b[1..n]$ be two strictly increasing arrays. Construct a strictly increasing array which contains the distinct elements from a and b.

Solution. We apply the merging technique by comparing the element to be added with the last element in the new array. The element is added only if it is different from the last transferred element.

```
merging(integer a[1..m],b[1..n])
integer i,j, k, c[1..k]
i ← 1; j ← 1; k ← 0
if a[i] < b[j] then k ← k + 1; c[k] ← a[i]; i ← i + 1
        else if a[i] > b[j] then k ← k + 1; c[k] ← b[j]; j ← j + 1
            else k ← k + 1; c[k] ← a[i]; i ← i + 1; j ← j + 1
        endif
endif
while i ≤ m AND j ≤ n do
    if a[i] < b[j] then
        if a[i] ≠ c[k] thenk ← k + 1; c[k] ← a[i]; i ← i + 1
        else i ← i + 1 endif
    else if a[i] > b[j] then
        if b[j] ≠ c[k] thenk ← k + 1; c[k] ← b[j]; j ← j + 1
        else j ← j + 1 endif
        if a[i] ≠ c[k] thenk ← k + 1; c[k] ← a[i]; i ← i + 1;j ← j + 1
        else i ← i + 1;j ← j + 1 endif
        endif
    endif
endwhile
while i ≤ m do
    if a[i] ≠ c[k] thenk ← k + 1; c[k] ← a[i]; i ← i + 1
    else i ← i + 1 endif
endwhile
while j ≤ n do
    if b[j] ≠ c[k] thenk ← k + 1; c[k] ← b[j]; j ← j + 1
    else j ← j + 1 endif
endwhile
return c[1..k]
```

The complexity order of the algorithm is as in the case of plain merging, i.e. $\mathcal{O}(m+n)$.

Problem 5 (S) Let us consider two integer values given by the arrays corresponding to their decompositions in prime factors (each number has associated an array of prime factors and an array of corresponding powers). Construct the arrays of prime factors and powers corresponding to the least common multiplier.

Solution. Let us consider the values: $3415 = 3 \cdot 5^3 \cdot 7 \cdot 13$ and $966280 = 2^3 \cdot 5 \cdot 7^2 \cdot 17 \cdot 29$. The first value is associated with the following arrays: $(3, 5, 7, 13)$ and $(1, 3, 1, 1)$ and the second value can be described by the arrays: $(2, 5, 7, 17, 29)$ and $(3, 1, 2, 1, 1)$. The arrays corresponding to the least common multiplier will be:

- *prime factors:* the array will contain all prime factors corresponding to both number and it can be obtained by applying the merging taking such that the final array will contain only distinct values.
- *powers:* the array will contain for each prime factor the maximum of the corresponding powers in the arrays associated with the initial values.

In the case of the above example the corresponding arrays will be: $(2, 3, 5, 7, 13, 17, 29)$ and $(3, 1, 3, 2, 1, 1, 1)$.

```
lcm(integer fa[1..m],pa[1..m],fb[1..n],pb[1..n] )
integer i,j, k, fc[1..k],pc[1..k]
i ← 1; j ← 1; k ← 0
while i ≤ m AND j ≤ n do
    if fa[i] < fb[j] then
        k ← k + 1; fc[k] ← fa[i]; pc[k] ← pa[i]; i ← i + 1
    else if a[i] > b[j] then
        k ← k + 1; fc[k] ← fb[j]; pc[k] ← pb[j]; j ← j + 1
            else k ← k + 1; fc[k] ← fa[i]; pc[k] ← max(pa[i], pb[j]); i ← i + 1;j ← j + 1
        endif
    endif
endwhile
while i ≤ m do
    k ← k + 1; fc[k] ← fa[i]; pc[k] ← pa[i]; i ← i + 1
endwhile
while j ≤ n do
    k ← k + 1; fc[k] ← fb[j]; pc[k] ← pb[j]; j ← j + 1
endwhile
return fc[1..k], pc[1..k]
```

Problem 6 (S) *Selection of kth element.* Let $a[1..n]$ be an array, not necessarily sorted. Find the kth element of a in increasing order (for $k = 1$ one obtains the minimum, for $k = n$ one obtains the maximum, for $k = n/2$ one obtains the median etc.).

Solution. A first variant to solve the problem consists of partially sorting the array, by using the selection sort, until when on the first k positions are placed the right values. The number of operations executed in this case is of order $\Theta(kn)$. When k is at least $n/2$ one obtains an algorithm of quadratic complexity. By sorting the entire algorithm using quicksort we obtain an algorithm of complexity $\mathcal{O}(n \log n)$ in the average case.

A better algorithm can be obtained by applying the partitioning idea from quicksort but without sorting the array. The aim of partitioning is to divide the array in two subarrays such that all elements in the left subarray are smaller than the elements in the right subarray. Such a partitioning algorithm is:

```
partitioning(integer a[li..ls])
v ← a[li]; i ← li − 1; j ← lj + 1;
while i < j do
    repeat i ← i + 1 until a[i] ≥ v
    repeat j ← j − 1 until a[j] ≤ v
    if i < j then a[i] ↔ a[j] endif
endwhile
return j
```

If the partitioning position is q and $k \leq q - li + 1$ then the problem can be reduced to the selection of kth element of the subarray $a[li..q]$. If $k > q - li + 1$ then the problem can be reduced to the selection of the $k - (q - li + 1)$th element from the subarray $a[q + 1..ls]$. The value of k will always be between 1 and the number of elements in the (sub)array (when $li = ls$ the value of k will be 1). The algorithm can be described as follows:

```
selection(integer a[li..ls], k)
if li = ls then return a[li]
else
    q ←partitioning(a[li..ls])
    r ← q − li + 1
    if k ≤ r then selection(a[li..q], k)
        else selection(a[q + 1..ls], k − r)
    endif
endif
```

In the best case the partitioning is balanced (the subarrays have almost the same number of elements). Since the partitioning algorithm is of linear complexity it follows that the number of comparisons satisfies the following recurrence relation:

$$T(n) = \begin{cases} 0 & n = 1 \\ T(n/2) + n & n > 1 \end{cases}$$

By applying the master theorem (for $m = 2$, $d = 1$, $k = 1$) one obtains that in the best case the algorithm is of linear complexity. In the worst case the partitioning would be extremely unbalanced at each step (one subarray would have only one element while the other elements would be contained in the second subarray). The recurrence relation is in this case:

$$T(n) = \begin{cases} 0 & n = 1 \\ T(n-1) + n & n > 1 \end{cases}$$

which leads to a quadratic complexity. As in the case of quicksort this quick selection algorithm behaves in the average case as in the best case, i.e. it has a linear complexity.

Problem 7 (S+L) *Selection of activities.* Let us consider a set of activities which all share the same resource but at a given moment only one activity can use the resource (e.g. the activity is an exam and the resource is a room). Let us suppose that for each activity, A_i, the starting and the ending moments (p_i and t_i, respectively) are known and the activity is executed during the time interval $[p_i, t_i)$. Two activities, A_i and A_j are *compatible* if their corresponding intervals are disjoint. Find a maximal subset of compatible activities.

Solution. A solution of this problem consists of a subset of activities $S = (a_{i_1}, \ldots, a_{i_m})$ which satisfy $[p_{i_j}, t_{i_j}) \cap [p_{i_k}, t_{i_k}) = \emptyset$ for any $j \neq k$. The greedy technique can be applied by using one of the following selection criteria: (i) the smallest duration of an activity; (ii) the smallest starting time of an activity; (iii) the smallest ending time of an activity; (iv) the activity for which the number of activities compatible with it is maximal.

The selection criterion which satisfies the greedy choice property leading to an optimal solution is the third one: at each step one chooses the activity which ends the earliest.

If we denote by $A[i].p$, $A[i].t$ the starting and the ending moment of activity $A[i]$, the algorithm can be described as follows:

```
selection_activities(A[1..n])
A[1..n] ← increasing_sort(A[1..n]) // sorting on the ending time
S[1] ← A[1]
i ← 2; k ← 1
while i ≤ n do
    if S[k].t ≤ A[i].p
        then k ← k + 1; S[k] ← A[i]; endif
    i ← i + 1
return S[1..k]
```

We can prove that this algorithm leads to the optimal solution. By increasingly sorting the activities by the ending time we obtain: $t_1 \leq t_2 \leq \ldots \leq t_n$. Let us consider an optimal solution $O = ((p_{i_1}, t_{i_1}), \ldots, (p_{i_m}, t_{i_m})$. It is obvious that $t_{i_1} \geq t_1$. Thus by replacing A_{i_1} with A_1 in O, one obtains a solution with the same number of compatible activities but the first one is chosen based on the greedy choice. Thus the problem satisfies the greedy choice property. Once we chose the first activity, the problem is reduced to the optimally selecting the activities which are compatible with the first one. If O is the optimal solution of the initial problem then $O' = ((p_{i_2}, t_{i_2}), \ldots, (p_{i_m}, t_{i_m})$ is an optimal solution for the problem to which we arrive after the first activity has been chosen. Thus the problem also satisfy the optimal substructure property.

Let us consider the case $A = ((1,8),(3,5),(1,4),(4,6))$. By sorting A in increasing order based on the activity duration one obtains that the first selected activity should be: $S = ((3,5))$ or $S = ((4,6))$. No other activity can be selected.

By sorting the activities increasingly by the starting time, the first selected activity would be $((1,8))$ or $((1,4))$ leading to the solution $S = ((1,4),(4,6))$. The same solution is obtained when the selection criterion is the ending time.

Problem 8 (S+L) *Activities scheduling.*

Let us consider a set of n tasks which should be executed by the same processor. All tasks have the same duration (e.g. 1 time unit). Each task has a deadline and if it executed it leads to a given profit. For the task i, let us denote by $t_i \leq n$ the deadline and by p_i the corresponding profit. Find a scheduling (a starting time for the tasks) of the tasks such that the total profit is maximal.

Solution. A schedule is a sequence $S = (s_1, s_2, \ldots, s_n)$, where $s_i \in \{1, \ldots, n\}$ denotes the index of the task scheduled at moment i. The greedy technique applied to solve the problem is characterized by:

- the tasks are sorted decreasingly by their profit;
- each task is scheduled in a free time slot before the deadline as close as possible to the deadline.

```
schedule(A[1..n])
A[1..n] ← sort(A[1..n]) // Decreasingly sorting by profit (A[i].p)
for i ← 1, n do S[i] ← 0 endfor
for i ← 1, n do
    poz ← A[i].t // deadline of task i
    while S[poz] ≠ 0 ∧ poz ≥ 1 do poz ← poz − 1 endwhile
    if poz ≠ 0 then S[poz] ← i endif // the task is scheduled
    // otherwise the task i cannot be scheduled
return S[1..n]
```

We can remark that if for each $i \in \{1, \ldots, n\}$ the number of tasks having the deadline smaller than i is at most i ($\text{card}\{j | t_j \leq i\} \leq i$) then all tasks can be scheduled, otherwise there will be unscheduled activities.

Let us consider $n = 4$ having the deadlines: $(2,3,4,2)$ and the corresponding profits $(4,3,2,1)$. By applying the greedy technique we will obtains $(4,1,2,3)$. If the deadlines would be: $(2,4,1,2)$ then the greedy solution will be $(3,1,0,2)$ but the task 4 is not scheduled.

Problem 9 (S) Let A be a set of natural values. Find a partition (B, C) of A such that $A = B \cup C$, $B \cap C = \emptyset$ and the sum of elements in B is as close as possible to the sum of elements in C.

Solution. The elements of A are decreasingly sorted. Then the elements in A are successively transferred in the subset having the smaller sum.

```
partition(integer a[1..n])
integer k1, k2, b[1..k1], c[1..k2], S1, S2, i
k1 ← 1; b[k1] ← a[1]; S1 ← a[1]; k2 ← 0; S2 ← 0
a[1..n] ← sort(a[1..n])
for i ← 2, n
    if S1 < S2 then k1 ← k1 + 1; b[k1] ← a[i]; S1 ← S1 + a[i];
    else k2 ← k2 + 1; c[k2] ← a[i]; S2 ← S2 + a[i];endif
return b[1..k1], c[1..k2]
```

The greedy technique does not always lead to an optimal solution but only to a suboptimal one. By applying the greedy technique to the set $\{10, 8, 7, 5, 4, 2\}$ one obtains $A = \{10, 5, 4\}$ and $B = \{8, 7, 2\}$ while an optimal solution would be $A = \{10, 8\}$ and $B = \{7, 5, 4, 2\}$.

Homework

1. Write a recursive version for the algorithm described in Exercise 1.
2. Let $a[1..m]$ and $b[1..n]$ be two strictly increasing arrays. Describe an algorithm of linear complexity to construct the array containing only the elements which are in both arrays.
3. Let us consider two natural numbers given by their factorization (as in Exercise 5 above). Construct the corresponding factorization of the greatest common divisor of those two values.
4. Write an algorithm of complexity $\mathcal{O}(n \log n)$ to check if the elements in an array are all distinct.
5. Let $\{x_1, x_2, \ldots, x_n\}$ be a set of real values. Find a minimal set of intervals of length 1 which contain all elements of the initial set.