CS3001, Algorithm Design and Analysis
Sample solution

12) Assume first that all entries are positive. Then we can proceed like in the sum subsequence case. The only difference is that an empty product is 1 (and not 0) and that we disregard suffixes which are smaller than 1.
Things get more complicated if negative entries arise. As the product of two negative numbers is positive, we may not disregard these. Instead we carry two suffixes, a maximal positive one and a maximal negative one. A maximal subsequence product from i to j means that either $x[j] > 0$ and i to $j-1$ gives a maximal positive suffix product or $x[j] < 0$ and i to $j-1$ gives a maximal negative suffix product. Thus, when running over the entries in x, depending on $x[i]$ three situations are possible:

$x[i] > 0$ We multiply both suffixes with $x[i]$.

$x[i] < 0$ We multiply both suffixes with abs$(x[i])$ and then swap their rôle.

$x[i] = 0$ An sequence entry zero will "stop" all sequences. Both suffixes (but not the largest value so far) must be reinitialized.

Afterwards we test whether the positive suffix is larger than the maximum computed so far. If yes, we update this maximum.
A positive suffix < 1 again can be ignored – it will only diminish a result.
However a negative suffix must be considered even if of very small absolute value, as it might give a valuable sign: Consider $-1/2, -10000$, here we want $-1/2$ to form a negative suffix.
Finally, initialization has a further problem: The product over an empty sequence is not negative. Therefore the negative suffix is initialized to 0, so when multiplying with any number it remains zero and so never will become a valid positive suffix.
This gives the following algorithm (again only the value is computed. One would need further index variables to get the subsequence start and end positions. This is left as an exercise):

```
begin
  $MaxVal := 1$;$MaxPos := 1$;$MaxNeg := 0$;
  for $i \in [1..n]$ do
    if $x[i] > 0$ then
      $MaxPos := MaxPos \cdot x[i]$;
      $MaxNeg := MaxNeg \cdot x[i]$;
    elif $x[i] < 0$ then
      $MaxNeg := -MaxPos \cdot x[i]$;
      $MaxPos := -MaxNeg \cdot x[i]$; {if $MaxNeg = 0$ will cause empty positive suffix}
    else
      $MaxPos := 1$;$MaxNeg := 0$;
    fi;
    if $MaxPos < 1$ then $MaxPos := 1$; fi;
    if $MaxPos > MaxVal$ then $MaxVal := MaxPos$ fi;
  od;
end
```

Remark 1: Some solutions tried to work with only one suffix (absolute value) and a flag on whether the suffix was positive or negative. This however creates problems with sequences like

$$2, -3, -3 \qquad \text{and} \qquad -3, 2, 3$$

In the first case the largest product is $-3 \cdot -3 = 9$, in the second case $2 \cdot 3 = 6$. However when processing the last entry, the maximum absolute suffix is both times 6 with negative sign.
(I believe — though cannot prove — that one *must* use two suffix variables.)
Remark 2:A substantial part of the solutions covered the *sum* subsequence problem again. Unfortunately, a nice solution is still wrong, if it solves only a different problem . **Read the exercise text in full!**

13) a) Sort first (heapsort uses $O(n \log n)$ steps), then use part b).
Remark 1: Quicksort is not $O(n \log n)$, but only *average case* $O(n \log n)$.
Remark 2: An algorithm that runs over all elements x_i and for every element considers all other elements will be $O(n^2)$ or worse!

15) a) The simplest example is 1,1,2; $k = 2$, in which the algorithm will find the solution $1 + 1$ (give a table of the intermediate values computed by the algorithm). Two more elaborate examples: 7,4,2,2,1; $k = 8$ shows that sorting the entries does not help, 4,3,3,1,1; $k = 6$ shows that it is not sufficient to use the largest element.
b) The knapsack algorithm as given will use a solution with fewer elements and then will not consider extending smaller solutions with the current element. Therefore the **if**-statement must be modified to consider both cases, based on the number of items that will be used (use a further counting field).

```
begin
  $P[0,0].exist$ := true;
  $P[0,0].number := 0$;
  for $k \in [1..K]$ do
    $P[0,k].exist$ := false; {$P[i,0]$ will be computed and thus does not need to be initialized}
  od;
  for $i \in [1..n]$ do
    for $k \in [0..K]$ do
      $P[i,k].exist$ := false;{default}
      if $P[i-1,k].exist$ then
        $P[i,k].exist$ := true;{Can do with subset}
        $P[i,k].belong$ := false;
        $s := P[i-1,k].number$;
        $P[i,k].number := s$;
      fi;
      if $k - S[i] \geq 0$ then
        if $P[i-1,k-S[i]].exist$ and ($P[i,k].exist$ = false or $s > P[i-1,k-S[i]].number + 1$) then
          $P[i,k].exist$ := true;{extend solution}
          $P[i,k].belong$ := true;
          $P[i,k].number := P[i-1,k-S[i]].number + 1$;
        fi;
      fi;
    od;
  od;
end
```

This approach is guaranteed to find an optimal solution: The optimal solution either uses the n-th item, then the remaining items must form an optimal solution for a smaller problem (show this!) or it does not use the n-th item, then it must be an optimal solution using fewer items. The modified algorithm considers both cases and selects the better one.