CS3001, Algorithm Design and Analysis
Sample solution

16. It is obviously sufficient to consider the following situation: We have found a position where the m letters of B match and a new letter b_{m+1} is entered. We can now assume that B has length $m+1$ and we got already a partial match for the first m letters – so we can just continue in the string-match algorithm. For this however we also need a value $next(m+1)$ and we can compute this by a single iteration of the **for**-loop from the algorithm that computes $next$ for the value $i = m+1$.

17. Suppose the algorithm A is guaranteed to find the k-th largest element (lets call it x) using element comparisons. Assume furthermore that there is an element y among the n elements, for which we cannot decide whether $x < y$ or $x > y$ (we want to show that this cannot be and so aim for a contradiction). Then the algorithms result must be the same if we change y from being a number $> x$ to a number $< x$ or vice versa (depending on whether $y < x$ or $y > x$ in reality). But this change would change the k-th largest element, contradiction.

19. We first compute a maximum matching for the graph and count the number of edges, call this number l. If we remove all edges with weight $> x$ from the graph and check, whether a maximum matching with l edges exists, we can test whether a matching with bottleneck weight $\leq x$ exists. Each such test takes time $O\left(\sqrt{n}(n+m)\right)$ (replace $|V| = n$, $|E| = m$ in the runtime of the improved algorithm).
Now we use a binary search (as we know that the edge weights are sorted, we know the range of weights to search within) the largest value of x such that we still find a matching with l edges, this value of x must be the bottleneck weight. This binary search over m edges takes $O\left(\log m\right)$ steps. So the total runtime is

$$O\left(\sqrt{n}(n+m)\right) + O\left(\sqrt{n}(n+m)\right) O\left(\log m\right) = O\left(\sqrt{n}(n+m)\log m\right)$$

Remark 1: The runtime is bigger than $m \log m$, so we could actually sort the edges first.
Remark 2: The solution in Manber gives a better time estimate but assumes that it can find a maximum matching in $O\left(\sqrt{n}m\right)$ which I don't see how this is done.
Remark 3: There are better algorithms for this concrete problem.

18. a) To get the optimal sequence of editing steps we store not only the cost matrix C, but also a matrix S that stores in each place what the optimal edit operation to get to this place was. Once we have computed all entries we start at position (n,m) and trace the optimum steps backwards through s.
The only slight problem is the indexing of letters: the indices i,j refer to the positions of the letters in the *original* words. Once we start changing A to B however they may become shifted. (Say we change `TEXT` to `EXIT`. After deleting the first `T`, the index of E will be 1, but the algorithm still considers it as the E which was in position 2.)
A solution to this problem however is easy. We keep a further array P that contains at every point the indices of the original letters. (A letter will only be touched once by the optimal edit sequence.) Initially P is set to $[1..n+m]$ (to cope that we add all letters of B to an empty A); every insert step increases the indices after the insert position by 1, every delete decreases the following indices. We then do not print the position i but $P[i]$.
The following program gives an explicit algorithm in GAP (you can find the source under `http://www-gap.dcs.st-and.ac.uk/~ahulpke/cs3001/diff`). When printing the editing steps it also actually traces them through the word to allow us to check what happened:

```
Diff:=function(a,b)
local C,S,n,m,o,i,j,del,ins,repl,v,P,d;
  n:=Length(a);m:=Length(b);
  C:=List([1..n+1],i->List([1..m+1]),j->0);
  S:=List([1..n+1],i->List([1..m+1]),j->0);
  # the entries of S will be of the form
  # [i,j,op] where i,j are the coordinates we
  # come from and op the operation we did.
  # (We could easily reconstruct the
  # coordinates from the operation, but this
  # would require a few further case
  # distinctions.

  for i in [1..n+1] do
    # get rid of letters by delete
    C[i][1]:=i-1;
    if i>1 then
      S[i][1]:=[i-1,1,"d"];
    fi;
  od;
  for j in [1..m+1] do
    # add letters by insert
    C[1][j]:=j-1;
    if j>1 then
      S[1][j]:=[1,j-1,"i"];
    fi;
  od;
  for i in [2..n+1] do
    for j in [2..m+1] do
      del:=C[i-1][j]+1;
      ins:=C[i][j-1]+1;
      if a[i-1]=b[j-1] then
        repl:=C[i-1][j-1];
      else
        repl:=C[i-1][j-1]+1;
      fi;
      o:=Minimum(del,ins,repl);
      C[i][j]:=o;
      # store what gave the optimum: (*)
      if del=o then
        S[i][j]:=[i-1,j,"d"];
      elif ins=o then
        S[i][j]:=[i,j-1,"i"];
      elif a[i-1]<>b[j-1] then
        S[i][j]:=[i-1,j-1,"r"];
      else
        S[i][j]:=[i-1,j-1,"x"];
      fi;
    od;
  od;

  # collect the editing steps we needed (in
  # reverse order)
  i:=n+1;
  j:=m+1;
  v:=[];
  while i>1 or j>1 do
     o:=S[i][j];
     Add(v,o);
     i:=o[1];
     j:=o[2];
  od;
  v:=Reversed(v);

  # store shifted indices.  P[i] gives the
  # current position of the original $i$-th
  # letter
  P:=[1..n+m]; # we might add $m$ letters.

  # Now print out the editing steps and
  # perform them in parallel to show that
  # everything worked.

  d:=ShallowCopy(a); # save original
  for o in v do
    # o is of the form [old_i,old_j,action]
    if o[3]="i" then
      d:=Concatenation(d{[1..P[o[1]]-1]},
              b{[o[2]]},d{[P[o[1]]..Length(d)]});
      Print("i:",P[o[1]],b[o[2]]," ",d,"\n");
      for i in [o[1]..Length(P)] do
        P[i]:=P[i]+1;
      od;
    elif o[3]="d" then
      d:=Concatenation(d{[1..P[o[1]]-1]},
              d{[P[o[1]]+1..Length(d)]});
      Print("d:",P[o[1]],a[o[1]]," ",d,"\n");
      for i in [o[1]..Length(P)] do
        P[i]:=P[i]-1;
      od;
    elif o[3]="r" then
      d:=Concatenation(d{[1..P[o[1]]-1]},
             b{[o[2]]},d{[P[o[1]]+1..Length(d)]});
      Print("r:",P[o[1]],":",a[o[1]],"->",
                  b[o[2]]," ",d,"\n");
    fi;
  od;
end;
```

b) The above program produces (The solution is not unique. If we change the sequence of **if** statements at (*), we may get another optimal edit sequence):

```
gap> Diff("APPLE MACINTOSH","LAPTOP MACHINES");
i:1'L' LAPPLE MACINTOSH
r:4:'P'->'T' LAPTLE MACINTOSH
r:5:'L'->'O' LAPTOE MACINTOSH
r:6:'E'->'P' LAPTOP MACINTOSH
i:11'H' LAPTOP MACHINTOSH
r:14:'T'->'E' LAPTOP MACHINEOSH
d:15'O' LAPTOP MACHINESH
d:16'H' LAPTOP MACHINES
```