Answers to EDAF05 exam 20180529

Note: pseudo code was requested for some questions. For question 4b you can find acceptable minimal textual pseudo code. The point with pseudo code is to make a description as clear as possible. This is a thin line because too abstract pseudo code may hide the time complexity of an implementation. So what is acceptable depends on what the examiner thinks is sufficiently obvious (and not a way to hide limited knowledge of the student). To avoid possible problem, it can be better to be more detailed, but as far as I remember from the exam, nobody got minus for the level of abstraction used in 4b (of course, depending on the contents of the solutions). For other cases I have provided C code but that detail is not necessary.

1a true

1b true

1c false

2a true

3a $O(\log n)$

4a Planning

4b The key is to consider the union of all constraints. The constraints describe a graph and if there is a cycle, the output should be *impossible*. If there is no cycle, one possible itenary should be printed. Maintain a set with nodes which have no predecessors. Take any node from this set, remove it from the set and the graph, print it, and add any nodes to the set which now have no predecessor, by checking each successor of the removed node. Continue until all nodes have been printed or until there are nodes left which have not been printed but all of them have a predecessor (i.e. there is a cycle).

4c The above can easily be done in linear time.

5a Attack

5b C solution which prints the preferable interval.

```
#include <limits.h>

int f(int a[], int u, int w, int* from, int* to)
{
        int     v;
        int     x;
        int     y;
        int     z;
        int     i;
        int     s;
        int     l;
        int     r;
        int     m;
        int     left_from;
        int     left_to;
        int     right_from;
        int     right_to;
        int     max_left_index;
        int     max_right_index;

        if (u == w) {
                *from = *to = u;
                return a[u];
```

```
        }

        v = (u + w)/2;
        x = f(a, u, v, &left_from, &left_to);
        y = f(a, v+1, w, &right_from, &right_to);

        s = 0;
        l = INT_MIN;
        max_left_index = v;
        for (i = v; i >= u; i -= 1) {
                s += a[i];
                if (s > l) {
                        l = s;
                        max_left_index = i;
                }
        }

        s = 0;
        r = INT_MIN;
        max_right_index = v+1;
        for (i = v+1; i <= w; i += 1) {
                s += a[i];
                if (s > r) {
                        r = s;
                        max_right_index = i;
                }
        }

        if (x >= y) {
                m = x;
                *from = left_from;
                *to = left_to;
        } else {
                m = y;
                *from = right_from;
                *to = right_to;
        }

        if (m < l+r) {
                m = l+r;
                *from = max_left_index;
                *to = max_right_index;
        }

        return m;
}
```

5c $O(n \log n)$

6a Menus

6b The following is a simple solution translated directly from the idea of minimizing the sum of the cost for ”the current line” plus the cost of the remaining lines, using memoization. It is possible to make a linear solution. Words are indexed from 0 to $n - 1$.

```
#include <assert.h>
```

```c
#include <ctype.h>
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define M       (60)    // line width
#define N       (1000)  // max words
#define W       (40)    // max word size

char    word[N][W];     // input words
int     length[N];      // length of word[i]
int     table[N][N];    // optimal cost of placing words i,j with i first on a new line.
int     k_index[N][N];  // optimal choice of last word index for i,j.

int cost(int w, int k, int n)
{
        int     c;

        if (k == n-1)
                return 0;
        else {
                c = M-w;
                return c*c*c;
        }
}

int opt(int i, int j, int n)
{
        int     k;
        int     width;
        int     best;
        int     best_k;
        int     x;
        int     y;

        /* compute minimum cost of placing words i..j,
         * with words i..k on one line and words k+1..j
         * on additional lines.
         *
         */

        if (i > j)
                return 0;

        if (table[i][j] >= 0)
                return table[i][j];

        best = INT_MAX;

        width = -1; // no space to the left of first word.

        for (k = i; k <= j; k += 1) {
```

```
                width += 1 + length[k];

                if (width > M)
                        break;

                x = cost(width, k, n);
                y = opt(k+1, j, n);

                if (x + y < best) {
                        best = x + y;
                        best_k = k;
                }
        }

        table[i][j] = best;
        k_index[i][j] = best_k;

        return best;
}

int main()
{
        int     c;
        int     i;
        int     j;
        int     k;
        int     n;
        int     x;
        int     w;

        j = i = 0;
        memset(table, -1, sizeof table);
        while ((c = getchar()) != EOF) {
                if (isspace(c)) {
                        if (j > 0) {
                                length[i] = j;
                                j = 0;
                                i += 1;
                        }

                        continue;
                }
                word[i][j++] = c;
        }

        n = i;

        opt(0, n-1, n);

        for (i = 0; i < n; i += 1) {
                k = k_index[i][n-1];
                w = 0;
                for (j = i; j <= k; j += 1) {
                        printf("%s", word[j]);
```

```
                        w += length[j];
                        if (j < k) {
                                w += 1;
                                printf(" ");
                        }
                }
                for (j = w; j < M; j += 1)
                        printf(" ");
                printf("\n");
                i = k;
        }
}
```

6c $O(n^2)$

7a Labs

7b A graph is created with students and courses as nodes. There is an edge from a student to each course it is interested in, with capacity 1. A source node with an edge to each student with capacity 2, and a sink node with an edge from each course are also created. The capacity on an edge to the sink is equivalent to the needed number of lab assistants for the corresponding course. Computing the maximum flow gives the answer.

7c $O(Cm)$

7d Ford-Fulkerson, and C is the sum of the capacities on the edges from the source node and m the number of edges.

8a Distribution

8b Independent Set

8c $P_2 \leq_P P_1$

8d Given an instance to problem Independent Set, I_{IS}, we create an instance to problem Distribution, I_D, such that for each node u in I_{IS}, we create a city c_u in I_D, and for each edge (u, v) in I_{IS}, we set the distance between the cities corresponding to u and v, i.e. c_u and c_v, to less than 250 km, and for each pair of nodes u and v without an edge (u, v) we set the corresponding distance to more than 250 km. For a solution X_D to I_D of size S, there is a solution X_{IS} to I_{IS}, also of size S, of nodes corresponding to the cities in X_D, i.e. if $c_u \in X_D$ then $u \in X_{IS}$. Therefore, since Independent Set is NP-complete, Distribution is also NP-complete.