Листинг 1.1. Реализация алгоритма сортировки вставками

```
void insertion_sort(item_type s[], int n) {[
    int i, j;    /* counters */

    for (i = 1; i < n; i++) {
        j = i;
        while ((j > 0) && (s[j] < s[j - 1])) {
            swap(&s[j], &s[j - 1]);
            j = j-1;
        }
    }
}
```

Листинг 2.1. Реализация алгоритма сортировки методом выбора на языке C

```
void selection_sort(item_type s[], int n) {
    int i, j;   /* counters */
    int min;    /* index of minimum */

    for (i = 0; i < n; i++) {
        min = i;
        for (j = i + 1; j < n; j++) {
            if (s[j] < s[min]) {
                min = j;
            }
        }
        swap(&s[i], &s[min]);
    }
}
```

Листинг 2.2. Внутренние циклы алгоритма сортировки вставками на языке C

```
for (i = 1; i < n; i++) {
    j = i;
    while ((j > 0) && (s[j] < s[j - 1])) {
        swap(&s[j], &s[j - 1]);
        j = j-1;
    }
}
```

Листинг 2.3. Реализация алгоритма поиска строки в тексте

```
int findmatch(char *p, char *t) {
    int i, j;          /* counters */
    int plen, tlen;    /* string lengths */

    plen = strlen(p);
    tlen = strlen(t);

    for (i = 0; i <= (tlen-plen); i = i + 1) {
        j = 0;
        while ((j < plen) && (t[i + j] == p[j])) {
            j = j + 1;
        }
        if (j == plen) {
            return(i);  /* location of the first match */
        }
    }
    return(-1);         /* there is no match */
}
```

Листинг 2.4. Умножение матриц

```
for (i = 1; i <= a->rows; i++) {
    for (j = 1; j <= b->columns; j++) {
        c->m[i][j] = 0;
        for (k = 1; k <= b->rows; k++) {
            c->m[i][j] += a->m[i][k] * b->m[k][j];
        }
    }
}
```

Листинг 3.1. Объявление структуры связного списка

```
typedef struct list {
    item_type item;         /* data item */
    struct list *next;      /* point to successor */
} list;
```

Листинг 3.2. Рекурсивный поиск элемента в связном списке

```
list *search_list(list *l, item_type x) {
    if (l == NULL) {
        return(NULL);
    }

    if (l->item == x) {
        return(l);
    } else {
        return(search_list(l->next, x));
    }
}
```

Листинг 3.3. Вставка элемента в однонаправленный связный список

```
void insert_list(list **l, item_type x) {
    list *p;    /* temporary pointer */

    p = malloc(sizeof(list));
    p->item = x;
    p->next = *l;
    *l = p;
}
```

Листинг 3.4. Поиск указателя на элемент, предшествующий удаляемому

```
list *item_ahead(list *l, list *x) {
    if ((l == NULL) || (l->next == NULL)) {
        return(NULL);
    }

    if ((l->next) == x) {
        return(l);
    } else {
        return(item_ahead(l->next, x));
    }
}
```

Листинг 3.5. Удаление элемента связного списка

```
void delete_list(list **l, list **x) {
    list *p;            /* item pointer */
    list *pred;         /* predecessor pointer */

    p = *l;
    pred = item_ahead(*l, *x);

    if (pred == NULL) { /* splice out of list */
        *l = p->next;
    } else {
        pred->next = (*x)->next;
    }
    free(*x);           /* free memory used by node */
}
```

Листинг 3.6. Объявление типа для структуры дерева

```
typedef struct tree {
    item_type item;         /* data item */
    struct tree *parent;    /* pointer to parent */
    struct tree *left;      /* pointer to left child */
    struct tree *right;     /* pointer to right child */
} tree;
```

Листинг 3.7. Алгоритм рекурсивного поиска произвольного элемента в двоичном дереве

```
tree *search_tree(tree *l, item_type x) {
    if (l == NULL) {
        return(NULL);
    }

    if (l->item == x) {
        return(l);
    }

    if (x < l->item) {
        return(search_tree(l->left, x));
    } else {
        return(search_tree(l->right, x));
    }
}
```

Листинг 3.8. Поиск наименьшего элемента в двоичном дереве

```
tree *find_minimum(tree *t) {
    tree *min;    /* pointer to minimum */

    if (t == NULL) {
        return(NULL);
    }

    min = t;
    while (min->left != NULL) {
        min = min->left;
    }
    return(min);
}
```

Листинг 3.9. Рекурсивный алгоритм симметричного обхода двоичного дерева

```
void traverse_tree(tree *l) {
    if (l != NULL) {
        traverse_tree(l->left);
        process_item(l->item);
        traverse_tree(l->right);
    }
}
```

Листинг 3.10. Вставка узла в двоичное дерево поиска

```
void insert_tree(tree **l, item_type x, tree *parent) {
    tree *p;    /* temporary pointer */

    if (*l == NULL) {
        p = malloc(sizeof(tree));
        p->item = x;
        p->left = p->right = NULL;
        p->parent = parent;
        *l = p;
        return;
    }

    if (x < (*l)->item) {
        insert_tree(&((*l)->left), x, *l);
    } else {
        insert_tree(&((*l)->right), x, *l);
    }
}
```

Код 4.1. Функция сортировки `qsort`

```
#include <stdlib.h>

void qsort(void *base, size_t nel, size_t width,
          int (*compare) (const void *, const void *));
```

Листинг 4.1. Реализация функции сравнения

```
int intcompare(int *i, int *j)
{
    if (*i > *j) return (1);
    if (*i < *j) return (-1);
    return (0);
}
```

Код 4.2. Вызов функции сортировки

```
qsort(a, n, sizeof(int), intcompare);
```

Код 4.3. Структура данных пирамиды

```
typedef struct {
    item_type q[PQ_SIZE+1];      /* body of queue */
    int n;                       /* number of queue elements */
} priority_queue;
```

Листинг 4.2. Код для работы с пирамидой

```
int pq_parent(int n) {
    if (n == 1) {
        return(-1);
    }
    return((int) n/2);      /* implicitly take floor(n/2) */
}


int pq_young_child(int n) {
    return(2 * n);
}
```

Листинг 4.3. Вставка элемента в пирамиду

```
void pq_insert(priority_queue *q, item_type x) {
    if (q->n >= PQ_SIZE) {
        printf("Warning: priority queue overflow! \n");
    } else {
        q->n = (q->n) + 1;
        q->q[q->n] = x;
        bubble_up(q, q->n);
    }
}


void bubble_up(priority_queue *q, int p) {
    if (pq_parent(p) == -1) {
        return;     /* at root of heap, no parent */
    }

    if (q->q[pq_parent(p)] > q->q[p]) {
        pq_swap(q, p, pq_parent(p));
        bubble_up(q, pq_parent(p));
    }
}
```

Листинг 4.4. Создание пирамиды повторяющимися вставками

```
void pq_init(priority_queue *q) {
    q->n = 0;
}


void make_heap(priority_queue *q, item_type s[], int n) {
    int i;          /* counter */

    pq_init(q);
    for (i = 0; i < n; i++) {
        pq_insert(q, s[i]);
    }
}
```

Листинг 4.5. Удаление наименьшего элемента пирамиды

```
item_type extract_min(priority_queue *q) {
    int min = -1;    /* minimum value */

    if (q->n <= 0) {
        printf("Warning: empty priority queue.\n");
    } else {
        min = q->q[1];

        q->q[1] = q->q[q->n];
        q->n = q->n - 1;
        bubble_down(q, 1);
    }
    return(min);
}



void bubble_down(priority_queue *q, int p) {
    int c;          /* child index */
    int i;          /* counter */
    int min_index;  /* index of lightest child */

    c = pq_young_child(p);
    min_index = p;

    for (i = 0; i <= 1; i++) {

         if ((c + i) <= q->n) {
             if (q->q[min_index] > q->q[c + i]) {
                 min_index = c + i;
             }
         }
     }

     if (min_index != p) {
         pq_swap(q, p, min_index);
         bubble_down(q, min_index);
     }
}
```

Листинг 4.6. Алгоритм пирамидальной сортировки

```
void heapsort_(item_type s[], int n) {
    int i;              /* counters */
    priority_queue q;   /* heap for heapsort */

    make_heap(&q, s, n);

    for (i = 0; i < n; i++) {
        s[i] = extract_min(&q);
    }
}
```

Листинг 4.7. Алгоритм быстрого создания пирамиды

```
void make_heap_fast(priority_queue *q, item_type s[], int n) {
    int i;              /* counter */

    q->n = n;
    for (i = 0; i < n; i++) {
        q->q[i + 1] = s[i];
    }

    for (i = q->n/2; i >= 1; i--) {
         bubble_down(q, i);
    }
}
```

Листинг 4.8. Сравнение *k*-го элемента с числом *x*

```
int heap_compare(priority_queue *q, int i, int count, int x) {
    if ((count <= 0) || (i > q->n)) {
        return(count);
    }

    if (q->q[i] < x) {
        count = heap_compare(q, pq_young_child(i), count-1, x);
        count = heap_compare(q, pq_young_child(i)+1, count, x);
    }

    return(count);
}
```

Листинг 4.9. Сортировка вставками

```
for (i = 1; i < n; i++) {
     j = i;
     while ((j > 0) && (s[j] < s[j - 1])) {
          swap(&s[j], &s[j - 1]);
          j = j-1;
     }
}
```

Листинг 4.10. Код реализации алгоритма сортировки слиянием

```
void merge_sort(item_type s[], int low, int high) {
    int middle;    /* index of middle element */

    if (low < high) {
        middle = (low + high) / 2;
        merge_sort(s, low, middle);
        merge_sort(s, middle + 1, high);

        merge(s, low, middle, high);
    }
}
```

Листинг 4.11. Процедура слияния массивов

```
void merge(item_type s[], int low, int middle, int high) {
    int i;                  /* counter */
    queue buffer1, buffer2; /* buffers to hold elements for merging */

    init_queue(&buffer1);
    init_queue(&buffer2);

    for (i = low; i <= middle; i++) enqueue(&buffer1, s[i]);
    for (i = middle + 1; i <= high; i++) enqueue(&buffer2, s[i]);

    i = low;
    while (!(empty_queue(&buffer1) || empty_queue(&buffer2))) {
        if (headq(&buffer1) <= headq(&buffer2)) {
            s[i++] = dequeue(&buffer1);
        } else {
            s[i++] = dequeue(&buffer2);
        }
    }

    while (!empty_queue(&buffer1)) {
        s[i++] = dequeue(&buffer1);
    }

    while (!empty_queue(&buffer2)) {
        s[i++] = dequeue(&buffer2);
    }
}
```

Листинг 4.12. Код алгоритма быстрой сортировки

```
void quicksort(item_type s[], int l, int h) {
    int p;    /* index of partition */

    if (l < h) {
        p = partition(s, l, h);
        quicksort(s, l, p - 1);
        quicksort(s, p + 1, h);
    }
}
```

Листинг 4.13. Процедура разбиения массива на части

```
int partition(item_type s[], int l, int h) {
    int i;          /* counter */
    int p;          /* pivot element index */
    int firsthigh;  /* divider position for pivot element */

    p = h;          /* select last element as pivot */
    firsthigh = l;
    for (i = l; i < h; i++) {
        if (s[i] < s[p]) {
            swap(&s[i], &s[firsthigh]);
            firsthigh++;
        }
    }
    swap(&s[p], &s[firsthigh]);

    return(firsthigh);
}
```

Листинг 5.1. Реализация алгоритма двоичного поиска

```
int binary_search(item_type s[], item_type key, int low, int high) {
    int middle;    /* index of middle element */

    if (low > high) {
        return (-1);    /* key not found */
    }

    middle = (low + high) / 2;

    if (s[middle] == key) {
        return(middle);
    }

    if (s[middle] > key) {
        return(binary_search(s, key, low, middle - 1));
    } else {
        return(binary_search(s, key, middle + 1, high));
    }
}
```

Код 5.1. Границы циклов

```
for (i = 0; i < n+m-1; i++) {
    for (j = max(0,i-(n-1)); j <= min(m-1,i); j++) {
        c[i] = c[i] + a[j] * b[i-j];
    }
}
```

Листинг 7.1. Реализация графов посредством списков смежности

```
#define MAXV        100     /* maximum number of vertices */

typedef struct edgenode {
    int y;                  /* adjacency info */
    int weight;             /* edge weight, if any */
    struct edgenode *next;  /* next edge in list */
} edgenode;

typedef struct {
    edgenode *edges[MAXV+1];  /* adjacency info */
    int degree[MAXV+1];       /* outdegree of each vertex */
    int nvertices;            /* number of vertices in the graph */
    int nedges;               /* number of edges in the graph */
    int directed;             /* is the graph directed? */
} graph;
```

Листинг 7.2. Формат графа

```
void initialize_graph(graph *g, bool directed) {
    int i;    /* counter */

    g->nvertices = 0;
    g->nedges = 0;
    g->directed = directed;

    for (i = 1; i <= MAXV; i++) {
        g->degree[i] = 0;
    }

    for (i = 1; i <= MAXV; i++) {
        g->edges[i] = NULL;
    }
}
```

Листинг 7.3. Считывание графа

```
void read_graph(graph *g, bool directed) {
    int i;              /* counter */
    int m;              /* number of edges */
    int x, y;           /* vertices in edge (x,y) */

    initialize_graph(g, directed);

    scanf("%d %d", &(g->nvertices), &m);

    for (i = 1; i <= m; i++) {
        scanf("%d %d", &x, &y);
        insert_edge(g, x, y, directed);
    }
}
```

Листинг 7.4. Вставка ребра

```
void insert_edge(graph *g, int x, int y, bool directed) {
    edgenode *p;        /* temporary pointer */

    p = malloc(sizeof(edgenode));    /* allocate edgenode storage */

    p->weight = 0;
    p->y = y;
    p->next = g->edges[x];

    g->edges[x] = p;    /* insert at head of list */

    g->degree[x]++;

    if (!directed) {
        insert_edge(g, y, x, true);
    } else {
        g->nedges++;
    }
}
```

Листинг 7.5. Вывод графа на экран

```
void print_graph(graph *g) {
    int i;          /* counter */
    edgenode *p;    /* temporary pointer */

    for (i = 1; i <= g->nvertices; i++) {
        printf("%d: ", i);
        p = g->edges[i];
        while (p != NULL) {
            printf(" %d", p->y);
            p = p->next;
        }
        printf("\n");
    }
}
```

Код 7.1. Булевы переменные для хранения информации о каждой вершине графа

```
bool processed[MAXV+1];     /* which vertices have been processed */
bool discovered[MAXV+1];    /* which vertices have been found */
int parent[MAXV+1];         /* discovery relation */
```

Листинг 7.8. Инициализация вершин

```
void initialize_search(graph *g) {
    int i;                  /* counter */

    time = 0;

    for (i = 0; i <= g->nvertices; i++) {
        processed[i] = false;
        discovered[i] = false;
        parent[i] = -1;
    }
}
```

Листинг 7.9. Обход графа в ширину

```
void bfs(graph *g, int start) {
    queue q;            /* queue of vertices to visit */
    int v;              /* current vertex */
    int y;              /* successor vertex */
    edgenode *p;        /* temporary pointer */

    init_queue(&q);
    enqueue(&q, start);
    discovered[start] = true;

    while (!empty_queue(&q)) {
        v = dequeue(&q);
        process_vertex_early(v);
        processed[v] = true;
        p = g->edges[v];
        while (p != NULL) {
            y = p->y;
            if ((!processed[y]) || g->directed) {
                process_edge(v, y);
            }
            if (!discovered[y]) {
                enqueue(&q,y);
                discovered[y] = true;
                parent[y] = v;
            }
            p = p->next;
        }
        process_vertex_late(v);
    }
}
```

Листинг 7.10. Функция `process_vertex_late()`

```
void process_vertex_late(int v) {

}
```

Листинг 7.11. Функции process_vertex_early() и process_edge()

```
void process_vertex_early(int v) {
    printf("processed vertex %d\n", v);
}


void process_edge(int x, int y) {
    printf("processed edge (%d,%d)\n", x, y);
}
```

Листинг 7.12. Функция `process_edge()` для подсчета количества ребер

```
void process_edge(int x, int y) {
    nedges = nedges + 1;
}
```

Листинг 7.13. Изменение направления пути посредством рекурсии

```
void find_path(int start, int end, int parents[]) {
    if ((start == end) || (end == -1)) {
        printf("\n%d", start);
    } else {
        find_path(start, parents[end], parents);
        printf(" %d", end);
    }
}
```

Листинг 7.14. Процедура поиска компонентов связности

```
void connected_components(graph *g) {
    int c;              /* component number */
    int i;              /* counter */

    initialize_search(g);

    c = 0;
    for (i = 1; i <= g->nvertices; i++) {
        if (!discovered[i]) {
            c = c + 1;
            printf("Component %d:", c);
            bfs(g, i);
            printf("\n");
        }
    }
}

void process_vertex_early(int v) {        /* vertex to process */
    printf(" %d", v);
}



void process_edge(int x, int y) {

}
```

Листинг 7.15. Процедура раскраски графов двумя цветами

```
void twocolor(graph *g) {
    int i;    /* counter */

    for (i = 1; i <= (g->nvertices); i++) {
        color[i] = UNCOLORED;
    }

    bipartite = true;

    initialize_search(g);

    for (i = 1; i <= (g->nvertices); i++) {
        if (!discovered[i]) {
            color[i] = WHITE;
            bfs(g, i);
        }
    }
}


void process_edge(int x, int y) {
    if (color[x] == color[y]) {
        bipartite = false;
        printf("Warning: not bipartite, due to (%d,%d)\n", x, y);
    }

    color[y] = complement(color[x]);
}


int complement(int color) {
    if (color == WHITE) {
        return(BLACK);
    }
```

Листинг 7.17. Обход графа в глубину

```
void dfs(graph *g, int v) {
    edgenode *p;        /* temporary pointer */
    int y;              /* successor vertex */

    if (finished) {
        return;         /* allow for search termination */
    }
    discovered[v] = true;
    time = time + 1;
    entry_time[v] = time;

    process_vertex_early(v);

    p = g->edges[v];
    while (p != NULL) {
        y = p->y;
        if (!discovered[y]) {
            parent[y] = v;
            process_edge(v, y);
            dfs(g, y);
        } else if ((!processed[y]) && (parent[v] != y)) || (g->directed)) {
            process_edge(v, y);
        }

        if (finished) {
            return;
        }
        p = p->next;
    }

    process_vertex_late(v);
    time = time + 1;
    exit_time[v] = time;
    processed[v] = true;
}
```

Листинг 7.18. Поиск цикла

```
void process_edge(int x, int y) {
    if (parent[y] != x) {    /* found back edge! */
        printf("Cycle from %d to %d:", y, x);
        find_path(y, x, parent);
        finished = true;
    }
}
```

Листинг 7.19. Инициализация массива достижимых предшественников

```
int reachable_ancestor[MAXV+1]; /* earliest reachable ancestor of v */
int tree_out_degree[MAXV+1];    /* DFS tree outdegree of v */

void process_vertex_early(int v) {
    reachable_ancestor[v] = v;
}
```

Листинг 7.20. Определение возраста предшественников

```
void process_edge(int x, int y) {
    int class;    /* edge class */

    class = edge_classification(x, y);

    if (class == TREE) {
        tree_out_degree[x] = tree_out_degree[x] + 1;
    }

    if ((class == BACK) && (parent[x] != y)) {
        if (entry_time[y] < entry_time[reachable_ancestor[x]]) {
            reachable_ancestor[x] = y;
        }
    }
}
```

Листинг 7.21. Определение типа шарнира

```
void process_vertex_late(int v) {
    bool root;              /* is parent[v] the root of the DFS tree? */
    int time_v;             /* earliest reachable time for v */
    int time_parent;        /* earliest reachable time for parent[v] */

    if (parent[v] == -1) {  /* test if v is the root */
        if (tree_out_degree[v] > 1) {
            printf("root articulation vertex: %d \n",v);
        }
        return;
    }

    root = (parent[parent[v]] == -1);       /* is parent[v] the root? */

    if (!root) {
        if (reachable_ancestor[v] == parent[v]) {
            printf("parent articulation vertex: %d \n", parent[v]);
        }

        if (reachable_ancestor[v] == v) {
            printf("bridge articulation vertex: %d \n",parent[v]);

            if (tree_out_degree[v] > 0) {    /* is v is not a leaf? */
                printf("bridge articulation vertex: %d \n", v);
            }
        }
    }

    time_v = entry_time[reachable_ancestor[v]];
    time_parent = entry_time[reachable_ancestor[parent[v]]];

    if (time_v < time_parent) {
        reachable_ancestor[parent[v]] = reachable_ancestor[v];
    }
}
```

Листинг 7.22. Определение типа ребра

```
int edge_classification(int x, int y) {
    if (parent[y] == x) {
        return(TREE);
    }

    if (discovered[y] && !processed[y]) {
        return(BACK);
    }

    if (processed[y] && (entry_time[y]>entry_time[x])) {
        return(FORWARD);
    }

    if (processed[y] && (entry_time[y]<entry_time[x])) {
        return(CROSS);
    }

    printf("Warning: self loop (%d,%d)\n", x, y);

    return -1;
}
```

Листинг 7.23. Топологическая сортировка

```
void process_vertex_late(int v) {
    push(&sorted, v);
}


void process_edge(int x, int y) {
    int class;    /* edge class */

    class = edge_classification(x, y);

    if (class == BACK) {
        printf("Warning: directed cycle found, not a DAG\n");
    }
}


void topsort(graph *g) {
    int i;    /* counter */

    init_stack(&sorted);

    for (i = 1; i <= g->nvertices; i++) {
        if (!discovered[i]) {
            dfs(g, i);
        }
    }
    print_stack(&sorted);    /* report topological order */
}
```

Листинг 7.24. Транспонирование графа

```
graph *transpose(graph *g) {
    graph *gt;    /* transpose of graph g */
    int x;        /* counter */
    edgenode *p;  /* temporary pointer */

    gt = (graph *) malloc(sizeof(graph));
    initialize_graph(gt, true);       /* initialize directed graph */
    gt->nvertices = g->nvertices;

    for (x = 1; x <= g->nvertices; x++) {
        p = g->edges[x];
        while (p != NULL) {
            insert_edge(gt, p->y, x, true);
            p = p->next;
        }
    }

    return(gt);
}
```

Листинг 7.25. Алгоритм разложения графа на сильно связные компоненты

```
void strong_components(graph *g) {
    graph *gt;           /* transpose of graph g */
    int i;               /* counter */
    int v;               /* vertex in component */
    init_stack(&dfs1order);
    initialize_search(g);
    for (i = 1; i <= (g->nvertices); i++) {
        if (!discovered[i]) {
            dfs(g, i);
        }
    }

    gt = transpose(g);
    initialize_search(gt);

    components_found = 0;
    while (!empty_stack(&dfs1order)) {
        v = pop(&dfs1order);
        if (!discovered[v]) {
            components_found ++;
            printf("Component %d:", components_found);
            dfs2(gt, v);
            printf("\n");
        }
    }
}
```

Код 7.2. Учет ресурсов в бесконтурных ориентированных графах

```
void process_vertex_late(int v) {
    push(&dfs1order,v);
}
```

Код 7.3. Определение сильно связного компонента в транспонированном графе

```
void process_vertex_early2(int v) {
    printf(" %d", v);
}
```

Листинг 8.1. Определение структуры списка смежности

```
typedef struct {
    edgenode *edges[MAXV+1];  /* adjacency info */
    int degree[MAXV+1];       /* outdegree of each vertex */
    int nvertices;            /* number of vertices in the graph */
    int nedges;               /* number of edges in the graph */
    int directed;             /* is the graph directed? */
} graph;
```

Листинг 8.2. Структура переменной edgenode

```
typedef struct edgenode {
    int y;                  /* adjacency info */
    int weight;             /* edge weight, if any */
    struct edgenode *next;  /* next edge in list */
} edgenode;
```

Листинг 8.4. Реализация алгоритма Прима

```
int prim(graph *g, int start) {
    int i;                  /* counter */
    edgenode *p;            /* temporary pointer */
    bool intree[MAXV+1];    /* is the vertex in the tree yet? */
    int distance[MAXV+1];   /* cost of adding to tree */
    int v;                  /* current vertex to process */
    int w;                  /* candidate next vertex */
    int dist;               /* cheapest cost to enlarge tree */
    int weight = 0;         /* tree weight */

    for (i = 1; i <= g->nvertices; i++) {
        intree[i] = false;
        distance[i] = MAXINT;
        parent[i] = -1;
    }

    distance[start] = 0;
    v = start;

    while (!intree[v]) {
        intree[v] = true;
        if (v != start) {
            printf("edge (%d,%d) in tree \n",parent[v],v);
            weight = weight + dist;
        }
        p = g->edges[v];
        while (p != NULL) {
            w = p->y;
            if ((distance[w] > p->weight) && (!intree[w])) {
                distance[w] = p->weight;
                parent[w] = v;
            }
            p = p->next;
        }

        dist = MAXINT;
        for (i = 1; i <= g->nvertices; i++) {
            if ((!intree[i]) && (dist > distance[i])) {
                dist = distance[i];
                v = i;
            }
        }
    }

    return(weight);
}
```

Листинг 8.6. Реализация алгоритма Крускала

```
int kruskal(graph *g) {
    int i;                  /* counter */
    union_find s;           /* union-find data structure */
    edge_pair e[MAXV+1];    /* array of edges data structure */
    int weight=0;           /* cost of the minimum spanning tree */

    union_find_init(&s, g->nvertices);

    to_edge_array(g, e);
    qsort(&e,g->nedges, sizeof(edge_pair), &weight_compare);

    for (i = 0; i < (g->nedges); i++) {
        if (!same_component(&s, e[i].x, e[i].y)) {
            printf("edge (%d,%d) in MST\n", e[i].x, e[i].y);
            weight = weight + e[i].weight;
            union_sets(&s, e[i].x, e[i].y);
        }
    }

    return(weight);
}
```

Листинг 8.7. Определение структуры данных `union_find`

```
typedef struct {
    int p[SET_SIZE+1];      /* parent element */
    int size[SET_SIZE+1];   /* number of elements in subtree i */
    int n;                  /* number of elements in set */
} union_find;
```

Листинг 8.8. Реализация операций `union` и `find`

```
void union_find_init(union_find *s, int n) {
    int i;    /* counter */

    for (i = 1; i <= n; i++) {
        s->p[i] = i;
        s->size[i] = 1;
    }
    s->n = n;
}


int find(union_find *s, int x) {
    if (s->p[x] == x) {
        return(x);
    }
    return(find(s, s->p[x]));
}


void union_sets(union_find *s, int s1, int s2) {
    int r1, r2;    /* roots of sets */

    r1 = find(s, s1);
    r2 = find(s, s2);

    if (r1 == r2) {
        return;    /* already in same set */
    }

    if (s->size[r1] >= s->size[r2]) {
        s->size[r1] = s->size[r1] + s->size[r2];
        s->p[r2] = r1;
    } else {
        s->size[r2] = s->size[r1] + s->size[r2];
        s->p[r1] = r2;
    }
}
bool same_component(union_find *s, int s1, int s2) {
    return (find(s, s1) == find(s, s2));
}
```

Листинг 8.10. Реализация алгоритма Дейкстры

```
int dijkstra(graph *g, int start) {
    int i;                  /* counter */
    edgenode *p;            /* temporary pointer */
    bool intree[MAXV+1];    /* is the vertex in the tree yet? */
    int distance[MAXV+1];   /* cost of adding to tree */
    int v;                  /* current vertex to process */
    int w;                  /* candidate next vertex */
    int dist;               /* cheapest cost to enlarge tree */
    int weight = 0;         /* tree weight */

    for (i = 1; i <= g->nvertices; i++) {
        intree[i] = false;
        distance[i] = MAXINT;
        parent[i] = -1;
    }

    distance[start] = 0;
    v = start;
    while (!intree[v]) {
        intree[v] = true;
        if (v != start) {
            printf("edge (%d,%d) in tree \n",parent[v],v);
            weight = weight + dist;
        }
        p = g->edges[v];
        while (p != NULL) {
            w = p->y;
            if (distance[w] > (distance[v]+p->weight)) { /* CHANGED */
                distance[w] = distance[v]+p->weight;     /* CHANGED */
                parent[w] = v;                           /* CHANGED */
            }
            p = p->next;
        }

        dist = MAXINT;
        for (i = 1; i <= g->nvertices; i++) {
            if ((!intree[i]) && (dist > distance[i])) {
                dist = distance[i];
                v = i;
            }
        }
    }

    return(weight);
}
```

Листинг 8.11. Определение типа матрицы смежности

```
typedef struct {
    int weight[MAXV+1][MAXV+1];  /* adjacency/weight info */
    int nvertices;               /* number of vertices in graph */
} adjacency_matrix;
```

Листинг 8.12. Реализация алгоритма Флойда — Уоршелла

```
void floyd(adjacency_matrix *g) {
    int i, j;           /* dimension counters */
    int k;              /* intermediate vertex counter */
    int through_k;      /* distance through vertex k */

    for (k = 1; k <= g->nvertices; k++) {
        for (i = 1; i <= g->nvertices; i++) {
            for (j = 1; j <= g->nvertices; j++) {
                through_k = g->weight[i][k]+g->weight[k][j];
                if (through_k < g->weight[i][j]) {
                    g->weight[i][j] = through_k;
                }
            }
        }
    }
}
```

Листинг 8.13. Модифицированная структура ребра

```
typedef struct {
    int v;                  /* neighboring vertex */
    int capacity;           /* capacity of edge */
    int flow;               /* flow through edge */
    int residual;           /* residual capacity of edge */
    struct edgenode *next;  /* next edge in list */
} edgenode;
```

Листинг 8.14. Процедура поиска оптимального потока

```
void netflow(flow_graph *g, int source, int sink) {
    int volume;    /* weight of the augmenting path */

    add_residual_edges(g);

    initialize_search(g);
    bfs(g, source);

    volume = path_volume(g, source, sink);

    while (volume > 0) {
        augment_path(g, source, sink, volume);
        initialize_search(g);
        bfs(g, source);
        volume = path_volume(g, source, sink);
    }
}
```

Листинг 8.15. Процедура для различения насыщенных и ненасыщенных ребер

```
bool valid_edge(edgenode *e) {
    return (e->residual > 0);
}
```

Листинг 8.16. Добавление увеличивающих путей в поток

```
int path_volume(flow_graph *g, int start, int end) {
    edgenode *e;    /* edge in question */

    if (parent[end] == -1) {
        return(0);
    }

    e = find_edge(g, parent[end], end);

    if (start == parent[end]) {
        return(e->residual);
    } else {
        return(min(path_volume(g, start, parent[end]), e->residual));
    }
}
```

Листинг 8.17. Модификация ребер

```
void augment_path(flow_graph *g, int start, int end, int volume) {
    edgenode *e;    /* edge in question */

    if (start == end) {
        return;
    }

    e = find_edge(g, parent[end], end);
    e->flow += volume;
    e->residual -= volume;

    e = find_edge(g, end, parent[end]);
    e->residual += volume;

    augment_path(g, start, parent[end], volume);
}
```

Листинг 9.2. Реализация алгоритма перебора с возвратом

```
void backtrack(int a[], int k, data input) {
    int c[MAXCANDIDATES];       /* candidates for next position */
    int nc;                     /* next position candidate count */
    int i;                      /* counter */

    if (is_a_solution(a, k, input)) {
        process_solution(a, k,input);
    } else {
        k = k + 1;
        construct_candidates(a, k, input, c, &nc);
        for (i = 0; i < nc; i++) {
            a[k] = c[i];
            make_move(a, k, input);
            backtrack(a, k, input);
            unmake_move(a, k, input);

            if (finished) {
                return;         /* terminate early */
            }
        }
    }
}
```

Листинг 9.3. Реализация базовых процедур процедуры `backtrack()`

```
int is_a_solution(int a[], int k, int n) {
    return (k == n);
}

void construct_candidates(int a[], int k, int n, int c[], int *nc) {
    c[0] = true;
    c[1] = false;
    *nc = 2;
}
void process_solution(int a[], int k, int input) {
    int i;    /* counter */

    printf("{");
    for (i = 1; i <= k; i++) {
        if (a[i] == true) {
            printf(" %d", i);
        }
    }

    printf(" }\n");
}
```

Листинг 9.4. Вызов процедуры `backtrack()` для генерирования подмножеств

```
void generate_subsets(int n) {
    int a[NMAX];                    /* solution vector */

    backtrack(a, 0, n);
}
```

Листинг 9.5. Процедура `construct_candidates()` для генерирования всех перестановок

```
void construct_candidates(int a[], int k, int n, int c[], int *nc) {
    int i;                  /* counter */
    bool in_perm[NMAX];     /* what is now in the permutation? */

    for (i = 1; i < NMAX; i++) {
        in_perm[i] = false;
    }

    for (i = 1; i < k; i++) {
        in_perm[a[i]] = true;
    }

    *nc = 0;
    for (i = 1; i <= n; i++) {
        if (!in_perm[i]) {
            c[ *nc ] = i;
            *nc = *nc + 1;
        }
    }
}
```

Листинг 9.6. Процедуры генерирования перестановок

```
void process_solution(int a[], int k, int input) {
    int i;    /* counter */

    for (i = 1; i <= k; i++) {
        printf(" %d", a[i]);
    }
    printf("\n");
}


int is_a_solution(int a[], int k, int n) {
    return (k == n);
}


void generate_permutations(int n) {
    int a[NMAX];                    /* solution vector */

    backtrack(a, 0, n);
}
```

Листинг 9.7. Создание структуры для хранения входных данных для процедуры `backtrack`

```
typedef struct {
    int s;                      /* source vertex */
    int t;                      /* destination vertex */
    graph g;                    /* graph to find paths in */
} paths_data;
```

Листинг 9.8. Процедура `construct_candidates()` для перечисления всех путей в графе

```
void construct_candidates(int a[], int k, paths_data *g, int c[],
    int *nc) {
    int i;                /* counters */
    bool in_sol[NMAX+1];  /* what's already in the solution? */
    edgenode *p;          /* temporary pointer */
    int last;             /* last vertex on current path */

    for (i = 1; i <= g->g.nvertices; i++) {
        in_sol[i] = false;
    }

    for (i = 0; i < k; i++) {
        in_sol[a[i]] = true;
    }

    if (k == 1) {
        c[0] = g->s;      /* always start from vertex s */
        *nc = 1;
    } else {
        *nc = 0;
        last = a[k-1];
        p = g->g.edges[last];
        while (p != NULL) {
            if (!in_sol[ p->y ]) {
                c[*nc] = p->y;
                *nc= *nc + 1;
            }
            p = p->next;
        }
    }
}
```

Листинг 9.9. Процедуры для определения решения и его обработки

```
int is_a_solution(int a[], int k, paths_data *g) {
    return (a[k] == g->t);
}
```

Листинг 9.10. Процедура для подсчета количества обнаруженных путей

```
void process_solution(int a[], int k, paths_data *input) {
    int i;    /* counter */

    solution_count ++;

    printf("{");
    for (i = 1; i <= k; i++) {
        printf(" %d",a[i]);
    }
    printf(" }\n");
}
```

Листинг 9.11. Определение основных структур данных

```
#define DIMENSION   9                       /* 9*9 board */
#define NCELLS      DIMENSION*DIMENSION     /* 81 cells in 9-by-9-board */
#define MAXCANDIDATES   DIMENSION+1         /* max digit choices per cell */

bool finished = false;

typedef struct {
    int x, y;     /* row and column coordinates of square */
} point;

typedef struct {
   int m[DIMENSION+1][DIMENSION+1];    /* board contents */
   int freecount;                      /* open square count */
   point move[NCELLS+1];               /* which cells have we filled? */
} boardtype;
```

Листинг 9.12. Генерирование кандидатов на заполнение клеточки

```
void construct_candidates(int a[], int k, boardtype *board, int c[],
    int *nc) {
    int i;                      /* counter */
    bool possible[DIMENSION+1]; /* which digits fit in this square */

    next_square(&(board->move[k]), board); /* pick square to fill next */

    *nc = 0;

    if ((board->move[k].x < 0) && (board->move[k].y < 0)) {
        return;    /* error condition, no moves possible */
    }

    possible_values(board->move[k], board, possible);
    for (i = 1; i <= DIMENSION; i++) {
        if (possible[i]) {
            c[*nc] = i;
            *nc = *nc + 1;
        }
    }
}
```

Листинг 9.13. Процедуры make_move и unmake_move

```
void make_move(int a[], int k, boardtype *board) {
    fill_square(board->move[k], a[k], board);
}

void unmake_move(int a[], int k, boardtype *board) {
    free_square(board->move[k], board);
}
```

Листинг 9.14. Процедура отслеживания пустых клеточек

```
bool is_a_solution(int a[], int k, boardtype *board) {
    steps = steps + 1;          /* count steps for results table */

    return (board->freecount == 0);
}
```

Листинг 9.15. Завершение поиска и обработка решения

```
void process_solution(int a[], int k, boardtype *board) {
    finished = true;
    printf("process solution\n");
    print_board(board);
}
```

Листинг 9.16. Поиск кандидатов кратчайшего пути методом «лучший-первый»

```
void branch_and_bound (tsp_solution *s, tsp_instance *t) {
    int c[MAXCANDIDATES];       /* candidates for next position */
    int nc;                     /* next position candidate count */
    int i;                      /* counter */

    first_solution(&best_solution,t);
    best_cost = solution_cost(&best_solution, t);
    initialize_solution(s,t);
    extend_solution(s,t,1);
    pq_init(&q);
    pq_insert(&q,s);

    while (top_pq(&q).cost < best_cost) {
        *s = extract_min(&q);
        if (is_a_solution(s, s->n, t)) {
            process_solution(s, s->n, t);
        }
        else {
            construct_candidates(s, (s->n)+1, t, c, &nc);
            for (i=0; i<nc; i++) {
                extend_solution(s,t,c[i]);
                pq_insert(&q,s);
                contract_solution(s,t);
            }
        }
    }
}
```

Листинг 9.17. Процедуры extend_solution и contract_solution

```
void extend_solution(tsp_solution *s, tsp_instance *t, int v) {
    s->n++;
    s->p[s->n] = v;
    s->cost = partial_solution_lb(s,t);
}

void contract_solution(tsp_solution *s, tsp_instance *t) {
    s->n--;
    s->cost = partial_solution_lb(s,t);
}
```

Листинг 9.18. Вычисление стоимости решения

```
double partial_solution_cost(tsp_solution *s, tsp_instance *t) {
    int i;                  /* counter */
    double cost = 0.0;      /* cost of solution */

    for (i = 1; i < (s->n); i++) {
        cost = cost + distance(s, i, i + 1, t);
    }

    return(cost);
}

double partial_solution_lb(tsp_solution *s, tsp_instance *t) {
     return(partial_solution_cost(s,t) + (t->n - s->n + 1) * minlb);
}
```

Листинг 10.1. Рекурсивная функция для вычисления *n*-го числа Фибоначчи

```
long fib_r(int n) {
    if (n == 0) {
        return(0);
    }

    if (n == 1) {
        return(1);
    }

    return(fib_r(n-1) + fib_r(n-2));
}
```

Листинг 10.2. Вычисление чисел Фибоначчи с использованием кэширования

```
#define MAXN    92      /* largest n for which F(n) fits in a long */
#define UNKNOWN -1      /* contents denote an empty cell */
long f[MAXN+1];         /* array for caching fib values */

long fib_c(int n) {
    if (f[n] == UNKNOWN) {
        f[n] = fib_c(n-1) + fib_c(n-2);
    }

    return(f[n]);
}


long fib_c_driver(int n) {
    int i;      /* counter */

    f[0] = 0;
    f[1] = 1;

    for (i = 2; i <= n; i++) {
        f[i] = UNKNOWN;
    }

    return(fib_c(n));
}
```

Листинг 10.3. Вычисления числа Фибоначчи без рекурсии

```
long fib_dp(int n) {
    int i;                      /* counter */
    long f[MAXN+1];             /* array for caching values */

    f[0] = 0;
    f[1] = 1;

    for (i = 2; i <= n; i++) {
        f[i] = f[i-1] + f[i-2];
    }

    return(f[n]);
}
```

Листинг 10.4. Окончательная версия процедуры вычисления чисел Фибоначчи

```
long fib_ultimate(int n)
{
        int i;                  /* counter */
        long back2=0, back1=1;  /* last two values of f[n] */
        long next;              /* placeholder for sum */

        if (n == 0) return (0);

        for (i=2; i<n; i++) {
                next = back1+back2;
                back2 = back1;
                back1 = next;
        }
        return(back1+back2);
}
```

Листинг 10.5. Вычисление биномиального коэффициента

```
long binomial_coefficient(int n, int k) {
    int i, j;                   /* counters */
    long bc[MAXN+1][MAXN+1];    /* binomial coefficient table */

    for (i = 0; i <= n; i++) {
        bc[i][0] = 1;
    }

    for (j = 0; j <= n; j++) {
        bc[j][j] = 1;
    }

    for (i = 2; i <= n; i++) {
        for (j = 1; j < i; j++) {
            bc[i][j] = bc[i-1][j-1] + bc[i-1][j];
        }
    }

    return(bc[n][k]);
}
```

Листинг 10.6. Вычисление стоимости редактирования методом рекурсии

```
#define MATCH      0       /* enumerated type symbol for match */
#define INSERT     1       /* enumerated type symbol for insert */
#define DELETE     2       /* enumerated type symbol for delete */



int string_compare_r(char *s, char *t, int i, int j) {
    int k;            /* counter */
    int opt[3];       /* cost of the three options */
    int lowest_cost; /* lowest cost */

    if (i == 0) {    /* indel is the cost of an insertion or deletion */
        return(j * indel(' '));
    }

    if (j == 0) {
        return(i * indel(' '));
    }
                      /* match is the cost of a match/substitution */

    opt[MATCH]  = string_compare_r(s,t,i-1,j-1) + match(s[i],t[j]);
    opt[INSERT] = string_compare_r(s,t,i,j-1) + indel(t[j]);
    opt[DELETE] = string_compare_r(s,t,i-1,j) + indel(s[i]);

    lowest_cost = opt[MATCH];
    for (k = INSERT; k <= DELETE; k++) {
        if (opt[k] < lowest_cost) {
            lowest_cost = opt[k];
        }
    }

    return(lowest_cost);
}
```

Листинг 10.7. Структура таблицы для вычисления стоимости редактирования

```
typedef struct {
    int cost;                /* cost of reaching this cell */
    int parent;              /* parent cell */
} cell;


cell m[MAXLEN+1][MAXLEN+1];     /* dynamic programming table */
```

Листинг 10.8. Вычисление стоимости редактирования

```
int string_compare(char *s, char *t, cell m[MAXLEN+1][MAXLEN+1]) {
    int i, j, k;      /* counters */
    int opt[3];       /* cost of the three options */

    for (i = 0; i <= MAXLEN; i++) {
        row_init(i, m);
        column_init(i, m);
    }

    for (i = 1; i < strlen(s); i++) {
        for (j = 1; j < strlen(t); j++) {
            opt[MATCH] = m[i-1][j-1].cost + match(s[i], t[j]);
            opt[INSERT] = m[i][j-1].cost + indel(t[j]);
            opt[DELETE] = m[i-1][j].cost + indel(s[i]);

            m[i][j].cost = opt[MATCH];
            m[i][j].parent = MATCH;
            for (k = INSERT; k <= DELETE; k++) {
                if (opt[k] < m[i][j].cost) {
                    m[i][j].cost = opt[k];
                    m[i][j].parent = k;
                }
            }
        }
    }

    goal_cell(s, t, &i, &j);
    return(m[i][j].cost);
}
```

Листинг 10.9. Восстановление решения в прямом порядке

```
void reconstruct_path(char *s, char *t, int i, int j,
                      cell m[MAXLEN+1][MAXLEN+1]) {
    if (m[i][j].parent == -1) {
        return;
    }

    if (m[i][j].parent == MATCH) {
        reconstruct_path(s, t, i-1, j-1, m);
        match_out(s, t, i, j);
        return;
    }

    if (m[i][j].parent == INSERT) {
        reconstruct_path(s, t, i, j-1, m);
        insert_out(t, j);
        return;
    }

    if (m[i][j].parent == DELETE) {
        reconstruct_path(s, t, i-1, j, m);
        delete_out(s, i);
        return;
    }
}
```

Листинг 10.10. Процедуры инициализации строк и столбцов таблицы

```
row_init(int i)
{
  m[0][i].cost = i;
  if (i>0)
     m[0][i].parent =  INSERT;
  else
     m[0][i].parent = -1;
}
```

```
column_init(int i)
{
  m[i][0].cost = i;
  if (i>0)
     m[i][0].parent = DELETE;
  else
     m[i][0].parent = -1;
}
```

Листинг 10.11. Функции стоимости

```
int match(char c, char d)
{
  if (c == d) return(0);
  else return(1);
}
```

```
int indel(char c)
{
  return(1);
}
```

Листинг 10.12. Функция определения местонахождения целевой ячейки

```
void goal_cell(char *s, char *t, int *i, int *j) {
    *i = strlen(s) - 1;
    *j = strlen(t) - 1;
}
```

Листинг 10.13. Функции трассировки решения

```
insert_out(char *t, int j)
{
   printf("I");
}

delete_out(char *s, int i)
{
   printf("D");
}
```

```
match_out(char *s, char *t,
          int i, int j)
{
  if (s[i]==t[j]) printf("M");
  else printf("S");
}
```

Листинг 10.14. Модифицированные функции для поиска неточно совпадающих строк

```
void row_init(int i, cell m[MAXLEN+1][MAXLEN+1]) {
    m[0][i].cost = 0;        /* NOTE CHANGE */
    m[0][i].parent = -1;     /* NOTE CHANGE */
}
void goal_cell(char *s, char *t, int *i, int *j) {
    int k;    /* counter */

    *i = strlen(s) - 1;
    *j = 0;

    for (k = 1; k < strlen(t); k++) {
        if (m[*i][k].cost < m[*i][*j].cost) {
            *j = k;
        }
    }
}
```

Листинг 10.15. Модифицированная функция стоимости совпадений

```
int match(char c, char d) {
    if (c == d) {
        return(0);
    }
    return(MAXLEN);
}
```

Листинг 10.16. Алгоритм для определения возможности получения суммы *k*

```
bool sum[MAXN+1][MAXSUM+1];     /* table of realizable sums */
int parent[MAXN+1][MAXSUM+1];   /* table of parent pointers */

bool subset_sum(int s[], int n, int k) {
    int i, j;                    /* counters */

    sum[0][0] = true;
    parent[0][0] = NIL;

    for (i = 1; i <= k; i++) {
        sum[0][i] = false;
        parent[0][i] = NIL;
    }

    for (i = 1; i <= n; i++) {    /* build table */
        for (j = 0; j <= k; j++) {
            sum[i][j] = sum[i-1][j];
            parent[i][j] = NIL;

            if ((j >= s[i-1]) && (sum[i-1][j-s[i-1]]==true)) {
                sum[i][j] = true;
                parent[i][j] = j-s[i-1];
            }
        }
    }

    return(sum[n][k]);
}
```

Листинг 10.17. Поиск подходящего родительского элемента

```
void report_subset(int n, int k) {
   if (k == 0) {
        return;
   }

   if (parent[n][k] == NIL) {
        report_subset(n-1,k);
   }
   else {
        report_subset(n-1,parent[n][k]);
           printf(" %d ",k-parent[n][k]);
   }
}
```

Листинг 10.18. Реализация алгоритма решения задачи линейного разбиения

```
void partition(int s[], int n, int k) {
    int p[MAXN+1];              /* prefix sums array */
    int m[MAXN+1][MAXK+1];      /* DP table for values */
    int d[MAXN+1][MAXK+1];      /* DP table for dividers */
    int cost;                   /* test split cost */
    int i,j,x;                  /* counters */

    p[0] = 0;                   /* construct prefix sums */
    for (i = 1; i <= n; i++) {
        p[i] = p[i-1] + s[i];
    }
    for (i = 1; i <= n; i++) {
        m[i][1] = p[i];     /* initialize boundaries */
    }

    for (j = 1; j <= k; j++) {
        m[1][j] = s[1];
    }

    for (i = 2; i <= n; i++) {   /* evaluate main recurrence */
        for (j = 2; j <= k; j++) {
            m[i][j] = MAXINT;
            for (x = 1; x <= (i-1); x++) {
                cost = max(m[x][j-1], p[i]-p[x]);
                if (m[i][j] > cost) {
                    m[i][j] = cost;
                    d[i][j] = x;
                }
            }
        }
    }
    reconstruct_partition(s, d, n, k);     /* print book partition */
}
```

Листинг 10.19. Рекурсивная процедура восстановления решения

```
void reconstruct_partition(int s[],int d[MAXN+1][MAXK+1], int n, int k) {
    if (k == 1) {
        print_books(s, 1, n);
    } else {
        reconstruct_partition(s, d, d[n][k], k-1);
        print_books(s, d[n][k]+1, n);
    }
}


void print_books(int s[], int start, int end) {
    int i;    /* counter */

    printf("\{");
    for (i = start; i <= end; i++) {
        printf(" %d ", s[i]);
    }
    printf("}\n");
}
```

Листинг 12.4. Процедура произвольного выбора решений

```
void random_sampling(tsp_instance *t, int nsamples, tsp_solution *s) {
    tsp_solution s_now;             /* current tsp solution */
    double best_cost;               /* best cost so far */
    double cost_now;                /* current cost */
    int i;                          /* counter */

    initialize_solution(t->n, &s_now);
    best_cost = solution_cost(&s_now, t);
    copy_solution(&s_now, s);

    for (i = 1; i <= nsamples; i++) {
        random_solution(&s_now);
        cost_now = solution_cost(&s_now, t);

        if (cost_now < best_cost) {
            best_cost = cost_now;
            copy_solution(&s_now, s);
        }

        solution_count_update(&s_now, t);
    }
}
```

Листинг 12.6. Процедура восхождения по выпуклой поверхности

```
void hill_climbing(tsp_instance *t, tsp_solution *s) {
    double cost;            /* best cost so far */
    double delta;           /* swap cost */
    int i, j;               /* counters */
    bool stuck;             /* did I get a better solution? */

    initialize_solution(t->n, s);
    random_solution(s);
    cost = solution_cost(s, t);

    do {
        stuck = true;
        for (i = 1; i < t->n; i++) {
            for (j = i + 1; j <= t->n; j++) {
                delta = transition(s, t, i, j);
                if (delta < 0) {
                    stuck = false;
                    cost = cost + delta;
                } else {
                    transition(s, t, j, i);
                }
                solution_count_update(s, t);
            }
        }
    } while (stuck);
}
```

Листинг 12.8. Реализация метода имитации отжига

```
void anneal(tsp_instance *t, tsp_solution *s) {
    int x, y;                       /* pair of items to swap */
    int i, j;                       /* counters */
    bool accept_win, accept_loss;   /* conditions to accept transition */
    double temperature;             /* the current system temp */
    double current_value;           /* value of current state */
    double start_value;             /* value at start of loop */
    double delta;                   /* value after swap */
    double exponent;                /* exponent for energy funct */

    temperature = INITIAL_TEMPERATURE;

    initialize_solution(t->n, s);
    current_value = solution_cost(s, t);

    for (i = 1; i <= COOLING_STEPS; i++) {
        temperature *= COOLING_FRACTION;

        start_value = current_value;

        for (j = 1; j <= STEPS_PER_TEMP; j++) {
            /* pick indices of elements to swap */
            x = random_int(1, t->n);
            y = random_int(1, t->n);

            delta = transition(s, t, x, y);
            accept_win = (delta < 0);       /* did swap reduce cost? */

            exponent = (-delta / current_value) / (K * temperature);
            accept_loss = (exp(exponent) > random_float(0,1));

            if (accept_win || accept_loss) {
                current_value += delta;
            } else {
                transition(s, t, x, y);     /* reverse transition */
            }
            solution_count_update(s, t);
        }

        if (current_value < start_value) {  /* rerun at this temp */
            temperature /= COOLING_FRACTION;
        }
    }
}
```

Листинг 17.1. Алгоритм тасования Фишера — Йейтса

for $i = 1$ to n do $a[i] = i$;
for $i = 1$ to $n - 1$ do swap$[a[i], a[\text{Random}[i, n]]]$;

Листинг 17.2. Алгоритм тасования Фишера — Йейтса (вариант)

for $i = 1$ to n do $a[i] = i$;
for $i = 1$ to $n - 1$ do swap$[a[i], a[\text{Random}[1, n]]]$;