

# Team Contest Reference

## getRandomNumber(){return 4;}

Universität zu Lübeck

23. Oktober 2013

## Inhaltsverzeichnis

<b>1</b>	<b>Mathematische Algorithmen</b>	<b>1</b>
1.1	Primzahlen . . . . .	1
1.1.1	Sieb des Eratosthenes . . . . .	1
1.1.2	Primzahlentest . . . . .	1
1.2	Binomial Koeffizient . . . . .	1
1.3	Modulare Arithmetik . . . . .	1
1.3.1	Erweiterter Euklidischer Algorithmus . . . . .	2
1.4	Matrixmultiplikation . . . . .	2
<b>2</b>	<b>Datenstrukturen</b>	<b>2</b>
2.1	Fenwick Tree (Binary Indexed Tree) . . . . .	2
<b>3</b>	<b>Graphenalgorithmen</b>	<b>2</b>
3.1	Topologische Sortierung . . . . .	2
3.2	Minimum Spanning Tree . . . . .	3
3.2.1	Prim's Algorithm . . . . .	3
3.2.2	Union and Find: Kruskal's Algorithm . . . . .	4
3.3	Maximaler Fluss (Ford-Fulkerson) . . . . .	4
3.4	Floyd-Warshall . . . . .	6
3.5	Dijkstra . . . . .	6
3.6	Bellmann-Ford . . . . .	7
<b>4</b>	<b>Geometrische Algorithmen</b>	<b>7</b>
4.1	Rotate a Point . . . . .	7
4.2	Graham Scan (Convex Hull) . . . . .	7
4.3	Maximum Distance in a Point Set . . . . .	8
4.4	Area of a Polygon . . . . .	8
4.5	Punkt in Polygon . . . . .	8
<b>5</b>	<b>Verschiedenes</b>	<b>9</b>
5.1	Potenzmenge . . . . .	9
5.2	Longest Common Subsequence . . . . .	9
5.3	Longest Increasing Subsequence . . . . .	10
<b>6</b>	<b>Eine kleine C-Referenz</b>	<b>10</b>

## 1 Mathematische Algorithmen

### 1.1 Primzahlen

Für Primzahlen gilt immer (aber nicht nur für Primzahlen)

$$a^p \equiv a \pmod{p} \quad \text{bzw.} \quad a^{p-1} \equiv 1 \pmod{p}.$$

Ein paar Primzahlen für den Hausgebrauch: 1000003, 2147483648( $2^{31}$ ), 4294967291( $2^{32}$ ), ... ( $2^{63}$ )

### 1.1.1 Sieb des Eratosthenes

```

1 static boolean[] sieve(int until) {
2     boolean[] a = new boolean[until + 1];
3     Arrays.fill(a, true);
4     for (int i = 2; i < Math.sqrt(a.length); i++) {
5         if (a[i]) {
6             for (int j = i * i; j < a.length; j += i) a[j] = false;
7         }
8     }
9     return a; // a[i] == true, iff. i is prime. a[0] is ignored
10 }

```

### 1.1.2 Primzahlentest

```

1 static boolean isPrim(int p) {
2     if (p < 2 || p > 2 && p % 2 == 0) return false;
3     for (int i = 3; i <= Math.sqrt(p); i += 2)
4         if (p % i == 0) return false;
5     return true;
6 }

```

## 1.2 Binomial Koeffizient

```

1 static int[][] mem = new int[MAX_N][(MAX_N + 1) / 2];
2 static int binoCo(int n, int k) {
3     if (k < 0 || k > n) return 0;
4     if (2 * k > n) binoCo(n, n - k);
5     if (mem[n][k] > 0) return mem[n][k];
6     int ret = 1;
7     for (int i = 1; i <= k; i++) {
8         ret *= n - k + i;
9         ret /= i;
10    mem[n][i] = ret;
11 }
12 return ret;
13 }

```

## 1.3 Modulare Arithmetik

Bedeutung der größten gemeinsamen Teiler:

$$d = \text{ggT}(a, b) = as + bt$$

Verwendung zu Berechnung des inversen Elements  $b$  zu  $a$  bezüglich einer Restklassengruppe  $n$  ( $a$  und  $n$  müssen teilerfremd sein):

$$a \equiv 1 \pmod n \Leftrightarrow s \equiv b \pmod n \quad \text{für } 1 = \text{ggT}(a, n)$$

### 1.3.1 Erweiterter Euklidischer Algorithmus

```

1 static int[] eea(int a, int b) {
2     int[] dst = new int[3];
3     if (b == 0) {
4         dst[0] = a;
5         dst[1] = 1;
6         return dst; // a, 1, 0
7     }
8     dst = eea(b, a % b);
9     int tmp = dst[2];
10    dst[2] = dst[1] - ((a / b) * dst[2]);
11    dst[1] = tmp;
12    return dst;
13 }

```

Zur Berechnung des Inversen von  $n$  im Restklassenring  $p$  gilt:  $d = \text{eea}(p, n)$ .

## 1.4 Matrixmultiplikation

Strassen-Algorithmus:  $C = AB \quad A, B, C \in R^{2^n \times 2^n}$

$$C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$$

$$C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$$

$$C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$$

$$C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$$

## 2 Datenstrukturen

### 2.1 Fenwick Tree (Binary Indexed Tree)

```

1 class FenwickTree {
2     private int[] values;
3     private int n;
4     public FenwickTree(int n) {
5         this.n = n;
6         values = new int[n];
7     }
8     public int get(int i) { //get value of i
9         int x = values[0];
10        while (i > 0) {
11            x += values[i];
12            i -= i & -i; }
13        return x;
14    }
15    public void add(int i, int x) { // add x to interval [i,n]
16        if (i == 0) values[0] += x;
17        else {
18            while (i < n) {
19                values[i] += x;
20                i += i & -i; }
21        }
22    }
23 }

```

## 3 Graphenalgorithmen

### 3.1 Topologische Sortierung

```

1 static List<Integer> topoSort(Map<Integer, List<Integer>> edges,
2     Map<Integer, List<Integer>> revedges) {
3     Queue<Integer> q = new LinkedList<Integer>();
4     List<Integer> ret = new LinkedList<Integer>();
5     Map<Integer, Integer> indeg = new HashMap<Integer, Integer>();
6     for (int v : revedges.keySet()) {
7         indeg.put(v, revedges.get(v).size());
8         if (revedges.get(v).size() == 0)
9             q.add(v);
10    }
11    while (!q.isEmpty()) {
12        int tmp = q.poll();
13        ret.add(tmp);
14        for (int dest : edges.get(tmp)) {
15            indeg.put(dest, indeg.get(dest) - 1);
16            if (indeg.get(dest) == 0)
17                q.add(dest);
18        }
19    }
20    return ret;
21 }

```

### 3.2 Minimum Spanning Tree

#### 3.2.1 Prim's Algorithm

```

1 #define WHITE 0
2 #define BLACK 1
3 #define INF INT_MAX
4
5 int baum( int **matrix, int N){
6     int i, sum = 0;
7
8     int color[N];
9     int dist[N];
10
11     // markiere alle Knoten ausser 0 als unbesucht
12     color[0] = BLACK;
13     for( i=1; i<N; i++){
14         color[i] = WHITE;
15         dist[i] = INF;
16     }

```

```

17
18 // berechne den Rand
19 for( i=1; i<N; i++){
20     if( dist[i] > matrix[i][nextIndex]){
21         dist[i] = matrix[i][nextIndex];
22     }
23 }
24
25 while( 1){
26     int nextDist = INF, nextIndex = -1;
27
28     /* Den naechsten Knoten waehlen */
29     for(i=0; i<N; i++){
30         if( color[i] != WHITE) continue;
31
32         if( dist[i] < nextDist){
33             nextDist = dist[i];
34             nextIndex = i;
35         }
36     }
37
38     /* Abbruchbedingung*/
39     if( nextIndex == -1) break;
40
41     /* Knoten in MST aufnehmen */
42     color[nextIndex] = RED;
43     sum += nextDist;
44
45     /* naechste kuerzeste Distanzen berechnen */
46     for( i=0; i<N; i++){
47         if( i == nextIndex || color[i] == BLACK ) continue;
48
49         if( dist[i] > matrix[i][nextIndex]){
50             dist[i] = matrix[i][nextIndex];
51         }
52     }
53 }
54
55 return sum;
56 }

```

### 3.2.2 Union and Find: Kruskal's Algorithm

Amortized time per operation is  $O(\alpha(n))$ .

```

1 // Only the tree root is stored. The edges must be stored separately.
2 // Path compression and union by rank
3
4 int *par = (int *) malloc(n * sizeof(int));
5 int *rank = (int *) malloc(n * sizeof(int));
6
7 // Create new forest of n vertices
8 void init(int n, int *par, int *rank) {
9     int i;
10    for (i = 1; i <= n; i++) {
11        par[i] = i; // every vertex is its own root
12        rank[i] = 0;
13    }
14 }
15
16 // Union two trees which contain x and y respectively, returns new root
17 int union(int n, int *par, int *rank, int x, int y) {
18     y = find(n, par, y);
19     x = find(n, par, x);
20     if (rank[x] > rank[y]) return par[y] = x;
21     if (rank[x] < rank[y]) return par[x] = y;
22     rank[x]++; // rank[x] == rank[y]
23     return par[y] = x;
24 }
25
26 // Find the tree root of x
27 int find(int n, int *par, int x) {
28     // if parent is not a tree root
29     if (par[x] != par[par[x]]) par[x] = find(n, par, par[x]);
30     return par[x];
31 }

```

### 3.3 Maximaler Fluss (Ford-Fulkerson)

```

1  /* die folgende Zeile anpassen! */
2
3  #define N_MAX 30*30+30
4
5  /* hier drunter nichts anfassen! */
6  /* ----- */
7  #define SIZE_MAX (N_MAX+2)
8  #define SIZE (N+2)
9  #define QUELLE (N)
10 #define SENKE (N+1)
11 extern int capacity[SIZE_MAX][SIZE_MAX];
12 extern int N;
13
14 int maxFlow();
15 void reset();
16
17
18 #include <stdio.h>
19 #include <limits.h>
20 #include <string.h>
21 #include "flow.h"
22
23 #define NONE -1
24 #define INF INT_MAX/2
25
26 int N;
27 int capacity[SIZE_MAX][SIZE_MAX];
28 int flow[SIZE_MAX][SIZE_MAX];
29 int queue[SIZE_MAX], *head, *tail;
30 int state[SIZE_MAX];
31 int pred[SIZE_MAX];
32
33 enum { UNVISITED, WAITING, PROCESSED };
34
35 void enqueue( int x){
36     *tail++ = x;
37     state[x] = WAITING;
38 }
39
40 int dequeue(){
41     int x = *head++;
42     state[x] = PROCESSED;
43     return x;
44 }
45
46 void reset(){
47     int i, j;
48     for(i=0; i<SIZE;i++){
49         memset( capacity[i], 0, sizeof(int)*SIZE );
50     }
51 }
52
53 int bfs( int start, int target){
54     int u, v;
55     for( u=0; u< SIZE; u++){
56         state[u] = UNVISITED;
57     }
58     head = tail = queue;
59     pred[start] = NONE;
60
61     enqueue(start);
62
63     while( head < tail){
64         u = dequeue();
65
66         for( v= 0; v< SIZE; v++){
67             if( state[v] == UNVISITED &&
68                 capacity[u][v] - flow[u][v] > 0){
69
70                 enqueue(v);
71                 pred[v] = u;
72             }
73         }
74     }
75 }

```

```

59     return state[target] == PROCESSED;
60 }
61
62 int maxFlow(){
63     int max_flow = 0;
64     int u;
65
66     int i, j;
67     for(i=0; i<SIZE;i++){
68         memset( flow[i], 0, sizeof(int)*SIZE );
69     }
70
71     while( bfs( QUELLE, SENKE)){
72         int increment = INF, temp;
73
74         for( u= SENKE; pred[u] != NONE; u = pred[u]){
75             temp = capacity[pred[u]][u] - flow[pred[u]][u];
76             if( temp < increment){
77                 increment = temp;
78             }
79         }
80
81         for( u= SENKE; pred[u] != NONE; u = pred[u]){
82             flow[pred[u]][u] += increment;
83             flow[u][pred[u]] -= increment;
84         }
85
86         max_flow += increment;
87     }
88
89     return max_flow;
90 }

```

```

1  /**
2  * Ford Fulkerson
3  * @param s source
4  * @param d destination
5  * @param c capacity
6  * @param f flow, init with 0
7  * @return
8  */
9  static int ff(int s, int d, int[][] c, int[][] f) {
10     List<Integer> path = dfs(s, d, c, f, new boolean[c.length]); // find path
11     if (path.size() < 2) {
12         int flow = 0;
13         for (int i = 0; i < f[s].length; i++) { // leaving flow of source
14             flow += f[s][i];
15         }
16         return flow;
17     }
18     int cap = Integer.MAX_VALUE; // capacity of current path
19     for (int i = 0; i < path.size() - 1; i++) {
20         int a = path.get(i), b = path.get(i + 1);
21         cap = Math.min(cap, c[a][b] - f[a][b]);
22     }
23     for (int i = 0; i < path.size() - 1; i++) { //update flow
24         int a = path.get(i), b = path.get(i + 1);
25         f[a][b] += cap;
26         f[b][a] -= cap;
27     }
28     return ff(s, d, c, f); // tail recursion
29 }
30
31 /**
32 * depth first search in flow network
33 * @param s source
34 * @param d destination
35 * @param c capacity
36 * @param f flow
37 * @param v visited, init with false
38 * @return
39 */
40 static List<Integer> dfs(int s, int d, int[][] c, int[][] f, boolean[] v) {
41     v[s] = true;
42     if (s == d) { // destination found
43         LinkedList<Integer> path = new LinkedList<Integer>();
44         path.add(d);

```

```

45     return path;
46 }
47 for (int i = 0; i < c[s].length; i++) {
48     if (!v[i] && c[s][i] - f[s][i] > 0) {
49         List<Integer> path = dfs(i, d, c, f, v);
50         if (path.size() > 0) {
51             ((LinkedList<Integer>) path).addFirst(s);
52             return path;
53         }
54     }
55 }
56 return ((List<Integer>) Collections.EMPTY_LIST);
57 }

```

### 3.4 Floyd-Warshall

```

1 static int n;
2 static int[][] path = new int[n][n];
3 static int[][] next = new int[n][n];
4 static void floyd(int[][] ad) {
5     for (int i = 0; i < n; i++)
6         path[i] = Arrays.copyOf(ad[i], n);
7     for (int i = 0; i < n; i++)
8         for (int j = 0; j < n; j++)
9             for (int k = 0; k < n; k++)
10                 if (path[i][k] + path[k][j] < path[i][j]) {
11                     path[i][j] = path[i][k] + path[k][j];
12                     next[i][j] = k;
13                 }
14     // there is a negative circle iff. there is a i such that path[i][i] < 0
15 }

```

### 3.5 Dijkstra

```

1 Funktion Dijkstra(Graph, Startknoten):
2     initialisiere(Graph, Startknoten, abstand[], vorgaenger[], Q)
3     solange Q nicht leer: // Der eigentliche Algorithmus
4         u := Knoten in Q mit kleinstem Wert in abstand[]
5         entferne u aus Q // fuer u ist der kuerzeste Weg nun bestimmt
6         fuer jeden Nachbarn v von u:
7             falls v in Q:
8                 distanz_update(u, v, abstand[], vorgaenger[]) // pruefe Abstand vom Startknoten zu v
9     return vorgaenger[]
10
11 Methode initialisiere(Graph, Startknoten, abstand[], vorgaenger[], Q):
12     fuer jeden Knoten v in Graph:
13         abstand[v] := unendlich
14         vorgaenger[v] := null
15     abstand[Startknoten] := 0
16     Q := Die Menge aller Knoten in Graph
17
18 Methode distanz_update(u, v, abstand[], vorgaenger[]):
19     alternativ := abstand[u] + abstand_zwischen(u, v) // Weglaenge vom Startknoten nach v ueber u
20     falls alternativ < abstand[v]:
21         abstand[v] := alternativ
22         vorgaenger[v] := u

```

### 3.6 Bellmann-Ford

Single source all paths, negative weights.

```

1 // returns true iff negative-weight cycle reachable
2 private static boolean bellmannford(Node start, int n, List<Edge> edges) {
3     start.dist = 0; // others: dist = Integer.MAX_VALUE
4     while (n-- > 0) { // number of nodes --> for all vertices
5         for (Edge edge : edges) { // --> for all edges
6             if (edge.from.dist < Integer.MAX_VALUE
7                 && edge.from.dist + edge.w < edge.to.dist)
8                 edge.to.dist = edge.from.dist + edge.w; // update predecessor
9         }
10        for (Edge edge : edges) {
11            if (edge.from.dist < Integer.MAX_VALUE
12                && edge.from.dist + edge.w < edge.to.dist)
13                return true;
14        }
15    }
16    return false;

```

```

16 }
17 class Node {}
18 class Edge {
19     Node from, to;
20     int w;
21     public Edge(Node from, Node to, int w) {
22         this.from = from; this.to = to; this.w = w;
23     }
24 }

```

## 4 Geometrische Algorithmen

### 4.1 Rotate a Point

```

1 static P rotate(P origin, P p, double ccw) {
2     double x = (p.x - origin.x) * Math.cos(ccw) - (p.y - origin.y) * Math.sin(ccw);
3     double y = (p.x - origin.x) * Math.sin(ccw) + (p.y - origin.y) * Math.cos(ccw);
4     return new P(x, y);
5 }

```

### 4.2 Graham Scan (Convex Hull)

```

1 class P {
2     double x, y;
3
4     P(double x, double y) {
5         this.x = x;
6         this.y = y;
7     }
8     // polar coordinates (not used in graham scan)
9     double r() { return Math.sqrt(x * x + y * y); }
10    double d() { return Math.atan2(y, x); }
11 }
12
13 // turn is counter-clockwise if > 0; collinear if = 0; clockwise else
14 static double ccw(P p1, P p2, P p3) {
15     return (p2.x - p1.x) * (p3.y - p1.y) - (p2.y - p1.y) * (p3.x - p1.x);
16 }
17
18 static List<P> graham(List<P> l) {
19     if (l.size() < 3)
20         return l;
21     P temp = l.get(0);
22     for (P p : l)
23         if (temp.y > p.y || temp.y == p.y && temp.x > p.x)
24             temp = p;
25     final P start = temp; // min y (then leftmost)
26
27     Collections.sort(l, new Comparator<P>() {
28         public int compare(P o1, P o2) {
29             if (new Double(Math.atan2(o1.y - start.y, o1.x - start.x)) // same angle
30                 .compareTo(Math.atan2(o2.y - start.y, o2.x - start.x)) == 0)
31                 return new Double((o1.x - start.x) * (o1.x - start.x)
32                     + (o1.y - start.y) * (o1.y - start.y))
33                     .compareTo((o2.x - start.x) * (o2.x - start.x)
34                     + (o2.y - start.y) * (o2.y - start.y)); // use distance
35             return new Double(Math.atan2(o1.y - start.y, o1.x - start.x))
36                 .compareTo(Math.atan2(o2.y - start.y, o2.x - start.x));
37         }
38     });
39     Stack<P> s = new Stack<P>();
40     s.add(start);
41     s.add(l.get(1));
42     for (int i = 2; i < l.size(); i++) {
43         while (s.size() >= 2
44             && ccw(s.get(s.size() - 2), s.get(s.size() - 1), l.get(i)) <= 0)
45             s.pop();
46         s.push(l.get(i));
47     }
48     return s;
49 }

```

### 4.3 Maximum Distance in a Point Set



```

1 List<P> hull = graham(list);
2 maxDist(hull);
3
4 static double dist(P p1, P p2) {
5     return Math.sqrt((p1.x - p2.x) * (p1.x - p2.x)
6         + (p1.y - p2.y) * (p1.y - p2.y));
7 }
8
9 static double maxDist(List<P> hull) {
10     double max = 0, tmp = 0;
11     int j = 0, n = hull.size();
12     for (P p : hull) {
13         while (tmp < dist(p, hull.get((j + 1) % n))) {
14             j = (j + 1) % n;
15             tmp = dist(p, hull.get(j));
16         }
17         max = Math.max(max, tmp);
18     }
19     return max;
20 }

```

## 4.4 Area of a Polygon

```

1 // area of a polygon, e.g. area(gham(list))
2 static double area(List<P> l) {
3     double sum = 0;
4     // points must be in ccw order, otherwise negative area returned
5     for (int i = 0; i < l.size(); i++) {
6         sum += l.get(i).x * l.get((i + 1) % l.size()).y;
7         sum -= l.get(i).y * l.get((i + 1) % l.size()).x;
8     }
9     return sum / 2;
10 }

```

## 4.5 Punkt in Polygon

```

1 /**
2  * -1: A liegt links von BC (ausser unterer Endpunkt)
3  * 0: A auf BC
4  * +1: sonst
5  */
6 public static int KreuzProdTest(double ax, double ay, double bx, double by,
7     double cx, double cy) {
8     if (ay == by && by == cy) {
9         if ((bx <= ax && ax <= cx) || (cx <= ax && ax <= bx)) return 0;
10        else return +1;
11    }
12    if (by > cy) {
13        double tmpx = bx, tmpy = by;
14        bx = cx;
15        by = cy;
16        cx = tmpx;
17        cy = tmpy;
18    }
19    if (ay == by && ax == bx) return 0;
20    if (ay <= by || ay > cy) return +1;
21    double delta = (bx - ax) * (cy - ay) - (by - ay) * (cx - ax);
22    if (delta > 0) return -1;
23    else if (delta < 0) return +1;
24    else return 0;
25 }
26
27 /**
28  * Input: P[i] (x[i],y[i]); P[0]:=P[n]
29  * -1: Q ausserhalb Polygon
30  * 0: Q auf Polygon
31  * +1: Q innerhalb des Polygons
32  */
33 public static int PunktInPoly(double[] x, double[] y, double qx, double qy) {
34     int t = -1;
35     for (int i = 0; i < x.length - 1; i++)
36         t = t * KreuzProdTest(qx, qy, x[i], y[i], x[i + 1], y[i + 1]);
37     return t;
38 }

```

## 5 Verschiedenes

### 5.1 Potenzmenge

```

1 static <T> Iterator<List<T>> powerSet(final List<T> l) {
2     return new Iterator<List<T>>() {
3         int i; // careful: i becomes 2^l.size()
4         public boolean hasNext() {
5             return i < (1 << l.size());
6         }
7         public List<T> next() {
8             Vector<T> temp = new Vector<T>();
9             for (int j = 0; j < l.size(); j++)
10                 if (((i >>> j) & 1) == 1)
11                     temp.add(l.get(j));
12             i++;
13             return temp;
14         }
15         public void remove() {}
16     };
17 }

```

### 5.2 Longest Common Subsequence

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5
6 int lcs( char *a, char *b){
7     int len = strlen( a);
8     int lenb =strlen(b);
9
10    int *zeile = malloc( (len+1) * sizeof(int)), *temp,
11        *neue = malloc( (len+1) * sizeof(int)), i, j;
12
13    for(i=0; i<len+1; i++){
14        zeile[i] = neue[i] = 0;
15    }
16
17    for(j=0; j<lenb; j++){
18        for(i=0; i<len; i++){
19            if( a[i] == b[j]){
20                neue[i+1] = zeile[i] + 1;
21            } else {
22                neue[i+1] = neue[i] > zeile[i+1] ? neue[i] : zeile[i+1];
23            }
24        }
25        temp = zeile;
26        zeile = neue;
27        neue = temp;
28    }
29
30    int res = zeile[len];
31    free( zeile);
32    free( neue);
33    return res;
34 }

```

### 5.3 Longest Increasing Subsequence

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int lis( int *list, int n){
5     int *sorted = malloc( n*sizeof(int)), sorted_n;
6     int i, *lower, *upper, *mid, *pos;
7
8     if( n == 0) return 0;
9
10    sorted[0] = list[0];
11    sorted_n = 1;
12
13    for( i=1; i<n; i++){
14        /* binaere Suche */

```

```
15     lower = list;
16     upper = list + sorted_n;
17     mid = list + sorted_n / 2;
18
19
20     while( lower < upper-1){
21         if( list[i] < *mid){
22             upper = mid;
23         } else {
24             lower = mid;
25         }
26
27         mid = lower + (upper-lower) / 2;
28     }
29
30     if( mid == list + sorted_n -1 && *mid < list[i]){
31         *mid = list[i];
32         sorted_n++;
33     }
34
35     if( list[i] < *mid){
36         *mid = list[i];
37     }
38 }
39
40 free( sorted);
41
42 return sorted_n;
43 }
```

## 6 Eine kleine C-Referenz

## C Reference Card (ANSI)

### Program Structure/Functions

```

type func(type_1,...)
    function declarations
    external variable declarations
main() {
    declarations
    statements
}
type func(arg_1,...) {
    function definition
    local variable declarations
    statements
    return value;
}
/* */
main(int argc, char *argv[])
    comments
    main with args
    terminate execution

```

### C Preprocessor

```

#include <filename>
#include "filename"
#define name text
#define name(var) text
Example. #define max(A,B) ((A)>(B) ? (A) : (B))
#undef name
#
#
#if, #else, #elif, #endif
#ifdef, #ifndef
defined(name)
\

```

### Data Types/Declarations

```

character (1 byte)
integer
float (single precision)
float (double precision)
short (16 bit integer)
long (32 bit integer)
positive and negative
only positive
pointer to int, float,...
enum
constant (unchanging) value
declare external variable
register
static
local to source file
no value
structure
create name by data type
size of an object (type is size_t)
size of a data type (type is size_t)

```

### Initialization

```

initialize variable
initialize array
initialize char string
    type name=value
    type name[]={value_1,...}
    char name[]="string"

```

### Constants

```

long (suffix)
float (suffix)
exponential form
octal (prefix zero)
hexadecimal (prefix zero-ex)
character constant (char, octal, hex)
newline, cr, tab, backspace
special characters
string constant (ends with '\0')

```

### Pointers, Arrays & Structures

```

declare pointer to type
declare function returning pointer to type type *f()
declare pointer to function returning type type (*pf)()
generic pointer type
void *
NULL
object pointed to by pointer
address of object name
array
multi-dim array
name[dim_1][dim_2]...
name[dim_1]
name[dim_1][dim_2]...

```

### Structures

```

struct tag {
    declarations
};
create structure
member of structure from template
member of pointed to structure
Example. (*p).x and p->x are the same
single value, multiple type structure
union
member : b

```

### Operators (grouped by precedence)

```

structure member operator
structure pointer
increment, decrement
plus, minus, logical not, bitwise not
indirection via pointer, address of object
cast expression to type
size of an object
multiply, divide, modulus (remainder)
add, subtract
left, right shift [bit ops]
comparisons
>, >=, <, <=
comparisons
==, !=
bitwise and
&
bitwise exclusive or
^
bitwise or (incl)
|
logical and
&&
logical or
||
conditional expression
expr1 ? expr2 : expr3
assignment operators
+=, -=, *=, ...
expression evaluation separator
,
Unary operators, conditional expression and assignment operators
group right to left; all others group left to right.

```

### Flow of Control

```

statement terminator
block delimiters
exit from switch, while, do, for
next iteration of while, do, for
goto label
label:
return expr
Flow Constructions
if statement
    if (expr) statement
    else if (expr) statement
    else statement
while (expr)
    statement
for (expr_1; expr_2; expr_3)
    statement
do statement
while(expr);
switch statement
    switch (expr) {
        case const_1: statement_1 break;
        case const_2: statement_2 break;
        default: statement
    }

```

### ANSI Standard Libraries

```

<assert.h> <ctype.h> <errno.h> <float.h> <limits.h>
<locale.h> <math.h> <setjmp.h> <signal.h> <stdarg.h>
<stddef.h> <stdio.h> <stdlib.h> <string.h> <time.h>

```

### Character Class Tests <ctype.h>

```

isalnum(c)
isalpha(c)
isascii(c)
isblank(c)
iscntrl(c)
isdigit(c)
islower(c)
isprint(c)
ispunct(c)
isspace(c)
isupper(c)
isxdigit(c)
tolower(c)
toupper(c)

```

### String Operations <string.h>

```

s, t are strings, cs, ct are constant strings
length of s
strcpy(s, ct)
strncpy(s, ct, n)
strcat(s, ct)
strncat(s, ct, n)
strcmp(cs, ct)
strncmp(cs, ct, n)
strchr(cs, c)
strrchr(cs, c)
memcpy(s, ct, n)
memmove(s, ct, n)
memcmp(cs, ct, n)
memchr(cs, c, n)
memset(s, c, n)

```

## C Reference Card (ANSI)

### Input/Output <stdio.h>

#### Standard I/O

standard input stream  
standard output stream  
standard error stream  
end of file  
get a character  
print a character  
print formatted data  
print to string *s*  
read formatted data  
read from string *s*  
read line to string *s* (< max chars)  
print string *s*  
**File I/O**  
declare file pointer  
pointer to named file  
modes: *r* (read), *w* (write), *a* (append)  
get a character  
write a character  
write to file  
read from file  
close file  
non-zero if error  
non-zero if EOF  
read line to string *s* (< max chars)  
write string *s*  
**Codes for Formatted I/O: "%-+ 0w.pmic"**  
- left justify  
+ print with sign  
*space* print space if no sign  
0 pad with leading zeros  
*w* min field width  
*p* precision  
*m* conversion character:  
  *h* short, *l* long, *L* long double  
*c* conversion character:  
  *d,i* integer  
  *u* unsigned  
  *c* single char  
  *s* char string  
  *f* double  
  *e,E* exponential  
  *o* octal  
  *x,X* hexadecimal  
  *p* pointer  
  *n* number of chars written  
*g,G* same as *f* or *e,E* depending on exponent

### Variable Argument Lists <stdarg.h>

declaration of pointer to arguments *va\_list name*;  
initialization of argument pointer *va\_start(name, lastarg)*  
*lastarg* is last named parameter of the function  
access next unnamed arg, update pointer *va\_arg(name, type)*  
call before exiting function *va\_end(name)*

### Standard Utility Functions <stdlib.h>

absolute value of int *n*  
absolute value of long *n*  
quotient and remainder of ints *n,d*  
return structure with *div\_t.quot* and *div\_t.rem*  
quotient and remainder of longs *n,d*  
returns structure with *ldiv\_t.quot* and *ldiv\_t.rem*  
pseudo-random integer [0, RAND\_MAX]  
rand()  
set random seed to *n*  
terminate program execution  
pass string *s* to system for execution  
**Conversions**  
convert string *s* to double  
convert string *s* to integer  
convert string *s* to long  
convert string *s* to long  
convert prefix of *s* to double  
convert prefix of *s* (base *b*) to long  
same, but unsigned long  
strtoul(*s*, endp, *b*)

#### Storage Allocation

allocate storage  
change size of object  
deallocate space  
**Array Functions**  
search array for key  
sort array ascending order  
malloc(*size*), calloc(*nobj*, *size*)  
realloc(*pts*, *size*)  
free(*ptr*)

### Time and Date Functions <time.h>

processor time used by program  
Example: clock()/CLOCKS\_PER\_SEC is time in seconds  
current calendar time  
time2-time1 in seconds (double)  
arithmetic types representing times  
structure type for calendar time comps  
tm\_sec seconds after minute  
tm\_min minutes after hour  
tm\_hour hours since midnight  
tm\_mday day of month  
tm\_mon months since January  
tm\_year years since 1900  
tm\_wday days since Sunday  
tm\_yday days since January 1  
tm\_isdst Daylight Savings Time flag  
convert local time to calendar time  
convert time in *tp* to string  
convert calendar time in *tp* to local time  
convert calendar time to GMT  
convert calendar time to local time  
format date and time info  
*tp* is a pointer to a structure of type *tm*

### Mathematical Functions <math.h>

Arguments and returned values are double  
trig functions  
inverse trig functions  
atan(*y/x*)  
hyperbolic trig functions  
exponentials & logs  
exponentials & logs (2 power)  
division & remainder  
powers  
rounding

### Integer Type Limits <limits.h>

The numbers given in parentheses are typical values for the constants on a 32-bit Unix system.

CHAR\_BIT bits in char (8)  
CHAR\_MAX max value of char (127 or 255)  
CHAR\_MIN min value of char (-128 or 0)  
INT\_MAX max value of int (+32,767)  
INT\_MIN min value of int (-32,768)  
LONG\_MAX max value of long (+2,147,483,647)  
LONG\_MIN min value of long (-2,147,483,648)  
SHAR\_MAX max value of signed char (+127)  
SHAR\_MIN min value of signed char (-128)  
SHRT\_MAX max value of short (+32,767)  
SHRT\_MIN min value of short (-32,768)  
UCHAR\_MAX max value of unsigned char (255)  
UINT\_MAX max value of unsigned int (65,535)  
ULONG\_MAX max value of unsigned long (4,294,967,295)  
USHRT\_MAX max value of unsigned short (65,536)

### Float Type Limits <float.h>

FLT\_RADIX radix of exponent rep (2)  
FLT\_ROUNDS floating point rounding mode (6)  
FLT\_DIG decimal digits of precision (10-5)  
FLT\_EPSILON smallest *x* so  $1.0 + x \neq 1.0$   
FLT\_MANT\_DIG number of digits in mantissa (10<sup>37</sup>)  
FLT\_MAX maximum floating point number  
FLT\_MAX\_EXP maximum exponent (10-37)  
FLT\_MIN minimum floating point number  
FLT\_MIN\_EXP minimum exponent (10)  
DBL\_DIG decimal digits of precision (10-9)  
DBL\_EPSILON smallest *x* so  $1.0 + x \neq 1.0$   
DBL\_MANT\_DIG number of digits in mantissa (10<sup>37</sup>)  
DBL\_MAX max double floating point number  
DBL\_MAX\_EXP maximum exponent (10-37)  
DBL\_MIN min double floating point number  
DBL\_MIN\_EXP minimum exponent

May 1999 v1.3. Copyright © 1999 Joseph H. Silverman

Permission is granted to make and distribute copies of this card provided the copyright notice and this permission notice are preserved on all copies.

Send comments and corrections to J.H. Silverman, Math. Dept., Brown Univ., Providence, RI 02912 USA. (jhs@math.brown.edu)