Team Contest Reference
Ballmer Peak

Universität zu Lübeck

13. November 2013

Inhaltsverzeichnis

1 Mathematische Algorithmen

1.1 Primzahlen

Für Primzahlen gilt immer (aber nicht nur für Primzahlen)

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

Ein paar Primzahlen für den Hausgebrauch: $1000003, 2147483647(2^{31}), 4294967291(2^{32})$

1.1.1 Sieb des Eratosthenes

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

1.1.2 Primzahlentest

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

1.2 Binomial Koeffizient

```
static int[][] mem = new int[MAX_N][(MAX_N + 1) / 2];
static int binoCo(int n, int k) {
  if (k < 0 || k > n) return 0;
  if (2 * k > n) binoCo(n, n - k);
  if (mem[n][k] > 0) return mem[n][k];
  int ret = 1;
  for (int i = 1; i <= k; i++) {
    ret *= n - k + i;
    ret /= i;
    mem[n][i] = ret;
  }
  return ret;
}
```

1.3 Modulare Arithmetik

Bedeutung der größten gemeinsamen Teiler:

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

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

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

1.3.1 Erweiterter Euklidischer Algorithmus

```
static int[] eea(int a, int b) {
  int[] dst = new int[3];
  if (b == 0) {
    dst[0] = a;
    dst[1] = 1;
    return dst; // a, 1, 0
  }
  dst = eea(b, a % b);
  int tmp = dst[2];
  dst[2] = dst[1] - ((a / b) * dst[2]);
  dst[1] = tmp;
  return dst;
}
```

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

1.4 Matrixmultiplikation

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

$$\begin{aligned}
\mathbf{C}_{1,1} &= \mathbf{A}_{1,1}\mathbf{B}_{1,1} + \mathbf{A}_{1,2}\mathbf{B}_{2,1} \\
\mathbf{C}_{1,2} &= \mathbf{A}_{1,1}\mathbf{B}_{1,2} + \mathbf{A}_{1,2}\mathbf{B}_{2,2} \\
\mathbf{C}_{2,1} &= \mathbf{A}_{2,1}\mathbf{B}_{1,1} + \mathbf{A}_{2,2}\mathbf{B}_{2,1} \\
\mathbf{C}_{2,2} &= \mathbf{A}_{2,1}\mathbf{B}_{1,2} + \mathbf{A}_{2,2}\mathbf{B}_{2,2}
\end{aligned}$$

2 Datenstukturen

2.1 Fenwick Tree (Binary Indexed Tree)

```
class FenwickTree {
  private int[] values;
  private int n;
  public FenwickTree(int n) {
    this.n = n;
    values = new int[n];
  }
  public int get(int i) { //get value of i
    int x = values[0];
    while (i > 0) {
      x += values[i];
      i -= i & -i; }
    return x;
  }
  public void add(int i, int x) { // add x to interval [i,n]
    if (i == 0) values[0] += x;
    else {
      while (i < n) {
        values[i] += x;
        i += i & -i; }
    }
  }
}
```

3 Graphenalgorithmen

3.1 Topologische Sortierung

```
static List<Integer> topoSort(Map<Integer, List<Integer>> edges,
    Map<Integer, List<Integer>> revedges) {
  Queue<Integer> q = new LinkedList<Integer>();
  List<Integer> ret = new LinkedList<Integer>();
  Map<Integer, Integer> indeg = new HashMap<Integer, Integer>();
  for (int v : revedges.keySet()) {
    indeg.put(v, revedges.get(v).size());
    if (revedges.get(v).size() == 0)
      q.add(v);
  }
  while (!q.isEmpty()) {
    int tmp = q.poll();
    ret.add(tmp);
    for (int dest : edges.get(tmp)) {
      indeg.put(dest, indeg.get(dest) - 1);
      if (indeg.get(dest) == 0)
        q.add(dest);
    }
  }
  return ret;
}
```

3.2 Minimum Spanning Tree

3.2.1 Prim's Algorithm

```
#define WHITE 0
#define BLACK 1
#define INF INT_MAX

int baum( int **matrix, int N){
  int i, sum = 0;

  int color[N];
  int dist[N];

    // markiere alle Knoten ausser 0 als unbesucht
  color[0] = BLACK;
  for( i=1; i<N; i++){
    color[i] = WHITE;
    dist[i] = INF;
  }
```

```

    // berechne den Rand
  for( i=1; i<N; i++){
      if( dist[i] > matrix[i][nextIndex]){
          dist[i] = matrix[i][nextIndex];
      }
    }

  while( 1){
    int nextDist = INF, nextIndex = -1;

    /* Den naechsten Knoten waehlen */
    for(i=0; i<N; i++){
      if( color[i] != WHITE) continue;

      if( dist[i] < nextDist){
        nextDist = dist[i];
        nextIndex = i;
      }
    }

    /* Abbruchbedingung*/
    if( nextIndex == -1) break;

    /* Knoten in MST aufnehmen */
    color[nextIndex] = RED;
    sum += nextDist;

    /* naechste kuerzeste Distanzen berechnen */
    for( i=0; i<N; i++){
          if( i == nextIndex || color[i] == BLACK ) continue;

          if( dist[i] > matrix[i][nextIndex]){
              dist[i] = matrix[i][nextIndex];
          }
    }
  }

  return sum;
}
```

3.2.2 Union and Find: Kruskal's Algorithm

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

```
// Only the tree root is stored. The edges must be stored separately.
// Path compression and union by rank

int *par = (int *) malloc(n * sizeof(int));
int *rank = (int *) malloc(n * sizeof(int));

// Create new forest of n vertices
void init(int n, int *par, int *rank) {
  int i;
  for (i = 1; i <= n; i++) {
    par[i] = i; // every vertex is its on root
    rank[i] = 0;
  }
}

// Union two trees which contain x and y respectively, returns new root
int union(int n, int *par, int *rank, int x, int y) {
  y = find(n, par, y);
  x = find(n, par, x);
  if (rank[x] > rank[y]) return par[y] = x;
  if (rank[x] < rank[y]) return par[x] = y;
  rank[x]++; // rank[x] == rank[y]
  return par[y] = x;
}

// Find the tree root of x
int find(int n, int *par, int x) {
  // if parent is not a tree root
  if (par[x] != par[par[x]]) par[x] = find(n, par, par[x]);
  return par[x];
}
```

3.3 Maximaler Fluss (Ford-Fulkerson)

```
/* die folgende Zeile anpassen! */

#define N_MAX 30*30+30

/* hier drunter nichts anfassen! */
/* ----------------------------- */
#define SIZE_MAX (N_MAX+2)
#define SIZE (N+2)
#define QUELLE (N)
#define SENKE (N+1)
extern int capacity[SIZE_MAX][SIZE_MAX];
extern int N;

int maxFlow();
void reset();
```

```
#include <stdio.h>
#include <limits.h>
#include <string.h>
#include "flow.h"

#define NONE -1
#define INF INT_MAX/2

int N;
int capacity[SIZE_MAX][SIZE_MAX];
int flow[SIZE_MAX][SIZE_MAX];
int queue[SIZE_MAX], *head, *tail;
int state[SIZE_MAX];
int pred[SIZE_MAX];

enum { UNVISITED, WAITING, PROCESSED };

void enqueue( int x){
    *tail++ = x;
    state[x] = WAITING;
}

int dequeue(){
    int x = *head++;
    state[x] = PROCESSED;
    return x;
}

void reset(){
    int i, j;
    for(i=0; i<SIZE;i++){
        memset( capacity[i], 0, sizeof(int)*SIZE );
    }
}

int bfs( int start, int target){
    int u, v;
    for( u=0; u< SIZE; u++){
        state[u] = UNVISITED;
    }
    head = tail = queue;
    pred[start] = NONE;

    enqueue(start);

    while( head < tail){
        u = dequeue();

        for( v= 0; v< SIZE; v++){
            if( state[v] == UNVISITED &&
                capacity[u][v] - flow[u][v] > 0){

                enqueue(v);
                pred[v] = u;
            }
        }
    }

```

```
    return state[target] == PROCESSED;
}

int maxFlow(){
    int max_flow = 0;
    int u;

    int i, j;
    for(i=0; i<SIZE;i++){
        memset( flow[i], 0, sizeof(int)*SIZE );
    }

    while( bfs( QUELLE, SENKE)){
        int increment = INF, temp;

        for( u= SENKE; pred[u] != NONE; u = pred[u]){
            temp = capacity[pred[u]][u] - flow[pred[u]][u];
            if( temp < increment){
                increment = temp;
            }
        }

        for( u= SENKE; pred[u] != NONE; u = pred[u]){
            flow[pred[u]][u] += increment;
            flow[u][pred[u]] -= increment;
        }

        max_flow += increment;
    }

    return max_flow;
}
```

```
/**
 * Ford Fulkersen
 * @param s source
 * @param d destination
 * @param c capacity
 * @param f flow, init with 0
 * @return
 */
static int ff(int s, int d, int[][] c, int[][] f) {
  List<Integer> path = dfs(s, d, c, f, new boolean[c.length]); // find path
  if (path.size() < 2) {
    int flow = 0;
    for (int i = 0; i < f[s].length; i++) { // leaving flow of source
      flow += f[s][i];
    }
    return flow;
  }
  int cap = Integer.MAX_VALUE; // capacity of current path
  for (int i = 0; i < path.size() - 1; i++) {
    int a = path.get(i), b = path.get(i + 1);
    cap = Math.min(cap, c[a][b] - f[a][b]);
  }
  for (int i = 0; i < path.size() - 1; i++) { //update flow
    int a = path.get(i), b = path.get(i + 1);
    f[a][b] += cap;
    f[b][a] -= cap;
  }
  return ff(s, d, c, f); // tail recursion
}

/**
 * depth first search in flow network
 * @param s source
 * @param d destination
 * @param c capacity
 * @param f flow
 * @param v visited, init with false
 * @return
 */
static List<Integer> dfs(int s, int d, int[][] c, int[][] f, boolean[] v) {
  v[s] = true;
  if (s == d) { // destination found
    LinkedList<Integer> path = new LinkedList<Integer>();
    path.add(d);
```

```
    return path;
  }
  for (int i = 0; i < c[s].length; i++) {
    if (!v[i] && c[s][i] - f[s][i] > 0) {
      List<Integer> path = dfs(i, d, c, f, v);
      if (path.size() > 0) {
        ((LinkedList<Integer>) path).addFirst(s);
        return path;
      }
    }
  }
  return ((List<Integer>) Collections.EMPTY_LIST);
}
```

3.4 Floyd-Warshall

```
static int n;
static int[][] path = new int[n][n];
static int[][] next = new int[n][n];
static void floyd(int[][] ad) {
  for (int i = 0; i < n; i++)
    path[i] = Arrays.copyOf(ad[i], n);
  for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
      for (int k = 0; k < n; k++)
        if (path[i][k] + path[k][j] < path[i][j]) {
          path[i][j] = path[i][k] + path[k][j];
          next[i][j] = k;
        }
  // there is a negative circle iff. there is a i such that path[i][i] < 0
}
```

3.5 Dijkstra

```
HashMap<Integer, List<Edge>> graph = new HashMap<Integer, List<Edge>>();
for (int i = 0; i < n; i++) graph.put(i, new ArrayList<Edge>());
int dist[] = new int[n];
Arrays.fill(dist, Integer.MAX_VALUE);
int shortest = dijkstra(source, dest, graph, dist);

static int dijkstra(int s, int d, HashMap<Integer, List<Edge>> graph, final int[] dist) {
  dist[s] = 0;
  TreeSet<Integer> queue = new TreeSet<Integer>(
      new Comparator<Integer>() {
        public int compare(Integer o1, Integer o2) {
          if (dist[o1] == dist[o2]) return o1.compareTo(o2);
          return ((Integer) o1).compareTo(o2);
      } });
  queue.add(s);
  while (queue.size() > 0) { // || queue.first() != d) {
    int c = queue.pollFirst();
    for (Edge e : graph.get(c)) {
      if (dist[e.to] > dist[c] + e.val) {
        queue.remove(e.to);
        dist[e.to] = dist[c] + e.val;
        queue.add(e.to);
  } } }
  return dist[d];
}

class Edge {
  int from, to, val;
  public Edge(int from, int to, int val) {
    this.from = from;
    this.to = to;
    this.val = val;
} }
```

3.6 Bellmann-Ford

Single source all paths, negative weights.

```
// returns true iff negative-weight cycle reachable
private static boolean bellmannford(Node start, int n, List<Edge> edges) {
  start.dist = 0; // others: dist = Integer.MAX_VALUE
  while (n-- > 0) { // number of nodes --> for all vertices
```

```
    for (Edge edge : edges) { // --> for all edges
      if (edge.from.dist < Integer.MAX_VALUE
          && edge.from.dist + edge.w < edge.to.dist)
        edge.to.dist = edge.from.dist + edge.w; // update predecessor
  } }
  for (Edge edge : edges) {
    if (edge.from.dist < Integer.MAX_VALUE
        && edge.from.dist + edge.w < edge.to.dist)
      return true;
  }
  return false;
}
class Node {}
class Edge {
  Node from, to;
  int w;
  public Edge(Node from, Node to, int w) {
    this.from = from; this.to = to; this.w = w;
  }
}
```

3.7 Starke Zusammenhangskomponenten (Kosaraju)

```
#define POS(X,Y) ((X)+size*(Y))
#define M(X,Y) (M[POS((X),(Y))])

int *top;
int *color;

void Kosaraju( int *M, int size);
void DFS( int *M, int u, int size);
void RDFS( int *M, int u, int size, int colorN);

void Kosaraju( int *M, int size){
  int i;
  int *stack = malloc( size * sizeof(int));
  top = stack;

  for(i=0;i<size;i++)
    color[i] = 0;

  for(i=0;i<size;i++){
    if(color[i] != 0) continue;

    DFS(M,i,size);
  }

  for(i=0;i<size;i++)
    color[i] = 0;

  int colorN = 1;

  while( top > stack ){
    int v = *(--top);
    if( color[v] != 0 ) continue;
    RDFS( M, v, size, colorN++);
  }

  free( stack);
}

void DFS( int *M, int u, int size){
  int v;
  color[u] = 1;
  for(v=0;v<size;v++){
    if( M(u,v) && color[v] == 0){
      DFS( M, v, size);
    }
  }

  *top++ = u;
}

void RDFS( int *M, int u, int size, int colorN){
  int v;
```

```
  color[u] = colorN;
  for(v=0;v<size;v++){
    if( M(v,u) && color[v] == 0){
      RDFS( M, v, size, colorN);
    }
  }
}
```

4 Geometrische Algorithmen

4.1 Rotate a Point

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

4.2 Graham Scan (Convex Hull)

```
class P {
  double x, y;

  P(double x, double y) {
    this.x = x;
    this.y = y;
  }
  // polar coordinates (not used in graham scan)
  double r() { return Math.sqrt(x * x + y * y); }
  double d() { return Math.atan2(y, x); }
}

// turn is counter-clockwise if > 0; collinear if = 0; clockwise else
static double ccw(P p1, P p2, P p3) {
  return (p2.x - p1.x) * (p3.y - p1.y) - (p2.y - p1.y) * (p3.x - p1.x);
}

static List<P> graham(List<P> l) {
  if (l.size() < 3)
    return l;
  P temp = l.get(0);
  for (P p : l)
    if (temp.y > p.y || temp.y == p.y && temp.x > p.x)
      temp = p;
  final P start = temp; // min y (then leftmost)

  Collections.sort(l, new Comparator<P>() {
    public int compare(P o1, P o2) {
      if (new Double(Math.atan2(o1.y - start.y, o1.x - start.x)) // same angle
          .compareTo(Math.atan2(o2.y - start.y, o2.x - start.x)) == 0)
        return new Double((o1.x - start.x) * (o1.x - start.x)
            + (o1.y - start.y) * (o1.y - start.y))
            .compareTo((o2.x - start.x) * (o2.x - start.x)
            + (o2.y - start.y) * (o2.y - start.y)); // use distance
      return new Double(Math.atan2(o1.y - start.y, o1.x - start.x))
          .compareTo(Math.atan2(o2.y - start.y, o2.x - start.x));
    }
  });
  Stack<P> s = new Stack<P>();
  s.add(start);
  s.add(l.get(1));
  for (int i = 2; i < l.size(); i++) {
    while (s.size() >= 2
        && ccw(s.get(s.size() - 2), s.get(s.size() - 1), l.get(i)) <= 0)
      s.pop();
    s.push(l.get(i));
  }
  return s;
}
```

4.3 Maximum Distance in a Point Set

```
List<P> hull = graham(list);
maxDist(hull);
```

```java

static double dist(P p1, P p2) {
  return Math.sqrt((p1.x - p2.x) * (p1.x - p2.x)
      + (p1.y - p2.y) * (p1.y - p2.y));
}

static double maxDist(List<P> hull) {
  double max = 0, tmp = 0;
  int j = 0, n = hull.size();
  for (P p : hull) {
    for( P q : hull){
      if( p == q ) continue;
      tmp = dist(p, q);
      max = Math.max(max, tmp);
    }
  }
  return max;
}
```

4.4 Area of a Polygon

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

4.5 Punkt in Polygon

```java
/**
 * -1: A liegt links von BC (ausser unterer Endpunkt)
 * 0: A auf BC
 * +1: sonst
 */
public static int KreuzProdTest(double ax, double ay, double bx, double by,
    double cx, double cy) {
  if (ay == by && by == cy) {
    if ((bx <= ax && ax <= cx) || (cx <= ax && ax <= bx)) return 0;
    else return +1;
  }
  if (by > cy) {
    double tmpx = bx, tmpy = by;
    bx = cx;
    by = cy;
    cx = tmpx;
    cy = tmpy;
  }
  if (ay == by && ax == bx) return 0;
  if (ay <= by || ay > cy) return +1;
  double delta = (bx - ax) * (cy - ay) - (by - ay) * (cx - ax);
  if (delta > 0) return -1;
  else if (delta < 0) return +1;
  else return 0;
}

/**
 * Input: P[i] (x[i],y[i]); P[0]:=P[n]
 * -1: Q ausserhalb Polygon
 * 0: Q auf Polygon
 * +1: Q innerhalb des Polygons
 */
public static int PunktInPoly(double[] x, double[] y, double qx, double qy) {
  int t = -1;
  for (int i = 0; i < x.length - 1; i++)
    t = t * KreuzProdTest(qx, qy, x[i], y[i], x[i + 1], y[i + 1]);
  return t;
}
```

5 Verschiedenes

5.1 Potenzmenge

```
static <T> Iterator<List<T>> powerSet(final List<T> l) {
  return new Iterator<List<T>>() {
    int i; // careful: i becomes 2^l.size()
    public boolean hasNext() {
      return i < (1 << l.size());
    }
    public List<T> next() {
      Vector<T> temp = new Vector<T>();
      for (int j = 0; j < l.size(); j++)
        if (((i >>> j) & 1) == 1)
          temp.add(l.get(j));
      i++;
      return temp;
    }
    public void remove() {}
    };
  }
```

5.2 Longest Common Subsequence

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>


int lcs( char *a, char *b){
    int len = strlen( a);
    int lenb =strlen(b);

    int *zeile = malloc( (len+1) * sizeof(int)), *temp,
        *neue = malloc( (len+1) * sizeof(int)), i, j;

    for(i=0; i<len+1; i++){
        zeile[i] = neue[i] = 0;
    }

    for(j=0; j<lenb; j++){
        for(i=0; i<len; i++){
            if( a[i] == b[j]){
                neue[i+1] = zeile[i] + 1;
            } else {
                neue[i+1] = neue[i] > zeile[i+1] ? neue[i] : zeile[i+1];
            }
        }
        temp = zeile;
        zeile = neue;
        neue = temp;
    }

    int res = zeile[len];
    free( zeile);
    free( neue);
    return res;
}
```

5.3 Longest Increasing Subsequence

```
#include <stdio.h>
#include <stdlib.h>

int lis( int *list, int n){
    int *sorted = malloc( n*sizeof(int)), sorted_n;
    int i, *lower, *upper, *mid, *pos;

    if( n == 0) return 0;

    sorted[0] = list[0];
    sorted_n = 1;

    for( i=1; i<n; i++){
        /* binaere Suche */
```

```
        lower = list;
        upper = list + sorted_n;
        mid = list + sorted_n / 2;


        while( lower < upper-1){
            if( list[i] < *mid){
                upper = mid;
            } else {
                lower = mid;
            }

            mid = lower + (upper-lower) / 2;
        }

  if( mid == list + sorted_n -1 && *mid < list[i]){
            *mid = list[i];
            sorted_n++;
        }

        if( list[i] < *mid){
            *mid = list[i];
        }
    }

    free( sorted);

    return sorted_n;
}
```

6 Eine kleine C-Referenz

C Reference Card (ANSI)

Program Structure/Functions

type fnc(*type*$_1$,...)	function declarations
type name	external variable declarations
main() {	main routine
declarations	local variable declarations
statements	
}	
type fnc(*arg*$_1$,...) {	function definition
declarations	local variable declarations
statements	
return *value*;	
}	
/* */	comments
main(int argc, char *argv[])	main with args
exit(*arg*)	terminate execution

C Preprocessor

include library file	#include <*filename*>
include user file	#include "*filename*"
replacement text	#define *name text*
replacement macro	#define *name*(*var*) *text*
Example. #define max(A,B) ((A)>(B) ? (A) : (B))	
undefine	#undef *name*
quoted string in replace	#
concatenate args and rescan	##
conditional execution	#if, #else, #elif, #endif
is *name* defined, not defined?	#ifdef, #ifndef
name defined?	defined(*name*)
line continuation char	\

Data Types/Declarations

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

Initialization

initialize variable	*type name*=*value*
initialize array	*type name*[]={*value*$_1$,...}
initialize char string	char *name*[]="*string*"

1

Constants

long (suffix)	L or l
float (suffix)	F or f
exponential form	e
octal (prefix zero)	0
hexadecimal (prefix zero-ex)	0x or 0X
character constant (char, octal, hex)	'a', '*ooo*', '\x*hh*'
newline, cr, tab, backspace	\n, \r, \t, \b
special characters	\\\\, \?, \', \"
string constant (ends with '\0')	"abc...de"

Pointers, Arrays & Structures

declare pointer to *type*	*type* **name*
declare function returning pointer to *type*	*type* *f()
declare pointer to function returning *type*	*type* (*pf)()
generic pointer type	void *
null pointer	NULL
object pointed to by *pointer*	**pointer*
address of object *name*	&*name*
array	*name*[*dim*]
multi-dim array	*name*[*dim*$_1$][*dim*$_2$]...

Structures

struct *tag* {	structure template
declarations	declaration of members
};	

create structure	struct *tag name*
member of structure from template	*name*.*member*
member of pointed to structure	*pointer* -> *member*
Example. (*p).x and p->x are the same	
single value, multiple type structure	union
bit field with *b* bits	*member* : *b*

Operators (grouped by precedence)

structure member operator	*name*.*member*
structure pointer	*pointer*->*member*
increment, decrement	++, --
plus, minus, logical not, bitwise not	+, -, !, ~
indirection via pointer, address of object	**pointer*, &*name*
cast expression to type	(*type*) *expr*
size of an object	sizeof
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	*expr*$_1$? *expr*$_2$: *expr*$_3$
assignment operators	+=, -=, *=, ...
expression evaluation separator	,

Unary operators, conditional expression and assignment operators group right to left; all others group left to right.

2

Flow of Control

statement terminator	;
block delimeters	{ }
exit from switch, while, do, for	break
next iteration of while, do, for	continue
go to	goto *label*
label	*label*:
return value from function	return *expr*

Flow Constructions

if statement	if (*expr*) *statement* else if (*expr*) *statement* else *statement*
while statement	while (*expr*) *statement*
for statement	for (*expr*$_1$; *expr*$_2$; *expr*$_3$) *statement*
do 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>

alphanumeric?	isalnum(c)
alphabetic?	isalpha(c)
control character?	iscntrl(c)
decimal digit?	isdigit(c)
printing character (not incl space)?	isgraph(c)
lower case letter?	islower(c)
printing character (incl space)?	isprint(c)
printing char except space, letter, digit?	ispunct(c)
space, formfeed, newline, cr, tab, vtab?	isspace(c)
upper case letter?	isupper(c)
hexadecimal digit?	isxdigit(c)
convert to lower case?	tolower(c)
convert to upper case?	toupper(c)

String Operations <string.h>

s,t are strings, cs,ct are constant strings

length of s	strlen(s)
copy ct to s	strcpy(s,ct)
up to n chars	strncpy(s,ct,n)
concatenate ct after s	strcat(s,ct)
up to n chars	strncat(s,ct,n)
compare cs to ct	strcmp(cs,ct)
only first n chars	strncmp(cs,ct,n)
pointer to first c in cs	strchr(cs,c)
pointer to last c in cs	strrchr(cs,c)
copy n chars from ct to s	memcpy(s,ct,n)
copy n chars from ct to s (may overlap)	memmove(s,ct,n)
compare n chars of cs with ct	memcmp(cs,ct,n)
pointer to first c in first n chars of cs	memchr(cs,c,n)
put c into first n chars of cs	memset(s,c,n)

3

C Reference Card (ANSI)

Input/Output <stdio.h>

Standard I/O

standard input stream	`stdin`
standard output stream	`stdout`
standard error stream	`stderr`
end of file	`EOF`
get a character	`getchar()`
print a character	`putchar(`*chr*`)`
print formatted data	`printf("`*format*`",`*arg*$_1$`,...)`
print to string `s`	`sprintf(s,"`*format*`",`*arg*$_1$`,...)`
read formatted data	`scanf("`*format*`",&`*name*$_1$`,...)`
read from string `s`	`sscanf(s,"`*format*`",&`*name*$_1$`,...)`
read line to string `s` (< `max` chars)	`gets(s,max)`
print string `s`	`puts(s)`

File I/O

declare file pointer	`FILE *`*fp*
pointer to named file	`fopen("`*name*`","`*mode*`")`
modes: `r` (read), `w` (write), `a` (append)	
get a character	`getc(`*fp*`)`
write a character	`putc(`*chr*`,`*fp*`)`
write to file	`fprintf(`*fp*`,"`*format*`",`*arg*$_1$`,...)`
read from file	`fscanf(`*fp*`,"`*format*`",`*arg*$_1$`,...)`
close file	`fclose(`*fp*`)`
non-zero if error	`ferror(`*fp*`)`
non-zero if EOF	`feof(`*fp*`)`
read line to string `s` (< `max` chars)	`fgets(s,max,`*fp*`)`
write string `s`	`fputs(s,`*fp*`)`

Codes for Formatted I/O: `"%-+ 0`*w.pmc*`"`

`-`	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 unamed arg, update pointer	`va_arg(`*name*`,`*type*`)`
call before exiting function	`va_end(`*name*`)`

Standard Utility Functions <stdlib.h>

absolute value of `int n`	`abs(n)`
absolute value of `long n`	`labs(n)`
quotient and remainder of `int`s `n,d`	`div(n,d)`
retursn structure with `div_t.quot` and `div_t.rem`	
quotient and remainder of `long`s `n,d`	`ldiv(n,d)`
returns structure with `ldiv_t.quot` and `ldiv_t.rem`	
pseudo-random integer `[0,RAND_MAX]`	`rand()`
set random seed to `n`	`srand(n)`
terminate program execution	`exit(status)`
pass string `s` to system for execution	`system(s)`

Conversions

convert string `s` to double	`atof(s)`
convert string `s` to integer	`atoi(s)`
convert string `s` to long	`atol(s)`
convert prefix of `s` to `double`	`strtod(s,endp)`
convert prefix of `s` (base `b`) to `long`	`strtol(s,endp,b)`
same, but `unsigned long`	`strtoul(s,endp,b)`

Storage Allocation

allocate storage	`malloc(size), calloc(nobj,size)`
change size of object	`realloc(pts,size)`
deallocate space	`free(ptr)`

Array Functions

search `array` for `key`	`bsearch(key,array,n,size,cmp())`
sort `array` ascending order	`qsort(array,n,size,cmp())`

Time and Date Functions <time.h>

processor time used by program	`clock()`
Example. `clock()/CLOCKS_PER_SEC` is time in seconds	
current calendar time	`time()`
`time`$_2$`-time`$_1$ in seconds (`double`)	`difftime(time`$_2$`,time`$_1$`)`
arithmetic types representing times	`clock_t,time_t`
structure type for calendar time comps	`tm`

`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	`mktime(tp)`
convert time in `tp` to string	`asctime(tp)`
convert calendar time in `tp` to local time	`ctime(tp)`
convert calendar time to GMT	`gmtime(tp)`
convert calendar time to local time	`localtime(tp)`
format date and time info	`strftime(s,smax,"`*format*`",tp)`
`tp` is a pointer to a structure of type `tm`	

Mathematical Functions <math.h>

Arguments and returned values are `double`

trig functions	`sin(x), cos(x), tan(x)`
inverse trig functions	`asin(x), acos(x), atan(x)`
arctan(y/x)	`atan2(y,x)`
hyperbolic trig functions	`sinh(x), cosh(x), tanh(x)`
exponentials & logs	`exp(x), log(x), log10(x)`
exponentials & logs (2 power)	`ldexp(x,n), frexp(x,*e)`
division & remainder	`modf(x,*ip), fmod(x,y)`
powers	`pow(x,y), sqrt(x)`
rounding	`ceil(x), floor(x), fabs(x)`

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)
`SCHAR_MAX`	max value of `signed char`	(+127)
`SCHAR_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	
`FLT_DIG`	decimal digits of precision	(6)
`FLT_EPSILON`	smallest x so $1.0 + x \neq 1.0$	(10^{-5})
`FLT_MANT_DIG`	number of digits in mantissa	
`FLT_MAX`	maximum floating point number	(10^{37})
`FLT_MAX_EXP`	maximum exponent	
`FLT_MIN`	minimum floating point number	(10^{-37})
`FLT_MIN_EXP`	minimum exponent	
`DBL_DIG`	decimal digits of precision	(10)
`DBL_EPSILON`	smallest x so $1.0 + x \neq 1.0$	(10^{-9})
`DBL_MANT_DIG`	number of digits in mantissa	
`DBL_MAX`	max `double` floating point number	(10^{37})
`DBL_MAX_EXP`	maximum exponent	
`DBL_MIN`	min `double` floating point number	(10^{-37})
`DBL_MIN_EXP`	minimum exponent	

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