Team Contest Reference
getRandomNumber(){return 4;}

Universität zu Lübeck

20. April 2013

Inhaltsverzeichnis

1 Mathematische Algorithmen

1.1 Primzahlen

Für Primzahlen gilt immer (aber nicht nur für Primzahlen)

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

1.1.1 Sieb des Eratosthenes

```java
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;
    }
  }
```

```
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:

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

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

```
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 }
```

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)

```java
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

```java
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 Prim (Minimum Spanning Tree)

```c
#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];
      }
    }
```

```
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 }
```

3.3 Maximaler Fluss (Ford-Fulkerson)

```
1  #include <stdio.h>
2  #include <limits.h>
3
4  #define n_MAX 36
5  #define m_MAX 30
6  #define SIZE (m+6+2)
7  #define SIZE_MAX 38
8  #define QUELLE (m+6)
9  #define SENKE (m+7)
10 #define NONE -1
11 #define INF INT_MAX/2
12
13 int n, m;
14 int capacity[SIZE_MAX][SIZE_MAX];
15 int flow[SIZE_MAX][SIZE_MAX];
16 int queue[SIZE_MAX], *head, *tail;
17 int state[SIZE_MAX];
18 int pred[SIZE_MAX];
19
20 enum { XS, S, M, L, XL, XXL };
21 enum { UNVISITED, WAITING, PROCESSED };
22
23 int strToOffset( char *str);
24 int maxFlow( int quelle, int senke);
25
26 int main(){
27
28     int numOfProps;
29     scanf("%d\n", &numOfProps);
30
31     while( numOfProps--){
32         scanf("%d %d\n", &n, &m);
33
34         int i, j;
35
36         /* Matrix initialisieren */
37         for( i=0; i< SIZE; i++){
38             for( j=0; j< SIZE; j++){
39                 capacity[i][j] = flow[i][j] = 0;
40
```

```c
41                     if( i == QUELLE && j < m){
42                         capacity[i][j] = 1;
43                         continue;
44                     }
45
46                     if( j == SENKE && i >= m && i < QUELLE){
47                         capacity[i][j] = n/6;
48                         continue;
49                     }
50                 }
51             }
52
53         char str[4];
54
55         /* Matrix einlesen */
56         for( i=0; i< m; i++){
57             scanf("%s", str);
58             capacity[i][m+strToOffset(str)] = 1;
59             scanf("%s", str);
60             capacity[i][m+strToOffset(str)] = 1;
61         }
62
63
64         int foo = maxFlow( QUELLE, SENKE);
65         printf("%s\n", foo >= m ? "YES" : "NO");
66
67     }
68
69     return 0;
70 }
71
72 int strToOffset( char *str){
73     /*snip*/
74 }
75
76 void enqueue( int x){
77     *tail++ = x;
78     state[x] = WAITING;
79 }
80
81 int dequeue(){
82     int x = *head++;
83     state[x] = PROCESSED;
84     return x;
85 }
86
87 int bfs( int start, int target){
88     int u, v;
89     for( u=0; u< SIZE; u++){
90         state[u] = UNVISITED;
91     }
92     head = tail = queue;
93     pred[start] = NONE;
94
95     enqueue(start);
96
97     while( head < tail){
98         u = dequeue();
99
100         for( v= 0; v< SIZE; v++){
101             if( state[v] == UNVISITED &&
102                 capacity[u][v] - flow[u][v] > 0){
103
104                 enqueue(v);
105                 pred[v] = u;
106             }
107         }
108     }
109
110     return state[target] == PROCESSED;
111 }
112
113 int maxFlow( int quelle, int senke){
114     int max_flow = 0;
115
116     int u;
```

```
117
118     while( bfs( quelle, senke)){
119         int increment = INF, temp;
120
121         for( u= senke; pred[u] != NONE; u = pred[u]){
122             temp = capacity[pred[u]][u] - flow[pred[u]][u];
123             if( temp < increment){
124                 increment = temp;
125             }
126         }
127
128         for( u= senke; pred[u] != NONE; u = pred[u]){
129             flow[pred[u]][u] += increment;
130             flow[u][pred[u]] -= increment;
131         }
132
133         max_flow += increment;
134     }
135
136     return max_flow;
137 }
```

```
1   /**
2    * Ford Fulkersen
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
```

4 Geometrische Algorithmen

4.1 Graham Scan (Convex Hull)

```
1 static List<P> graham(List<P> l) {
2   if (l.size() < 3)
3     return l;
4   P temp = l.get(0);
5   for (P p : l)
6     if (temp.y > p.y || temp.y == p.y && temp.x > p.x)
7       temp = p;
8   final P start = temp; // min y (then leftmost)
9
10  Collections.sort(l, new Comparator<P>() {
11    public int compare(P o1, P o2) {
12      if (new Double(Math.atan2(o1.y - start.y, o1.x - start.x)) // same angle
13          .compareTo(Math.atan2(o2.y - start.y, o2.x - start.x)) == 0)
14        return new Double(Math.sqrt((o1.x - start.x)
15            * (o1.x - start.x) + (o1.y - start.y)
16            * (o1.y - start.y))).compareTo((o2.x - start.x)
17            * (o2.x - start.x) + (o2.y - start.y)
18            * (o2.y - start.y)); // use distance
19      return new Double(Math.atan2(o1.y - start.y, o1.x - start.x))
20          .compareTo(Math.atan2(o2.y - start.y, o2.x - start.x));
21    }
22  });
23  Stack<P> s = new Stack<P>();
24  s.add(start);
```

```
25    s.add(l.get(1));
26    for (int i = 2; i < l.size(); i++) {
27      while (s.size() >= 2
28          && ccw(s.get(s.size() - 2), s.get(s.size() - 1), l.get(i)) <= 0)
29        s.pop();
30      s.push(l.get(i));
31    }
32    return s;
33  }
34
35  // turn is counter-clockwise if > 0; collinear if = 0; clockwise else
36  static double ccw(P p1, P p2, P p3) {
37    return (p2.x - p1.x) * (p3.y - p1.y) - (p2.y - p1.y) * (p3.x - p1.x);
38  }
39
40  public static class P {
41    double x, y;
42
43    P(double x, double y) {
44      this.x = x;
45      this.y = y;
46    }
47    // polar coordinates (not used)
48    // double r() { return Math.sqrt(x * x + y * y); }
49    // double d() { return Math.atan2(y, x); }
50  }
```

4.2 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()
```

```java
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

```c
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

```c
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 fnc(type`$_1$`,...)`	function declarations
	external variable declarations
`type name`	
`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"`

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', '\xhh'`
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.

Flow of Control

statement terminator	`;`
block delimiters	`{ }`
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)`

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*",*arg1*,...)
print to string s	sprintf(s,"*format*",*arg1*,...)
read formatted data	scanf("*format*",&*name1*,...)
read from string s	sscanf(s,"*format*",&*name1*,...)
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*",*arg1*,...)
read from file	fscanf(*fp*,"*format*",*arg1*,...)
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: "%-+ 0w.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 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	abs(n)
absolute value of long n	labs(n)
quotient and remainder of ints n,d	div(n,d)
return structure with div_t.quot and div_t.rem	
quotient and remainder of longs 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()
time2−time1 in seconds (double)	difftime(time2,time1)
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	$\operatorname{asin}(x)$, $\operatorname{acos}(x)$, $\operatorname{atan}(x)$
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	