Team Contest Reference
getRandomNumber(){return 4;}

Universität zu Lübeck

23. November 2012

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;
    }
  }
```

```
  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;
}
```

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 Prim (Minimum Spanning Tree)

```
#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;
}
```

3.3 Maximaler Fluss (Ford-Fulkerson)

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

#define n_MAX 36
#define m_MAX 30
#define SIZE (m+6+2)
#define SIZE_MAX 38
#define QUELLE (m+6)
#define SENKE (m+7)
#define NONE -1
#define INF INT_MAX/2

int n, m;
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 { XS, S, M, L, XL, XXL };
enum { UNVISITED, WAITING, PROCESSED };

int strToOffset( char *str);
int maxFlow( int quelle, int senke);

int main(){

    int numOfProps;
    scanf("%d\n", &numOfProps);

    while( numOfProps--){
        scanf("%d␣%d\n", &n, &m);

        int i, j;

        /* Matrix initialisieren */
        for( i=0; i< SIZE; i++){
            for( j=0; j< SIZE; j++){
                capacity[i][j] = flow[i][j] = 0;

```

```
                if( i == QUELLE && j < m){
                    capacity[i][j] = 1;
                    continue;
                }

                if( j == SENKE && i >= m && i < QUELLE){
                    capacity[i][j] = n/6;
                    continue;
                }
            }
        }

        char str[4];

        /* Matrix einlesen */
        for( i=0; i< m; i++){
            scanf("%s", str);
            capacity[i][m+strToOffset(str)] = 1;
            scanf("%s", str);
            capacity[i][m+strToOffset(str)] = 1;
        }


        int foo = maxFlow( QUELLE, SENKE);
        printf("%s\n", foo >= m ? "YES" : "NO");

    }

    return 0;
}

int strToOffset( char *str){
    /*snip*/
}

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

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

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 quelle, int senke){
    int max_flow = 0;

    int u;
```

```

    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;
}
```

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

```
 Funktion Dijkstra(Graph, Startknoten):
     initialisiere(Graph,Startknoten,abstand[],vorgaenger[],Q)
     solange Q nicht leer:                       // Der eigentliche Algorithmus
         u := Knoten in Q mit kleinstem Wert in abstand[]
         entferne u aus Q                                // fuer u ist der kuerzeste Weg nun bestimmt
         fuer jeden Nachbarn v von u:
             falls v in Q:
                distanz_update(u,v,abstand[],vorgaenger[])   // pruefe Abstand vom Startknoten zu v
     return vorgaenger[]

 Methode initialisiere(Graph,Startknoten,abstand[],vorgaenger[],Q):
     fuer jeden Knoten v in Graph:
         abstand[v] := unendlich
         vorgaenger[v] := null
     abstand[Startknoten] := 0
     Q := Die Menge aller Knoten in Graph

 Methode distanz_update(u,v,abstand[],vorgaenger[]):
     alternativ := abstand[u] + abstand_zwischen(u, v)   // Weglaenge vom Startknoten nach v ueber u
     falls alternativ < abstand[v]:
         abstand[v] := alternativ
         vorgaenger[v] := u
```

4 Geometrische Algorithmen

4.1 Graham Scan (Convex Hull)

```
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(Math.sqrt((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;
}

// 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);
}

public static class P {
  double x, y;

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

4.2 Punkt in Polygon

```
/**
 * -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*`"`

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.

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)`

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)`
returns 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⟩