Worked solutions to selected problems from the ACM International Collegiate Programming Contest (ICPC)

Colin Dooley, Stiofáin Fordham, Colm Ó Dúnlaing[*]
School of Mathematics, Trinity College, Dublin 2, Ireland

April 19, 2013

Abstract

This document presents solutions to 18 contest problems from regional competitions and from the world-finals of the ACM ICPC.

[*]E-mail: `dooleyct@maths.tcd.ie`, `odunlain@maths.tcd.ie`, `stevef@maths.tcd.ie`. Mathematics departmental website: `http://www.maths.tcd.ie.`

Contents

1 1989 Europe Northwestern, problem A

Problem statement: http://livearchive.onlinejudge.org/external/55/5566.pdf

Take (say) base 10, multiplier 4 and least-significant digit 7. Suppose a number $a_n a_{n-1} \ldots a_1 7$ satisfies the `rotamultproperty` then $a_1 = 8$ since $7 \times 4 = 28$ and $a_2 = 4$ since $4 \times 8 + 2 = 34$ (where the 2 is included since it carries over from the 28) and so on.

This is the algorithm we use, continue generating $a_i, a_{i+1}, \ldots$ until we find an a_i such that a_i is equal to the given least-significant digit, and the given equation (`rotamultproperty`) is satisfied.

1.1 Globals

- `lsd` is the least-significant digit read in from the file.
- `base` and `factor` are the base and first-factor respectively, that have been read-in from the input file.
- `count` keeps track of how long the number currently is (number of digits).
- `right-dig` is the value of the a_i being examined at that stage.
- `carry` is the value of the carry-over at that stage of the algorithm.

⟨*5566.globals*⟩≡
```
int base,lsd,factor;
int carry=0,container,count=0,right_dig;
div_t divis;
FILE *fin;
```

1.2 Main code

⟨*5566*⟩≡
```
#include <stdio.h>
#include <stdlib.h>

#define FILENAME "input.txt"

main()
{
        ⟨5566.globals⟩

        if( (fin=fopen(FILENAME, "r"))==NULL) exit(1);

        while(fscanf(fin,"%d %d %d",&base,&lsd,&factor)==3)
        {
                right_dig=lsd;
                do
                {
                        count++;
                        container=(right_dig*factor)+carry;

                        divis=div(container,base);
                        carry=divis.quot;
                        right_dig=divis.rem;
                } while(right_dig!=lsd || carry!=0);

                printf("%d\n",count);
                carry=right_dig=count=container=0;
        }
        fclose(fin);
        exit(0);
}
```

1.3 Makefile

Memorandum

```
notangle -t8 -R5566.Makefile report.nw > makefile
```

⟨*5566.Makefile*⟩≡

```
5566: report.nw
        /usr/bin/notangle -L -R5566 report.nw > 5566.c
        gcc -o 5566 5566.c

clean:
        rm *.c *.h *.tex *.aux *.log *.out

.SUFFIXES: .nw .tex .c

.nw.c: ;        /usr/bin/notangle $*.nw > $*.c
```

2 1993 World finals, problem A

Problem statement: http://livearchive.onlinejudge.org/external/51/5161.pdf

We construct a weighted digraph that includes a vertex for the originating city, the destination city and each of the stations, with a view to using Dijkstra's algorithm. Denote by $d(v)$ the distance from the originating city to the station v. The edges are directed such that if (u, v) is an edge with $v \neq u$ then $d(v) - d(u)$ is positive. This construction also ensures that the digraph has a single-source - the original city. However, there are further conditions that influence whether or not an edge is included:

- A driver never stops at a gasoline station when the gasoline tank contains more than half of its capacity unless the car cannot get to the following station (if there is one) or the destination with the amount of gasoline in the tank.

Thus, we must delete an edge (u, v) from the graph if $d(v) - d(u)$ is less than half the product of the efficiency of the car engine and the capacity of the tank - subject to the above stated condition. Furthermore, we must delete an edge (u, v) if $d(v) - d(u)$ is greater than the product of the efficiency of the car engine and the capacity of the tank since in this case the car would run out of gas before reaching the station.

Deleting edges may introduce new sources, so we must also include a routine to delete a vertex if it is a source and if it is not the originating city.

2.1 Globals

1. The integer `n` is the number of stations on the road.
2. `set` is the number of the current data set.
3. `tot-dist` is the distance from the source to the origin.
4. `cost` is the cost of petrol at the source.
5. The matrix `dist` contains station distances - `dist[i][j]` is the distance from station `i` to `j`.
6. The `price` array contains the fuel cost at each station.

⟨*5161.globals*⟩≡
```
int n;
int i,j,k,set=1;
double tot_dist,capacity,mpg,cost;
double** dist;
double* price;
FILE *fp;
```

2.2 Input & output

⟨*5161*⟩≡
```
#include <stdio.h>
#include <stdlib.h>

#define FILENAME "input.txt"

⟨5161.find the minimum cost⟩

int main()
{
  ⟨5161.globals⟩
```

```
    if((fp=fopen(FILENAME,"r"))==NULL) exit(1);

    fscanf(fp, "%lf", &tot_dist);
    while(tot_dist!=-1.0) {
      fscanf(fp, "%lf %lf %lf %d", &capacity, &mpg, &cost, &n);
      dist=malloc((n+2)*sizeof(double));

      for(i=0;i<n+2;i++) {
        dist[i]=malloc((n+2)*sizeof(double));
      }

      price=malloc((n+2)*sizeof(double));
```

The `dist` array is first initialised to 0.0.

⟨*5161*⟩+≡

```
      for(i=0;i<n+2;i++) {
        for(j=0;j<n+2;j++) {
          dist[i][j]=0.0;
        }
      }

      price[0] = cost/100.0;
      price[n+1] = 0.0;

      for(j=1;j<n+1;j++) {
        fscanf(fp, "%lf %lf",&dist[0][j], &price[j]);
      }

      dist[0][n+1] = tot_dist;

      for(k=0;k<n+2;k++) {
        for(j=k+1;j<n+2;j++) {
          dist[k][j] = dist[0][j]-dist[0][k];
        }
      }

      printf("Data set #%d\n", set);
      printf("   Minimum cost is $%.2lf\n\n",
                  find_min_cost(n, capacity, mpg, cost, price, dist));

      fscanf(fp, "%lf", &tot_dist);
      set++;
    }
    fclose(fp);

    return 0;
  }
```

2.3 Finding the minimum cost

The `weight` matrix contains the weights associated with each edge and the `w` array contains the weights associated with the vertices at any stage of Dijkstra's algorithm.

⟨*5161.find the minimum cost*⟩≡
```
  double find_min_cost(int n, double capacity,
          double mpg, double cost, double* price, double** dist) {
    int i,j,k,s,l=0,chk=n+1, examine=1;
    double min;
    double temp;
    double** weight;
    double* w;

    w=malloc((n+2)*sizeof(double));
    weight=malloc((n+2)*sizeof(double));

    for(i=0; i<n+2; i++) {
      weight[i]=malloc((n+2)*sizeof(double));
    }
```

The `T` is the array containing the tentative elements, if `T[i]=1` then `i` is tentative.

⟨*5161.find the minimum cost*⟩+≡
```
    double* T;
    T=malloc((n+2)*sizeof(double));

    ⟨5161.adjacency matrix construction⟩

    ⟨5161.source deletion⟩

    ⟨5161.dijkstra's algorithm⟩

    return (cost + w[n+1]/100.0);
  }
```

2.4 Constructing the graph

The graph is stored as an adjacency matrix with all entries initialised to -1.0. We weight the edges as

$$\text{weight}_{kj} = p_j \times \frac{d_{kj}}{\text{mpg}}$$

where p_j is the price at station j and d_{kj} is the distance from station k to station j and mpg is the efficiency of the car engine.

Finally, if (u, v) is an edge, where v is the destination city, then we set the weight of that edge to 0, because once we leave station u and reach the destination we are done.

⟨*5161.adjacency matrix construction*⟩≡
```
  for(k=0;k<n+2;k++) {
    for(j=0;j<n+2;j++) {
      weight[j][k] = -1.0;
    }
  }

  for(k=0;k<n+1;k++) {
    for(j=k+1;j<n+1;j++) {
      if(dist[k][j] <= capacity*mpg && (dist[k][j] >= 0.5*capacity*mpg
                  || dist[k][j+1]>capacity*mpg)) {
        weight[k][j] = price[j]*(dist[k][j]/mpg)+200.0;
      }
    }
  }

  for(k=0;k<n+1;k++) {
    if(dist[k][j] <= capacity*mpg) {
      weight[k][n+1] = 0.0;
    }
  }
```

2.5 Source deletion routine

Since some of the edges are deleted, new sources may be created, but we require the originating city to be the only source so that we can apply Dijkstra's algorithm. Only one pass through the graph is necessary to delete the sources. The method used is by examining a column of the adjacency matrix: if a node i of the graph is a source, then every entry in column i will be -1.0. The method checks if the sum of the entries in the column is equal to the product of -1.0 and the number of vertices in the graph.

⟨*5161.source deletion*⟩≡
```
  for(j=1;j<n+2;j++) {
    temp=0.0;

    for(i=0;i<n+2;i++) temp+=weight[i][j];

    if(temp==-(n+2)) {
      for(k=j+1;k<n+2;k++) weight[j][k]=-1.0;
    }
  }
```

2.6 Dijkstra's algorithm

The program uses a standard implementation of Dijkstra's algorithm.

⟨*5161.dijkstra's algorithm*⟩≡
```
  for(k=0;k<n+2;k++) T[k]=1.0;
  for(i=0; i<n+2; i++) w[i] = 100000.0;

  w[0] = 0.0;

  for(s=0;s<n+2;s++) {
    min=1000001;

    for(i=0;i<n+2;i++) {
      if(T[i]==1.0 && w[i]<min) {
        min=w[i];
        l=i;
      }
    }

    T[l]=0.0;

    for(j=l+1;j<n+2;j++) {
      if(weight[l][j]!=-1.0) {
        double x=w[l]+weight[l][j];
        if(x<w[j]) {
          w[j]=x;
        }
      }
    }
  }
```

2.7 Debugging history

The first instance of the program did not include the source deletion routine, as we hadn't realised that new sources could be created. After re-examining our algorithm, this routine was added.

It is worth noting that the Universidad de Valladolid online judge at http://livearchive.onlinejudge.org/ returns a "Wrong answer" result for this program. We have been unable to identify any other problems with this program, and consequently, leave it to the ambitious reader to try to resolve.

2.8 Makefile

Memorandum

```
notangle -t8 -R5161.Makefile report.nw > makefile
```

⟨*5161.Makefile*⟩≡

```
5161: report.nw
        /usr/bin/notangle -L -R5161 report.nw > 5161.c
        gcc -o 5161 5161.c

clean:
        rm *.c *.h *.tex *.aux *.log *.out

.SUFFIXES: .nw .tex .c

.nw.c: ;        /usr/bin/notangle $*.nw > $*.c
```

3 1993 World finals, problem B

Problem statement: http://livearchive.onlinejudge.org/external/51/5162.pdf

We categorise the edges as follows: "border" edges are edges where both vertices lie on the border, and "plot" edges are edges that are not border edges. We move in a clockwise direction around the border edges of the rectangle, and at each vertex we try to make a "left-turn", that is, we select the (next) vertex that is closest to a left-turn (see diagram below). This will allow us to move around a plot, and once we have moved back to the original border vertex, we then proceed along the border to the next point and repeat the process etc. So the problem amounts to constructing two functions `find-left-turn` to locate the leftmost edge and `follow-border` to locate the next border vertex given some border vertex.

3.1 Structures & globals

Edges, vertices and points are stored as structures. Each vertex contains an array `conns` of vertices that the vertex is connected to, and also an array `slopes` such that `slopes[i]` is the angle of the edge connecting that vertex and `conns[i]`. The `border-traversed` is necessary to ensure that the program moves in the correct direction around the rectangle. If `border-traversed[i]=1` then the edge from `vertex` to `conns[i]` has been visited by the border-traversal function. The `Edge` structures are used only for the reading in stage of the program, and for storing them into the `conns` elements of the vertices.

⟨*5162.structures*⟩≡
```
typedef struct {
  int x;
  int y;
} point;

typedef struct vertex {
  int label;
  point loc;

  int num_conns;
  point *conns;
  double *slopes;
  int *border_traversed;
} Vertex;

typedef struct edge {
  int code;
  point vert1;
  point vert2;
} Edge;
```

The `vertices` array contains the vertices of the graph and vice versa `edges` the edges. The `pos` integer keeps track of the location of left-most empty position in the `vertices` array and `set` is the current data set.

⟨*5162.globals*⟩≡
```
int num,pos=0,set=1;
Vertex *vertices;
Edge *edges;
int i,j,k;
point temp1,temp2;
FILE *inp;
```

```
int x1temp;
int y1temp;
int x2temp;
int y2temp;
```

3.2 Input & output

⟨*5162.main code*⟩≡

```
int main() {
  ⟨5162.globals⟩

  if((inp=fopen(FILENAME,"r"))==NULL) exit(1);

  fscanf(inp,"%d",&num);

  while(num!=0) {
    printf("Data set #%d\n",set);

    vertices=malloc( num*sizeof(Vertex) );
    edges=malloc( num*sizeof(Edge) );

    for(i=0;i<num;i++) vertices[i].loc=make_point(-1,-1);
    pos=0;

    /* Read-in data and fill the edges and vertices arrays */
    for(i=0;i<num;i++) {
      fscanf(inp,"%d %d %d %d",&x1temp,&y1temp,&x2temp,&y2temp);
      temp1=make_point(x1temp,y1temp);
      temp2=make_point(x2temp,y2temp);

      if(not_in_list(vertices,temp1,pos)) {
        vertices[pos].loc=make_point(x1temp,y1temp);
        vertices[pos].label=pos;
        vertices[pos].num_conns=0;
        vertices[pos].conns=malloc(MAX * sizeof(point));

        for(j=0;j<MAX;j++) {
          vertices[pos].conns[j].x=-1;
          vertices[pos].conns[j].y=-1;
        }

        vertices[pos].slopes=malloc(MAX * sizeof(double));
        vertices[pos].border_traversed=malloc(MAX * sizeof(double));

        for(j=0;j<MAX;j++) vertices[pos].slopes[j]=PI/2;
        pos++;
      }

      if(not_in_list(vertices,temp2,pos)) {
        vertices[pos].loc=make_point(x2temp,y2temp);
        vertices[pos].label=pos;
        vertices[pos].num_conns=0;
        vertices[pos].conns=malloc(MAX * sizeof(point));
        for(j=0;j<MAX;j++) {
          vertices[pos].conns[j].x=-1;
          vertices[pos].conns[j].y=-1;
        }

        vertices[pos].slopes=malloc(MAX * sizeof(double));
        vertices[pos].border_traversed=malloc(MAX * sizeof(double));
        for(j=0;j<MAX;j++) vertices[pos].slopes[j]= PI/2;
        pos++;
      }

      edges[i]=make_edge(x1temp,y1temp,x2temp,y2temp);
    }

    /* Fill the conns & slopes elements */
    for(i=0;i<num;i++) {
      for(j=0;j<num;j++) {
        if(edges[i].vert1.x==vertices[j].loc.x
               && edges[i].vert1.y==vertices[j].loc.y) {
          vertices[j].conns[vertices[j].num_conns]=
               make_point(edges[i].vert2.x,edges[i].vert2.y);
          vertices[j].slopes[vertices[j].num_conns]=
               get_slope(vertices[j].loc,edges[i].vert2);
          vertices[j].border_traversed[vertices[j].num_conns]=0;
          (vertices[j].num_conns)++;
        }

        if(edges[i].vert2.x==vertices[j].loc.x
               && edges[i].vert2.y==vertices[j].loc.y) {
          vertices[j].conns[vertices[j].num_conns]=
               make_point(edges[i].vert1.x,edges[i].vert1.y);
          vertices[j].slopes[vertices[j].num_conns]=
               get_slope(vertices[j].loc,edges[i].vert1);
          vertices[j].border_traversed[vertices[j].num_conns]=0;
          (vertices[j].num_conns)++;
        }
      }
    }

    ⟨5162.count the plot edges⟩

    fscanf(inp,"%d",&num);

    free(vertices);
    free(edges);
    set++;
  }

  fclose(inp);
  return 0;
}
```

3.3 Ancillary functions

The `make-point` and `make-edge` functions are, hopefully, self-explanatory.

⟨*5162.construct structures*⟩≡

```
point make_point(int x, int y) {
  point temp;
  temp.x = x;
  temp.y = y;
  return temp;
}

Edge make_edge(int x1, int y1, int x2, int y2) {
  Edge temp;
  temp.vert1.x=x1;
  temp.vert1.y=y1;
  temp.vert2.x=x2;
  temp.vert2.y=y2;
  return temp;
}
```

The `not-in-list` function is needed for filling the `vertices` array to ensure that a vertex is not added to that array twice. Given some point, the `get-label` function checks whether it is in the `list` array and if it is it returns the label tag, otherwise it returns -1.

⟨*5162.member checking and label retrieval*⟩≡

```
int not_in_list(Vertex *list, point a, int max) {
    int i;
    for(i=0;(list[i].loc.x!=a.x || list[i].loc.y!=a.y) && i<max;i++);

    if(max==0 || max==i) return 1;
    return 0;
}

int get_label(Vertex *list, point a,int max) {
    int i;
    for(i=0;i<max;i++) if(list[i].loc.x==a.x
        && list[i].loc.y==a.y) return list[i].label;
    return -1;
}
```

3.4 Assigning angles to edges

For each vertex and each edge connected to that vertex we first assign an angle $\theta \in [0, 2\pi)$ relative to the $+x$-axis.

⟨*5162.get angle*⟩≡

```
double get_slope(point a,point b) {
  int x1=a.x,y1=a.y,x2=b.x,y2=b.y;

  if(x2-x1 != 0) {
    /* First quadrant */
    if(x2-x1>0 && y2-y1>0) return atan( ((y2-y1) / (x2-x1)) );
    /* Second quadrant */
    else if(x2-x1<0 && y2-y1>0) return PI-atan(abs( ((y2-y1) / (x2-x1)) ));
    /* Third quadrant */
    else if(x2-x1<0 && y2-y1<0) return PI+atan(abs( ((y2-y1) / (x2-x1)) ));
```

```
      /* Fourth quadrant */
      else if(x2-x1>0 && y2-y1<0) return (2*PI)-atan(abs( ((y2-y1) / (x2-x1)) ));

      /* Case where it lies on x-axis */
      else if(x2-x1>0) return 0.0;
      else return PI;
    }

    /* Case where it lies on y-axis */
    else if(y2-y1 > 0) return PI/2;
    else return 3*(PI/2);
  }
```

3.5 Turning left

We generalise the idea of turning left[1]- it most easily explained using a diagram.

!!Put the diagram here!!

In the above diagram moving to vertex A is "turning left" - we choose the out-edge where the angle shown is the largest for that edge.

The function works by mapping all edges attached to the vertex so that the in-edge has an associated angle of π and rotates the others appropriately. It then simply compares the angles associated to each of the other out-edges.

⟨*5162.turn left*⟩≡

```
  int find_left_turn(Vertex *list, int vertex,
          int invertex, double inslope, int pos) {
    int maxlabel=-1,i;
    double max=0.0,adj_angle;

    for(i=0;i<list[vertex].num_conns;i++) {
      if(inslope>PI) {
        adj_angle=list[vertex].slopes[i]-(inslope-PI);
        if(adj_angle<0) adj_angle=(2*PI+adj_angle);

        if( adj_angle>=max
          && invertex!=get_label(list,list[vertex].conns[i],pos) ) {
          maxlabel=get_label(list,list[vertex].conns[i],pos);
          max=adj_angle;
        }
      }

      else if(inslope<PI) {
        adj_angle=list[vertex].slopes[i]+(PI-inslope);
        if( adj_angle>=(2*PI) ) adj_angle=(adj_angle-(2*PI));

        if( adj_angle>=max
          && invertex!=get_label(list,list[vertex].conns[i],pos) ) {
          maxlabel=get_label(list,list[vertex].conns[i],pos);
          max=adj_angle;
        }
      }
```

[1]We were later informed that the correct term for this process is "determining the cycle predecessor of an edge" - that is, the out-edge returned by this function will be the cyclic predecessor of the in-edge.

```
    else {
      if(list[vertex].slopes[i]>=max
        && invertex!=get_label(list,list[vertex].conns[i],pos) ) {
        maxlabel=get_label(list,list[vertex].conns[i],pos);
        max=list[vertex].slopes[i];
      }
    }
  }
  return maxlabel;
}
```

3.6 Traversing the border

We move along the rectangle's border in an anticlockwise direction. To achieve this, the program tries to move to the right, if it cannot then it moves upwards, and so on - the priority is go right, otherwise go up, otherwise go left, otherwise go down. The `prior` integer is used for this purpose. The `border-traversed` label is checked in each case, and once the next border vertex has been located, its label is turned on to mark it as visited.

⟨*5162.follow border*⟩≡

```
int follow_border(Vertex *list,int vertex,int pos) {
  int i,j;
  int label;
  int prior=0;
  double min=2*PI;

  for(i=0;i<list[vertex].num_conns && prior!=4;i++) {
    if(list[vertex].slopes[i]==0.0
        && list[vertex].border_traversed[i]!=1){
      label = get_label(list,list[vertex].conns[i],pos);
      prior=3;
    }
    else if(list[vertex].slopes[i]==PI/2
        && list[vertex].border_traversed[i]!=1 && prior!=3) {
      label = get_label(list,list[vertex].conns[i],pos);
      prior=2;
    }
    else if(list[vertex].slopes[i]==PI
        && list[vertex].border_traversed[i]!=1 && prior!=2) {
      label = get_label(list,list[vertex].conns[i],pos);
      prior=1;
    }
    else if(list[vertex].slopes[i]==(3*PI/2)
        && list[vertex].border_traversed[i]!=1 && prior!=1) {
      label = get_label(list,list[vertex].conns[i],pos);
    }
  }

  for(i=0;i<list[label].num_conns;i++) {
    if(list[label].conns[i].x==list[vertex].loc.x
        && list[label].conns[i].y==list[vertex].loc.y) {
      list[label].border_traversed[i]=1;
    }
  }
  return label;
}
```

3.7 Counting the number of edges on the plots

The program functions as follows

1. Locate the bottom left corner of the bordering rectangle
2. Begin traversing the bordering rectangle in an anticlockwise direction
3. As the program moves along the border, at each vertex check it we can "move inside" the rectangle - this will be true if the vertex returned by the `follow-border` function is not the same as the vertex returned by the `find-left-turn` function.
4. When inside the plot, move around counting the edges (surveyor edges, that is) until the next vertex returned by the `find-left-turn` function is the border vertex that we moved from to first enter the current plot.
5. Resume traversing border, repeating the above, until we return to the bottom left corner.

⟨*5162.count the plot edges*⟩≡

```
/* Find vertex nearest the origin */
int xmin=500;
int ymin=500;

for(i=0;i<num;i++) {
  if( (vertices[i].loc.x<xmin || vertices[i].loc.y<ymin) && vertices[i].loc.x!=-1) {
    xmin=vertices[i].loc.x;
    ymin=vertices[i].loc.y;
  }
}

int min=get_label(vertices,make_point(xmin,ymin),pos);
int nxt_border=follow_border(vertices,min,pos);
int previous_node=min,next_move,num_of_edges;
int container;

while(nxt_border!=min) {
  num_of_edges=0;
  next_move=find_left_turn(vertices,nxt_border,previous_node,
        get_slope(vertices[previous_node].loc, vertices[nxt_border].loc), pos);

  if(next_move!=follow_border(vertices,nxt_border,pos)) {
    previous_node=nxt_border;

    while(next_move!=nxt_border) {
      container=next_move;
      next_move=find_left_turn(vertices,next_move,previous_node,
        get_slope(vertices[previous_node].loc,vertices[next_move].loc),pos);
      previous_node=container;
      num_of_edges++;
    }
  }
  if(num_of_edges!=0) printf("Number of edges in this plot: %d\n",num_of_edges+1);
  previous_node=nxt_border;
  nxt_border=follow_border(vertices,nxt_border,pos);
}
```

3.8 Consolidated

⟨*5162*⟩≡
```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define FILENAME "input.txt"
#define MAX 10
#define PI 3.141596
```

⟨*5162.structures*⟩

⟨*5162.construct structures*⟩

⟨*5162.member checking and label retrieval*⟩

⟨*5162.get angle*⟩

⟨*5162.turn left*⟩

⟨*5162.follow border*⟩

⟨*5162.main code*⟩

3.9 Debugging history

The principle difficulty with this problem was the construction of the function to determine the “turn left” vertex; the function in its present form is the third version (the function became so complicated that it was easier just to rewrite it). Otherwise, there were no other major problems.

3.10 Makefile

Memorandum

```
notangle -t8 -R5162.Makefile report.nw > makefile
```

⟨*5162.Makefile*⟩≡
```
5162: report.nw
        /usr/bin/notangle -L -R5162 report.nw > 5162.c
        gcc -o 5162 5162.c

clean:
        rm *.c *.h *.tex *.aux *.log *.out

.SUFFIXES: .nw .tex .c

.nw.c: ;        /usr/bin/notangle $*.nw > $*.c
```

4 1993 World finals, problem C

Problem statement: http://livearchive.onlinejudge.org/external/51/5163.pdf

In this problem simple relationships, such as *aaaaa bbbbb* d, are inputted into the program, e.g. Bruce Thomas 1, meaning Bruce is Thomas's son. These relationships should be recorded so as to be able to give the relationship between any two people when asked. These relationships are ouputted as *aaaaa bbbbb are cousin-*m-n which is read as aaaaa is bbbbb's mth cousin $n$ce removed. For example Kal Kara are cousin-1-0 means Kal is Kara's first cousin. If two people are siblings then $m = n = 0$.

4.1 Solution

Firstly, a node structure was created, as can be seen below, the struct members included an initial capacity for ancestors and descendants, which were needed since a node could possibly have many parent nodes and child nodes. *anc_count* and *desc_count*, keeps track of the number of parent links and child links. The index of the node was used so as to identify it easily in an array.

Each node had an array of weights. A weight for each parent link. This denoted the generation difference between a node and it's parent.

⟨*5163.NODE struct*⟩≡
```
typedef struct le_node {
    int anc_capacity;
    int desc_capacity;
    int desc_count;
    int anc_count;
    int index;
    int *weight;
    struct le_node **anc, **desc;
    char *name;
} NODE;
```

The following code initialises a node. The *make_node* function with the name argument creates a node with that name and sets up its ancestor, descendant and weight arrays, with an initial capacity of five ancestors and descendants.

The subsequent function frees the node.

⟨*5163.makenode and freenode*⟩≡
```
NODE *make_node(char *name){
    int i;
    NODE *node;
    node = calloc(1, sizeof(NODE));
    node->name = malloc(6*sizeof(char));
    for(i=0;i<6;i++) node->name[i]=name[i];

    NODE **desc;
    desc = calloc(5, sizeof(NODE));
    node->desc=desc;

    NODE **anc;
    anc = calloc(5, sizeof(NODE));
    node->anc = anc;

    node->desc_count = 0;
    node->anc_count=0;
    node->anc_capacity=5;
    node->desc_capacity=5;

    node->weight=calloc(node->anc_capacity,sizeof(int));
    for(i=0;i<node->anc_capacity;i++) node->weight[i]=0;

    return node;
}

void free_node(NODE *p) {
    int i;
    free(p->name);
    for(i=0;i<p->desc_count;i++) {
        free(p->desc[i]);
    }
    free(p->desc);

    for(i=0;i<p->anc_count;i++) {
        free(p->anc[i]);
    }
    free(p->anc);
    free(p->weight);

    free(p);
}
```

The code below is implemented if a node requires more room for ancestors or descendants. It increases the capacity by one each time. Any more seemed like overkill. The *realloc()* function is used since the existing ancestors of that node need to be kept. This seemed like the most efficient way of doing that.

⟨*5163.update capacities*⟩≡
```
void update_anc_capacity(NODE *p) {
    NODE **anc;
    int *weight;
    anc = realloc(anc,p->anc_capacity+1);
    p->anc = anc;
    ++p->anc_capacity;
    weight=realloc(weight,p->anc_capacity);
    p->weight=weight;
}

void update_desc_capacity(NODE *p) {
    NODE **desc;
    desc = calloc(p->desc_capacity+1, sizeof(NODE));
    p->desc = desc;
    ++p->desc_capacity;
}
```

This function is called once a new node and relationship is given in the input. It assigns to each node the appropriate ancestor or descendant and the corresponding weight. It also calls the update capacities functions if required.

⟨*5163.miscellaneous node sorting*⟩≡
```
void sort_out_node(NODE *p, NODE *q, int rel) {
    if(q->desc_count>=q->desc_capacity) update_desc_capacity(q);
    else {
        q->desc[q->desc_count] = calloc(1, sizeof(NODE));
        q->desc[q->desc_count] = p;
        ++q->desc_count;
    }
    if(p->anc_count>=p->anc_capacity) update_anc_capacity(p);
    else {
        p->anc[p->anc_count] = calloc(1, sizeof(NODE));
        p->anc[p->anc_count] = q;
        ++p->anc_count;
    }
    p->weight[p->anc_count-1]=rel;
}
```

The following is a recursive function which creates an array the same size as the number of nodes read in. It assigns a distance from the node *p*, to each ancestor of *p* in the element corresponding to the index of the ancestor of *p*. the corresponding element for a node which is not an ancestor of *p*, is left as -1.

⟨*5163.collect ancestors*⟩≡
```
void collect_anc(NODE *p, int *marked, int index) {
    int i;
    int v[index];
    for(i=0;i<index;i++) v[i]=-1;
    if(p->anc_count == 0) return;
    if(p==NULL || v[p->index]>=0) return;
    else {
        for(i=0;i<p->anc_count;i++) {
            marked[p->anc[i]->index] = p->weight[i] + marked[p->index];
            ++v[p->index];
            collect_anc(p->anc[i], marked, index);
        }
    }
}
```

This function uses the arrays collected above and searches for an index in both such that the element at that index is non negative. It sums the values at these elements and searches for the least of these. If such an element exists, m and n, as defined above, are calculated. If not, *LCA* remains as -1, allowing for the program to return *aaaaa and bbbbb are not related*.

⟨*5163.LCA*⟩≡
```
void find_LCA(NODE *p, NODE *q, int index, int *m, int *n, int *LCA) {
    int least = 1000001, i;
    int *pmarked = malloc(index*sizeof(int));
    int *qmarked = malloc(index*sizeof(int));
    for(i=0;i<index;i++) pmarked[i] = qmarked[i]=-1;

    *LCA=-1;
    pmarked[p->index]=0;
    qmarked[q->index]=0;

    collect_anc(p, pmarked, index);
    collect_anc(q, qmarked, index);

    for(i=0; i< index; i++) {
        if(pmarked[i]>=0 && qmarked[i]>=0 && (pmarked[i]+qmarked[i] < least)) {
            least =pmarked[i]+qmarked[i];
            *LCA = i;
        }
    }
```

```
      if((*LCA)>-1) {
          if(pmarked[(*LCA)]<=qmarked[(*LCA)]) {
              *m=pmarked[(*LCA)] -1;
              *n=qmarked[(*LCA)] - (*m)- 1;
          }
          else {
              *m=qmarked[(*LCA)] -1;
              *n=pmarked[(*LCA)] - (*m)- 1;
          }
      }
      free(pmarked);
      free(qmarked);
  }
```

The main function while tying all the above functions together also reads in the input. It does this using the switch function. If the first character of the line is ”R”, the program knows to record a relationship. If it reads ”F”, it knows to return a relationship where one exists. A ”#” implies a comment, and ”E” ends the program.

The main function also matches names read in to nodes that may exist already. This is done where ”F” and ”R” are read. Each node read in is stored in an array of nodes. The search function searches through this linearly using the *strcmp* function defined in *string.h*. In the ”R” case, if no match is found, a node is created. If no match is found in the ”F” case, the output *aaaaa and bbbbb are not related* is given.

⟨*5163.main code*⟩≡

```
  int main() {
      char c;
      char *name1, *name2;
      char buffer[100];
      int rel, index=0, i, j,quit1, quit2, node_count=0;
      int n1, n2, LCA, m,n;

      NODE **nodes;
      nodes=calloc(10000, sizeof(NODE));

      name1=malloc(6*sizeof(char));
      name2=malloc(6*sizeof(char));

      NODE *node1;
      node1=calloc(1, sizeof(NODE));
      NODE *node2;
      node2=calloc(1, sizeof(NODE));

      while((c=getchar())!='E') {
          switch(c) {
              case 'R':
                  if(scanf("%5s %5s %d", name1, name2, &rel)!=3) {
                      exit(1);
                  }
                  j=quit1 =quit2=0;
                  while(j<index && (quit1<1 || quit2<1)) {
                      if(strcmp(name1, nodes[j]->name)==0) {
                          *node1 = *(nodes[j]);
                          n1=j;
                          ++quit1;
                      }
```

```
            if(strcmp(name2, nodes[j]->name)==0) {
                *node2 = *(nodes[j]);
                n2=j;
                ++quit2;
            }
            ++j;
        }

        if(quit1==0) {
            node1 = make_node(name1);
            nodes[index] = calloc(1, sizeof(NODE));
            *nodes[index] =*node1;
            nodes[index]->index = index;
            n1=index;
            ++index;
        }

        if(quit2==0) {
            node2 =make_node(name2);
            nodes[index] = calloc(1, sizeof(NODE));
            *nodes[index] =*node2;
            nodes[index]->index = index;
            n2=index;
            ++index;
        }

        sort_out_node(nodes[n1], nodes[n2], rel);

        fgets(buffer, 100, stdin);

        break;

    case 'F':

        if(scanf("%5s %5s", name1, name2)!=2) {
            exit(1);
        }
        j=quit1 =quit2=0;
        while(j<index && (quit1<1 || quit2<1)) {
            if(strcmp(name1, nodes[j]->name)==0) {
                *node1 = *(nodes[j]);
                ++quit1;
            }
            if(strcmp(name2, nodes[j]->name)==0) {
                *node2 = *(nodes[j]);
                ++quit2;
            }
            ++j;
        }

        if(quit1+quit2==2) {
            find_LCA(node1, node2, index, &m ,&n, &LCA);
            if(LCA==-1)
                printf("%s and %s are not related.\n", node1->name, node2->nam
                        else if(LCA == node2->index)
                            printf("%s and %s are descendant-%d\n", node1->name, node2->na
                        else if(LCA == node1->index)
                            printf("%s and %s are descendant-%d\n", node1->name, node2->na
                        else {
                            printf("%s and %s are cousin-%d-%d\n", node1->name, node2->nam
                        }
                    }
                    else printf("%s and %s are not related.\n", name1, name2);

                    fgets(buffer, 100, stdin);
                    break;
                case '#':
                    fgets(buffer, 100, stdin);
                    break;
            }
        }

        for(j=0;j<node_count;j++) free_node(nodes[j]);
        free(nodes);
        free_node(node1);
        free_node(node2);
        free(name1);
        free(name2);
        return 0;
    }
```

4.2 Gathered Code

⟨*5163*⟩≡

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

⟨*5163.NODE struct*⟩

⟨*5163.makenode and freenode*⟩

⟨*5163.update capacities*⟩

⟨*5163.miscellaneous node sorting*⟩

⟨*5163.collect ancestors*⟩

⟨*5163.LCA*⟩

⟨*5163.main code*⟩

4.3 History

This problem is tricky in that as relationships are read in, the family tree begins to look less and less like a tree. The cause of this lies mainly in the fact that each node can have any number of parents and children. An example where this happens is where *child father 1* and *child grandfather 2*, may be read in. Thus child has two parents, and could possibly have more. This problem was circumvented by allowing each node up to five parents and children initially, a function was then made to increase these capacities if required.

A variation of this code was submitted to the online judge, the difference between it and the present code was that an attempt was made to calculate generations of each node as they were read in. A verdict of wrong answer was returned. It was found that the method got out of hand very quickly. For example, if the following situation were to arise: Given two distinct trees T_1 and T_2, if a new relationship is read in, say n_1 at depth d_1 in T_1 is a parent node to n_2 at depth d_2 in T_2, $0 < d_1$, d_2. Re-building all depths in the resulting family tree would be a difficult and expensive task. Therefore it was decided to give every node an array that stored the distance between it and any parent node. Distances between nodes could then be calculated once a query is read in.

The above code, while returning the correct answer to the sample input, when submitted to the online judge, a verdict of wrong answer is returned.

Creating a pure graph might be a better way to solve this problem. At least it might be more efficient. There is some information on least common ancestor problems about, however it seems they can only be applied where a tree structure exists and so would need to be adapted in some way.

4.4 Makefile

Memorandum

```
notangle -t8 -R5163.Makefile report.nw > Makefile
```

⟨*5163.Makefile*⟩≡

```
5163: report.nw
	/usr/bin/notangle -L -R5163 report.nw > 5163.c
	gcc -o 5163 -lm 5163.c

clean:
	rm *.c *.h *.tex *.aux *.log *.out

.SUFFIXES: .nw .tex .c

.nw.c: ;	/usr/bin/notangle 5163.nw > 5163.c
```

5 1995 Central Europe, problem A

Problem statement: http://livearchive.onlinejudge.org/external/55/5515.pdf

5.1 Problem

This is a simple problem to solve. Once the month names and day names are stored in an array, and can be matched to the input, this is is just a modular arithmetic problem.

5.2 Solution

The solution is straightforward. The *strcmp* function is used to match the input with the stored month and day names. Then it is a simple process to find the total number days passed since the Mayan calendar began, first multiplying the number of years given by the number of days in a year, the index of the name of month by the number of days in a month and add the days given.

Knowing the days makes it simple translate to the other calendar. The number of the day, n, can be found easily by calculating

$$n \cong d\ (mod 13),$$

where d is the total number of days. The index of the day name can similarly be found by calculating

$$i \cong d\ (mod 20).$$

5.3 Code

⟨*5515*⟩≡
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
  int j, a=0, n;
  int day, year, tot_days;
  int tzname, tznumber, tzyear;
  char month[7];

  char *tzolkin[]= {"imix", "ik", "akbal", "kan",
                "chicchan", "cimi", "manik",
        "lamat", "muluk", "ok", "chuen", "eb", "ben", "ix", "mem",
         "cib", "caban", "eznab", "canac", "ahau"};
  char *haab[]= {"pop", "no", "zip", "zotz", "tzec", "xul", "yoxkin",
        "mol", "chen", "yax", "zac", "ceh", "mac",
                "kankin", "muan", "pax", "koyab", "cumhu", "uayet"};

  if(scanf("%d",&n)!=1) {
    exit(1);}

    printf("%d\n", n);
    while(a<n) {
      if(scanf("%d.%s%d", &day, month, &year)!=3){
      exit(1);
    }

    j=0;
    while(strcmp(haab[j], month)!=0 && j<19) {
      ++j;
    }
    if(j==19) {
      exit(1);
    }

    tot_days = day + j*20 + year*365;
    tzyear = (int)(tot_days/260.0);
    tznumber = tot_days%13;
    tzname = tot_days%20;

    printf("%d %s %d\n", tznumber+1, tzolkin[tzname], tzyear);

    ++a;
    }

  return 0;
}
```

5.4 Bug History

No real bugs were found in this code, however, there was difficulty when submitting to the online judge. A verdict of Runtime Error was given. The error turned out to be because of the method of output. Initially, answers were outputted once all the input had been read in. This angered the judge, which prompted the change to output after each input is read in.

5.5 Makefile

Memorandum

```
notangle -t8 -R5515.Makefile report.nw > 5515.Makefile
```

⟨*5515.Makefile*⟩≡

```
  5515: 5515.nw
      /usr/bin/notangle -L -R5515 report.nw > 5515.c
      gcc -o 5515 -lm 5515.c

  clean:
      rm *.c *.h *.tex *.aux *.log *.out

  .SUFFIXES: .nw .tex .c

  .nw.c: ;    /usr/bin/notangle 5515.nw > 5515.c
```

6 1995 Europe central, problem F

Problem statement: http://livearchive.onlinejudge.org/external/55/5520.pdf

Brute force method is the most obvious way to solve this problem, and it appears to be sufficiently quick.
The `div` variable keeps track of the divisor and the `res` variable keeps track of the residue calculated at each stage of the algorithm.

⟨*5520*⟩≡
```
#include <stdio.h>
#include <stdlib.h>

main()
{
    int i,k,div,res;
    printf("Enter value for k: ");
    if(scanf("%d",&k)!=1) exit(1);
    div=2*k;

    while(k!=0) {
        for(i=k;div!=k;i++) {
            div=2*k;
            res=0;
            do {
                res+=(i%div);
                res=(res%div);

                if(res==0) res=div;
                 if(res>k) div--;

                res--;
            } while(res+1>k && div!=k);
        }

        printf("%d\nEnter value for k: ",i-1);
        if(scanf("%d",&k)!=1) exit(1);
    }
    exit(0);
}
```

6.1 Makefile

Memorandum

```
notangle -t8 -R5520.Makefile report.nw > makefile
```

⟨*5520.Makefile*⟩≡
```
5520: report.nw
        /usr/bin/notangle -L -R5520 report.nw > 5520.c
        gcc -o 5520 5520.c

clean:
        rm *.c *.h *.tex *.aux *.log *.out

.SUFFIXES: .nw .tex .c

.nw.c: ;        /usr/bin/notangle $*.nw > $*.c
```

7 1996 Europe northwestern, problem H

Problem statement: http://livearchive.onlinejudge.org/external/55/5585.pdf

7.1 Overall

Gaston runs the mailroom…it boils down to maintaining a binary tree with sizes (descendant counts) so that

$$\text{size(left subtree)} \geq \text{size(right subtree)} \geq \text{size(left subtree)} - 1$$

7.2 Globals

First, we set up the data structures. Then the input routines.

⟨*5585*⟩≡
```
#include <stdio.h>
#include <stdlib.h>

typedef struct tree_node_tag
{
  int size;
  struct tree_node_tag *left, *right, *parent;
  int letter;
} TREE_NODE;

typedef struct
{
  TREE_NODE * root;
} TREE;

TREE_NODE * make_node ( int letter )
{
  TREE_NODE * node = calloc(1, sizeof(TREE_NODE));
  node -> letter  = letter;
  node -> size = 1;
  return node;
}

void free_node ( TREE_NODE * node )
{
  free ( node );
}
```

7.3 Input

Input trees are given in preorder. General form of input is

⟨*problem count*⟩

⟨*tree*⟩⟨*new letter count*⟩⟨*letter*⟩ … ⟨*letter*⟩

⟨*tree*⟩⟨*etcetera*⟩

An individual tree is presented as

$$\langle\textit{root}\rangle\langle\textit{left subtree}\rangle\langle\textit{right subtree}\rangle$$

⟨*5585*⟩+≡

```
TREE_NODE * read_tree ()
{
  int x;
  TREE_NODE * node;

  scanf ( "%d", &x );
  if ( x == 0 )
    return NULL;
  else
  {
    node = make_node ( x );
    node->left = read_tree ();
    node->right = read_tree ();
    return node;
  }
}
```

7.4 Output

The code below is a luxury, borrowed from elsewhere. It doesn't work as intended,[2] but is sufficient for debugging.

⟨*5585*⟩+≡

```
void rec_display (int depth, int marked[], TREE_NODE * p, int with_sizes )
{
  int i, on_left, on_right;

  if (p!=NULL)
  {
    on_left = (p -> parent != NULL && p == p->parent->left);
    on_right = (p -> parent != NULL && p == p->parent->right);

    marked[depth] = on_right;

    rec_display ( depth+1, marked, p->left, with_sizes );
    for (i=0; i<depth; ++i)
      if ( marked[i] )
        printf("| ");
      else
        printf("  ");

    if ( with_sizes )
      printf("+%d size %d\n",p->letter,p->size);
```

[2]Probably because the parent links have not been set.

```
      else
        printf("+%d\n",p->letter);

      marked[depth] = on_left;

      rec_display ( depth+1, marked, p->right, with_sizes );
    }
  }

  void display_tree ( TREE * tree, int with_sizes )
  {
    int marked[100];
    int i;

    for (i=0; i<100; ++i )
      marked[i] = 0;

    printf("\ndisplay tree\n");
    rec_display (0, marked, tree -> root, with_sizes );
    printf("\n");
  }
```

7.5 Balanced insert

This is the hard part. It begins with an unvarnished insertion which adjusts the sizes along the search path. Unlike red-black rebalancing, for instance, rebalancing is done top down: always following the most recent insertion path. Therefore rebalancing will follow the same path to the newly-inserted letter, but rebalancing will occur at the highest node. The rules are

- Rebalancing begins at the highest node p whose children fail the balancing requirement.
- The letter at the node will be replaced by its inorder successor or predecessor depending on which subtree is too heavy. Then the displaced letter will be re-inserted in the other subtree. In other words, the letter at the node gets 'bumped.'

$\langle 5585 \rangle{+}{\equiv}$

```
          /*
           * size_up computes the size of subtree
           * at p, in postorder using recursiong
           */

  void size_up ( TREE_NODE * p )
  {
    int r;
    if ( p != NULL )
    {
      r = 1;
      if ( p->left != NULL )
      {
        size_up ( p->left );
        r += p->left->size;
      }
      if ( p->right != NULL )
      {
        size_up ( p->right );
        r += p->right->size;
      }
      p -> size = r;
    }
  }

          /*
           * Insert makes no size adjustments: too messy.
           */

  void insert ( TREE * tree, int letter )
  {
    TREE_NODE * p, * q, * new;

    if ( tree -> root == NULL )
    {
      tree -> root = make_node ( letter );
      return;
    }

    p = tree->root;
    q = NULL;
    while ( p != NULL )
    {
      q = p;

      if ( p -> letter == letter )
      {
        printf("ERROR: inserting letter %d already there, abort\n", letter);
        display_tree ( tree, 1 );

        exit(-1);
      }

      if ( letter < p->letter )
        p = p->left;
      else
        p = p->right;
    }

    new = make_node ( letter );
    if ( letter < q->letter )
      q -> left = new;
    else
      q -> right = new;
  }

          /*
           * succ --- assumes p has right child
           */

  int delete_succ ( TREE_NODE * p )
```

```
{
  TREE_NODE * parent = p;
  TREE_NODE * q = p->right;
  int letter;

  while ( q->left != NULL )
  {
    parent = q;
    q = q->left;
  }

  letter = q->letter;
  if ( parent == p )
    p->right = NULL;
  else
    parent->left = NULL;

  free_node (q);

  return letter;
}

int delete_pred ( TREE_NODE * p )
{
  TREE_NODE * parent = p;
  TREE_NODE * q = p->left;
  int letter;

  while ( q->right != NULL )
  {
    parent = q;
    q = q->right;
  }

  letter = q->letter;
  if ( parent == p )
    p->left = NULL;
  else
    parent->right = NULL;

  free_node (q);

  return letter;
}

        /*
         * unbalanced_descendant () searches among descendants
         * of node for the highest unbalanced node, if any.
         * Warning: tree sizes must be correct.
         */

TREE_NODE * unbalanced_descendant ( TREE_NODE * node )
{
  TREE_NODE * p;
  int left, right;

  if ( node == NULL )
    return NULL;

  left  = node->left ==NULL? 0:node->left ->size;
  right = node->right==NULL? 0:node->right->size;

  if ( right > left || left > right + 1 )
    return node;

  p = unbalanced_descendant ( node->left );

  if ( p == NULL )
    p = unbalanced_descendant ( node->right );

  return p;
}

        /*
         * Rebalance is recursive.
         * It returns the total
         * number of recursions.
         */

int rebalance ( TREE * tree, int letter )
{
  TREE_NODE * p, * q;
  int saved_letter, other;
  int left, right;
  int count;

/* debugging */
printf("rebalance, letter %d, in tree..\n", letter);
display_tree( tree, 0 );

  size_up ( tree->root );

  p = unbalanced_descendant ( tree -> root );
  if ( p == NULL )
{
/* debugging */
printf("rebalancing %d, nothing to do\n", letter);
    return 0;
}

  saved_letter = p->letter;
  left  = p->left ==NULL? 0:p->left ->size;
  right = p->right==NULL? 0:p->right->size;
  if ( left > right + 1 )
    other = delete_pred ( p );
  else
    other = delete_succ ( p );
  p->letter = other;
```

```
    insert( tree, saved_letter );

    count = rebalance ( tree, saved_letter );
    return count + 1;
  }
```

7.6 Main

⟨*5585*⟩+≡

```
  main()
  {
    int prob_count, i, num_additional,j, letter, count;

    TREE * tree = calloc(1, sizeof(TREE));

    scanf("%d", & prob_count );

    for (i=0; i<prob_count; ++i )
    {
      count = 0;
      tree -> root = read_tree ();
      size_up ( tree->root );
      display_tree ( tree, 0 );
      scanf("%d", & num_additional );
      for (j=0; j<num_additional; ++j)
      {
        scanf( "%d", &letter );
        insert ( tree, letter );
        count += rebalance ( tree, letter );
      }
      display_tree ( tree, 1 );
      printf("Problem %d total of %d rebalancing operations\n",
          i, count);
    }
  }
```

7.7 Makefile

Memorandum

```
notangle -t8 -R5585.Makefile report.nw > makefile
```

⟨*5585.Makefile*⟩≡

```

  5585: report.nw
          /usr/bin/notangle -L -R5585 report.nw > 5585.c
          gcc -o 5585 5585.c

  clean:
          rm *.c *.h *.tex *.aux *.log *.out
```

```
.SUFFIXES: .nw .tex .c

.nw.c: ;        /usr/bin/notangle $*.nw > $*.c
```

7.8 Debugging log

Took maybe 6 hours to reach testing level. It would have been wise to begin testing when the i/o had been written, but it seems to work anyway. The contest requires the number of rebalancing operations to be counted. Get the code working before deciding what and how to count.

```
data file nw96h.in
2
3 2 0 0
5 0 0
4
1 6 7
9
400 250 0 0 0
2
511 100
```

7.9 Conclusions

A log of revisions is given in earlier versions. This version works (except that it has verbose output) on the data as expected.

- Total about 6 or 7 hours to write and about 3 hours to debug. Maybe another 30 minutes to titivate (such as adding this sentence.)
- Tree display seemed useful.
- One has to fight the impulse to do things in linear time.
- Recursion makes life much easier.

8 1999 Europe central, problem D

Problem statement: http://livearchive.onlinejudge.org/external/55/5564.pdf

The objective is to determine if there is a subset of the marbles whose value is equal to half the value of the entire set. Suppose we are given $\{1, 0, 1, 2, 0, 0\}$ as input, for which there is no division into two equal lots. Our algorithm works as follows: write the set as $\{4, 4, 3, 1\}$, if we start at the leftmost 4 and move right, then we have the subcollection $\{4, 4\}$, but clearly this is useless, since we require a subset that sums to $\frac{12}{2} = 6$. So, we try the second 4 and obtain $\{4, 3\}$ but $7 > 6$, so we try the 3 but at this point we should end our search since we can have a subset of maximum value $3 + 1 = 4$. So, there is no division. We implement this as follows: first we produce an array `val` of the marbles from largest to smallest viz. $\{4, 4, 3, 1\}$ and an array `r-tot` such that `r-tot[i]` is the sum of all elements `val[j]` where $j \geq i$ - in this case it would be $\{12, 8, 4, 1\}$. We call a recursive function (`scan()`) on the first entry of this array that moves from left to right, and recursively keeps callling the function again and again subject to

- Another recursive call occurs at position `i` if the value of `r-tot[i]` plus the current computed total is less than half the total value of the marbles.
- Another recurisve call occurs at position `i` if it will result in a new current total that is still less than half the total value of the marbles.

⟨*5564.scan function*⟩≡
```
void scan(int pos, int ctot) {
        int i;
        for(i=pos;pos<num && (ctot+r_tot[i])>=tot/2 && chk!=1;i++) {
                if(ctot+val[i]==tot/2) chk=1;
                else if(ctot+val[i]<tot/2) scan(i+1,ctot+val[i]);
        }
}
```

Another trivial observation is that, if the total marble value is odd then there is no subdivision, since the marbles have integral values.

8.1 Globals

- `tot` is the total value of all marbles.
- `num` is the number of marbles.
- `val[]` is an array of size `num-1` containing the marble values ordered from largest to smallest.
- `r-tot` is an array of size `num-1` whose structure has been described previously.
- `chk` is a flag used to tell the program that a sub-division has been found.
- `ctot` is the current calculated total.
- `pos` is the current position in that instance of the `scanf()` function.

⟨*5564.globals*⟩≡
```
int val[M_NUM],num,r_tot[M_NUM],tot,chk;
```

8.2 Building the arrays

The data is read in from the file in the opposite manner from which we require so we need the `hold[]` array to manipulate it to our required form.

⟨*5564.loading data*⟩≡
```
void load_data(FILE *fin) {
        int i,j,k,hold[M_NUM],set=1;
        num=0,tot=0;

        for(i=0;i<M_VAL;i++) {
                if(fscanf(fin,"%d",&hold[i])!=1) exit(1);
                num+=hold[i];
                tot+=(i+1)*hold[i];
        }

        k=0;
        for(i=0;i<M_VAL;i++)
                for(j=0;j<hold[M_VAL-i-1];j++) {
                        val[k]=M_VAL-i;
                        k++;
                }

        for(i=1;i<=num;i++) r_tot[num-i]=r_tot[num-i+1]+val[num-i];
}
```

8.3 Consolidated code

⟨*5564*⟩≡
```
#include <stdlib.h>
#include <stdio.h>

#define FILENAME "input.txt"
#define M_VAL 6
#define M_NUM 20000

⟨5564.globals⟩

⟨5564.loading data⟩

⟨5564.scan function⟩

main() {
        int set=1;
        FILE *fin;

        if( (fin=fopen(FILENAME,"r"))==NULL ) exit(1);
        load_data(fin);

        while(tot!=0) {
                printf("Collection #%d:\n",set);
                chk=0;
                scan(0,0);
                chk? printf("Can be divided.\n"):printf("Can't be divided.\n");
```

```
                        set++;
                        load_data(fin);
                        if(tot!=0) printf("\n");
                }
                exit(0);
        }
```

8.4 Debugging history

Debugging was necessary on the `scan` function. Initially a `return` statement was used to signal a positive result, but this yielded incorrect behaviour because the `return` statement was nested inside the recursive calls, therefore as the call ended it eventually reached the outer loop and executed the `return 0` statement. This issue was solved by including the `chk` flag inside the `scanf` function.

8.5 Makefile

Memorandum

```
notangle -t8 -R5564.Makefile report.nw > makefile
```

⟨*5564.Makefile*⟩≡

```
5564: report.nw
        /usr/bin/notangle -L -R5564 report.nw > 5564.c
        gcc -o 5564 5564.c

clean:
        rm *.c *.h *.tex *.aux *.log *.out

.SUFFIXES: .nw .tex .c

.nw.c: ;        /usr/bin/notangle $*.nw > $*.c
```

9 2002 Europe northwestern, problem C

Problem statement: http://livearchive.onlinejudge.org/external/26/2672.pdf

To solve the problem we construct the line through each point with the appropriate angle and then solve the simultaneous equations for x and y. Suppose line one has equation $y = m(\alpha_1)x+c_1$ and line two has equation $y = m(\alpha_2)x+c_2$. The angle is measured from the the north so the slope is given by

$$m(\alpha_i) = \tan\left(\frac{(90-\alpha_i)\pi}{180}\right)$$

and constant terms are determined from the given points - say we are given as input points (x_1, y_1) and (x_2, y_2) then the constants satisfy

$$c_1 = y_1 - m(\alpha_1)x_1 \tag{9.1}$$
$$c_2 = y_2 - m(\alpha_2)x_2 \tag{9.2}$$

The edge cases are where either of the angles are multiples of 90 degrees or equal to zero, these are dealt with later. We must solve $m(\alpha_1)x + c_1 = m(\alpha_2)x + c_2$ whence

$$x = \frac{c_2 - c_1}{m(\alpha_1) - m(\alpha_2)} = \frac{(y_2 - m(\alpha_2)x_2) - (y_1 - m(\alpha_1)x_1)}{m(\alpha_1) - m(\alpha_2)}$$

⟨*2672.slope and xpoint*⟩≡
```
double m(int a) {
  return tan((90-a)*PI/180);
}

double xpoint(int a1,int b1,int ang1,int a2,int b2,int ang2) {
  return ( (b2-m(ang2)*a2)-(b1-m(ang1)*a1) )/(m(ang1)-m(ang2));
}
```

Suppose α_1 is a multiple of 90 degrees or equal to zero, but α_2 is not, then it is a matter of determining where the equation of line two intersects a horizontal or vertical line. So we substitute into the other equation the value x_1 or y_1 as appropriate and vice versa in α_2's case etc.

The case where both angles satisfy this condition is easy to solve (see code below).

9.1 Variables

- `a1` and `b1` is the first point (x_1, y_1) read in and vice versa `a2` and `b1` with (x_2, y_2).
- `ang1` and `ang2` are the respective angles.
- `cases` are the number of cases to be read in from the file.

⟨*2672.variables*⟩≡
```
int i,cases,a1,b1,a2,b2,ang1,ang2;
double x,y;
FILE *fin;
```

9.2 Consolidated code

⟨*2672*⟩≡
```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#define FILENAME "input.txt"
#define PI 3.1415926

⟨2672.slope and xpoint⟩

main() {
  ⟨2672.variables⟩

  if( (fin=fopen(FILENAME,"r"))==NULL ) exit(1);

  fscanf(fin,"%d",&cases);

  for(i=0;i<cases;i++) {
    fscanf(fin,"%d %d %d",&a1,&b1,&ang1);
    fscanf(fin,"%d %d %d",&a2,&b2,&ang2);

    if( (ang2%90!=0 && ang2!=0) && (ang1%90!=0 && ang1!=0) ) {
      printf("%0.4f ",x=xpoint(a1,b1,ang1,a2,b2,ang2));
      printf("%0.4f\n",(double)(m(ang1)*(x-a1)+b1));
    }

    else if( (ang2==0 || ang2%90==0) && (ang1%90!=0 && ang1!=0) ) {
      if(ang2==90 || ang2==270) {
        printf("%0.4f %0.4f\n",(double)((b2-b1)/m(ang1)+a1),
                (double)b2);
      }
      else {
        printf("%0.4f %0.4f\n",(double)a2,
          (double)(m(ang1)*(a2-a1)+b1));
      }
    }

    else if( (ang2%90!=0 && ang2!=0) && (ang1%90==0 || ang1==0)) {
      if(ang1==90 || ang1==270) {
        printf("%0.4f %0.4f\n",(double)((b1-b2)/m(ang2)+a2),
          (double)b1);
      }
      else {
        printf("%0.4f %0.4f\n",(double)a1,
          (double)(m(ang2)*(a1-a2)+b2));
      }
    }

    else if((ang2%90==0 || ang2==0) && (ang1%90==0 || ang1==0)) {
      if(ang2==90 || ang2==270) {
        printf("%0.4f %0.4f\n",(double)a1,(double)b2);
      }
      else {
        printf("%0.4f %0.4f\n",(double)a2,(double)b1);
      }
    }
  }
  exit(0);
}
```

9.3 Debugging history

It is worth noting that the Universidad de Valladolid online judge at http://livearchive.onlinejudge.org/ returns a "Wrong answer" result for this program. We have been unable to identify any other problems with this program, and consequently, leave it to the ambitious reader to try to resolve.

9.4 Makefile

Memorandum

```
notangle -t8 -R2672.Makefile report.nw > makefile
```

⟨*2672.Makefile*⟩≡

```
2672: report.nw
        /usr/bin/notangle -L -R2672 report.nw > 2672.c
        gcc -o 2672 2672.c

clean:
        rm *.c *.h *.tex *.aux *.log *.out

.SUFFIXES: .nw .tex .c

.nw.c: ;        /usr/bin/notangle $*.nw > $*.c
```

10 2003 Europe northwestern, problem H

Problem statement: http://livearchive.onlinejudge.org/external/29/2938.pdf

A two player number game is given with the following rules. A number is written on a blackboard, each player takes turns either decomposing that number into two numbers if it is composite, decrementing by one if prime (and thus gaining a point), or if the number is one, removing and gaining a point. The game is over once there are no numbers left on the board.

Assuming each player adheres to a strategy of maximising there own points (i.e. if a point can be taken, that point is taken), the problem is therefore to predict the outcome of the game given the starting number.

10.1 Prime Number Check

Obviously, a function to check if a number is prime was required. This function after being given a number p, checks for all $2 \le k \le \sqrt{p}$, if $p(mod k) \equiv 0$. If not the case for any such k, p is prime. It then returns a 1 or 0 indicating whether or not p is prime or not respectively. If p is not prime it saves the divisor to k.

10.2 Main

The main part of the program essentially plays the game following the strategy outlined above. While keeping track of the sum of all numbers on the board, it checks first to see if there are any ones, then primes, and finally composite numbers. It keeps track of whose turn it is by assuming person A takes every even turn and person B every odd. Once $sum = 0$, the game is finished, and the program will print out the score.

10.3 Globals

1. *int* count keeps track of the number of the problem that is being solved.
2. *int* i, j, x are dummy variables.
3. *int* k stores the factor of a number being checked by the *prime_check* function.
4. *int* test is a dummy variable for a while statement. It checks for any change in sum after an iteration.
5. the *num* array stores the numbers that are written on the blackboard.
6. *int* last keeps track of the last position written to in the array.
7. *int* A and B are the scores of each player.

⟨*2938*⟩≡
```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int prime_check(int p, int *k) {
   (*k)=2;
   int prime=0, quit=0;

   while((*k)<=(int)(sqrt(p)) && quit!=1) {
      if((*k)>2 && (*k)%2==0) ++(*k);
      else if(p%(*k)==0) quit=1;
      else ++(*k);
   }
   if(((*k)==(int)(sqrt(p))+1 || p==2) && p!=1)
   prime = 1;

   return prime;
}

void update_go(int go, int count, int *A, int *B) {
   if(go%2 == 0) {
      ++A[count];
   }
   else {
      ++B[count];
   }
}

int main() {
   int *A, *B, prime, count=0, sum=0;
   int i,j,x=0,k1=1, k=1, prob, test=0, go=2;
   int num[25];
   int last=0;
   A=0;
   B=0;
   k=0;

   for(i=0;i<25;i++) num[i]=0;

   if(scanf("%d", &prob)!=1) {
      printf("didn't get that\n"); exit(1);}

      A=malloc(prob*sizeof(int));
      B=malloc(prob*sizeof(int));

      for(j=0;j<prob;j++) {
         A[j]=0;
         B[j]=0;
      }

      while(count<prob) {
         if(scanf("%d", &num[0])!=1) {
            exit(1);
         }

         sum=num[0];

         while(sum>0) {
            if(sum==1) {
               update_go(go, count, A, B);
               --sum;
            }
            else {
               for(j=0;j<20;j++) {
                  if(num[j]==1) {
                     --num[j];
                     update_go(go, count, A, B);
                     --sum;
                     ++go;
                  }
               }

               while(test!=sum) {
                  test=sum;
                  for(j=0;j<20;j++) {
                     if(prime_check(num[j], &k)==1) {
                        --num[j];
                        update_go(go, count, A, B);
                        --sum;
                        ++go;
                     }
                  }
               }
               test=0;

               for(j=0;j<20;j++) {
                  if(num[j]==1) {
                     --num[j];
                     update_go(go, count, A, B);
                     --sum;
                     ++go;
                  }
               }

               for(j=0;j<20;j++) x= num[j]>0? j:x;
               for(j=0;j<20;j++) x = (num[j] < num[x] && num[j])>0? j:x;

               if(prime_check(num[x], &k)==0 && num[x]!=0) {
                  sum-=num[x];
                  num[x]=num[x]/(double)k;

                  if(k!=2 && prime_check(num[x], &k1)==0) {
                     k*=k1;
                     num[x]=num[x]/(double)k1;
                  }

                  num[last+1] = k;
                  sum+=k+num[x];

                  ++last;
                  ++go;
               }
            }
         }
         x=0;
         last=0;
         go=2;
         ++count;
      }
      for(j=0;j<prob;j++) {
         printf("%d %d\n",A[j], B[j]);
```

```
        }

        free(A);
        free(B);

        exit(0);
}
```

10.4 History

There were some challenges in this code, for instance, how to store all the numbers on the board, and getting the while statements correct.

Firstly, the problem of storing the numbers was solved by making an array of size 20. The game only allowed starting numbers up to 1000000. Therefore by finding $\lceil log_2(1000000)\rceil = 20$, taking log to the base 2 since 2 is the least prime therefore maximum number of factors for a number less than 1000000, will not be more than 20. It was only needed to keep track of the last position written to in the array for inserting new elements.

While loops and for loops then needed to be widely used, as we needed to search through the array of factors for ones or primes on every turn. This was awkward when searching for primes, as the array needed to be searched through perhaps more than once, quitting only after all primes had been removed.i

The problem as a whole was straight forward, and required only simple algorithms to complete.

However, upon submission of this program to the online judge, a verdict of wrong answer was returned. While this program returns the correct output for the sample input, it does not seem to give the desired result for all numbers between 1 and 1000000.

It was discovered that strategies beyond those described in the problem statement play a part in the final score of a game. For instance, once all primes and ones are snapped up, the question of which composite number to take and how to decompose that number will make a difference. Different strategies were submitted to the online judge with no success. These included

- dividing the composite number by it's least prime factor,
- dividing by it's greatest prime factor,
- dividing into two more composite numbers if possible, and least primes after that.

The first strategy had the longest runtime before returning wrong answer indicating that it might be the closest to a correct answer.

10.5 Makefile

Memorandum

```
notangle -t8 -R2938.Makefile report.nw > report
```

⟨*2938.Makefile*⟩≡
```
2938: report.nw
	/usr/bin/notangle -L -R2938 report.nw > 2938.c
	gcc -o 2938 -lm 2938.c

clean:
	rm *.c *.h *.tex *.aux *.log *.out

.SUFFIXES: .nw .tex .c

.nw.c: ;	/usr/bin/notangle 2938.nw > 2938.c
```

11 2004 World finals, problem D

Problem statement: http://livearchive.onlinejudge.org/external/29/2996.pdf

This is a cryptography problem. The encryption method is given, the problem is to decrypt codewords where possible, returning ”Codeword not unique” if it is not posible.

11.1 Solution

A brute force method was used to solve this problem, however this resulted in a ”time limit exceeded” verdict when submitted to the online judge.

Firstly, a starting position and initial step size was set and an array was put together. Then a second starting poiny and step size was set and for each of these the maximum match was found.

Each starting position, stepsize and word length was tried, at the end of which, the largest word was returned. If this was unique an answer was obtained.

11.2 Code

⟨2996⟩≡
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void update_array(char array[], char tmp_c[],
                char c[], int s, int i, int m, int len) {
    int b, l=1,a=0,tmp_s=s;

    for(b=0;b<m;b++) {
        tmp_c[b]= c[b];
    }
    array[0]=tmp_c[s];
    tmp_c[s]='\0';
    while(l<len) {
        if(tmp_c[tmp_s]=='\0') {
            tmp_s = (tmp_s+1)%m;
        }
        else if(a<i) {
            tmp_s = (tmp_s+1)%m;
            ++a;
        }
        else {
            array[l] = tmp_c[tmp_s];
            tmp_c[tmp_s]  ='\0';
            ++l;
            a=0;
        }
    }
}

void update_t(char tmp_c[], int *tmp_t, int j, int m) {
    int a=0, quit=0;
    while(a<j && quit <m) {
        if(tmp_c[((*tmp_t)%m)]=='\0') {
            *tmp_t = ((*tmp_t)+1)%m;
            ++quit;
        }
        else {
            *tmp_t= ((*tmp_t) + 1)%m;
            ++a;
            quit=0;
        }
    }
    if(a==j)
        while(tmp_c[((*tmp_t)%m)]=='\0' && quit<m) {
            *tmp_t = ((*tmp_t)+1)%m;
            ++quit;
        }
}

void decode(char c[], char tmp_p[], int *chk) {
    int m = strlen(c);
    int i, j,len, s=0,t=0, k=0;
    int old_max=0;
    int max=0;
    int a;
    int count;

    int tmp_s, tmp_t;
    char tmp_c[m];

    for(len=(int)(m/2.0); len > max; len--) {
        char p[len];
        char array[len];

        for(a=0;a<len;a++) p[a]='\0';

        for(i=1;i<m;i++) {
            for(j=i+1;j<m;j++) {
                s=0;
                t=1;

                while(s<m) {
                    if(t==s && t<m)
                    ++t;
                    else if(t>=m) exit(1);

                    tmp_t = t;

                    update_array(array, tmp_c, c, s, i, m, len);

                    while(tmp_c[tmp_t]=='\0') {
                        tmp_t = (tmp_t+1)%m;
                    }

                    while(array[k] == tmp_c[tmp_t] && k<(len)) {
                        p[k] = array[k];
                        k++;

                        old_max=max;
                        *chk = max==k? 1:(*chk);
                        if(max<k) {
                            max=k;
                            *chk=0;
                        }

                        tmp_c[(tmp_t)] = '\0';

                        update_t(tmp_c, &tmp_t, j, m);
                    }

                    if(array[k] != tmp_c[tmp_t]) {
                        count=0;
                        if(old_max<k) {
                            for(a=0;a<len;a++) tmp_p[a] = p[a];
                        }
                        else if(old_max==k) {
                            for(a=0;a<len;a++)
                                if(tmp_p[a] == p[a]) {
                                    ++count;
                                }
                                if(count == len) (*chk)=0;
                        }
                        k=0;
                        ++t;
                        if(t==m) {
                            ++s;
                            t=0;
                        }
                        else if(t==m-1 && s==m-1) {
                            ++s;
                        }
                    }
                }
            }
        }
    }
}

int main() {
    int chk=0;
    char p[20], c[40];
    int count=1;

    if(scanf("%s", c)!=1) {
        printf("no\n");exit(1);
    }

    while(c[0]!='X' && c[1]!='\0') {
        printf("Code %d: ", count);
        ++count;
        decode(c, p, &chk);

        if(chk == 0) printf("%s\n", p);
        else printf("Codeword not unique\n");

        if(scanf("%s", c)!=1) {
            printf("no\n");exit(1);
        }
    }
    return 0;
}
```

11.3 Bug history

Initially, an attempt was made to match each character of both starting positions and characters after each step size. This however would not work because, when the code was being created, positions are written to before the second starting position is assigned. Thus these positions are skipped when writing in the second time. This was not taken into account with the original method. It was decided to set an array of varying length with the characters from the first starting postion and stepsize. Length started at half the size of the codeword and decreases until it is the size of the maximum match found.

11.4 Makefile

Memorandum

```
notangle -t8 -R2996.Makefile report.nw > Makefile
```

⟨*insecure.Makefile*⟩≡

```
2996: report.nw
    /usr/bin/notangle -L -R2996 report.nw > 2996.c
    gcc -o 2996 -lm 2996.c

clean:
    rm *.c *.h *.tex *.aux *.log *.out

.SUFFIXES: .nw .tex .c

.nw.c: ;    /usr/bin/notangle 2996.nw > 2996.c
```

12 2005 Europe northwestern, problem I

Problem statement: http://livearchive.onlinejudge.org/external/34/3416.pdf

There are N people carrying boxes up a tall building. Stairs are too narrow for two to pass. Anyone empty-handed is descending, full-handed ascending. Where an ascending meets a descending, they interchange states. Timing is such that they always meet halfway between floors.

…The state of the system can be presented as an 'up' and 'down' array, giving the number of people on each floor going up and down respectively, plus 'onbottom' plus 'ontop,' the numbers left on the bottom (not counting those instantaneously going up), and those on the top.

Write these as f (for the index of the penthouse floor), u_j, d_j, $0 \le j \le f$, b, t. Write m for the minute, so it indexes the 'steps' in the process

$$\begin{aligned}
u_f(m) = d_0(m) &= 0 \\
t(m+1) &= t(m) + u_{f-1}(m) \\
d_f(m+1) &= u_{f-1}(m) \\
u_{j+1}(m+1) &= u_j(m), \quad (0 \le j \le f-1) \\
d_j(m+1) &= d_{j+1}(m), \quad (0 \le j \le f-1) \\
u_0(m+1) &= \min(d_1(m), b(m)) \\
b(m+1) &= b(m) - u_0(m+1)
\end{aligned}$$

12.1 Solution code

⟨*3416*⟩≡

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

void next( int up[], int down[], int f, int * top, int * bottom ) {
    int i, j, init_down;
    init_down = down[1];
    (*bottom) -= up[0];
    (*top) += up[f-1];

    for (j=1; j<f; ++j) {
        down[j] = down[j+1];
    }

    for (j=f-1; j>=0; --j) up[j+1] = up[j];

    down[f] =up[f];
    up[f]= down[0] = 0;

    up[0] = init_down < (*bottom) ? init_down : (*bottom);
}

main() {
    int probcount, people, pent, boxes_remaining, moving_boxes=0;
    int atfloor, carrying;
    int up[1001], down[1001];
    int m,i,j, top, bottom;
    scanf("%d", &probcount );
    printf("%d problems...\n", probcount);

    for (i=0; i<probcount; ++i) {
        scanf("%d %d %d", &people, &pent, &boxes_remaining);

        for (j=0; j<=pent; ++j) up[j] = down[j] = 0;

        printf("problem %d;  %d people;  %d pent;  %d boxes_remaining\n",
            i, people, pent, boxes_remaining);
        for (j=0; j<people; ++j) {
            scanf("%d %d", &atfloor, &carrying);
            if ( carrying || atfloor==0 ){
                ++up[atfloor];
                if(atfloor!=0) {
                    ++moving_boxes;
                }
            }
            else ++down[atfloor];
        }

        top = 0;
        bottom = boxes_remaining;
        m = 0;
        printf(" remaining boxes = %d, moving boxes = %d\n", bottom, moving_boxes);

        while ( top != boxes_remaining + (moving_boxes)) {
            printf("\n");

            next (up, down, pent, & top, & bottom );
            ++m;
            printf("%d minutes with %d on bottom  and %d on top\n",
            m,bottom, top);
        }
    }
}
```

12.2 History

A few bugs were found while writing the code for this problem. Such problems as an infinite while loop and the updates on each floor not being evaluated correctly. It was found that these were caused by errors with for loop bounds, if and while statements and the ordering of update procedures. The following are the changes that were made:

- The while loop required a change to it's statement as it did not take into account the number of boxes currently on the stairs, leaving room for the program to quit once the number of boxes at the top of the stairs equalled the number at the bottom.
- The for loop for the u_j update was changed so as to update it from the top of the stairs down. This prevented the value of u_0 being copied to every u_j, $1 \geq j < F$.
- The bounds were changed for the for loops of the u_j and d_j updates so as u_j would begin at $F - 1$ and end at $f = 0$ and d_j would begin at $f = 1$ and end at $F - 1$.
- Lastly, the initial value of d_1 needed to be saved in the value *init_down* at the beginning of the *next* function, as it was needed for the update of u_0 at the end of the function.

After these changes were made the program ran smoothly, and gave the correct answer to both of the sample inputs given in the problem.

However, upon submitting this to the online judge, a verdict of ”time limit exceeded” was returned. The answers given by the program seem to be correct, at least as far as can be tested with pen and paper, therefore it’s only problem is inefficiency. It should be possible to write more efficient code, for instance if the program were to calculate in one step the length of time until it reaches the penthouse and positions of people thereafter, rather than calculating positions after every minute. Efficiency, however, is not very important as the actual competition does not look for the most efficient code.

12.3 Makefile

Memorandum

```
notangle -t8 -R3416.Makefile report.nw > Makefile
```

⟨*3416.Makefile*⟩≡

```
3416: report.nw
    /usr/bin/notangle -L -R3416 report.nw > 3416.c
    gcc -o 3416 -lm 3416.c

clean:
    rm *.c *.h *.tex *.aux *.log *.out

.SUFFIXES: .nw .tex .c

.nw.c: ;    /usr/bin/notangle 3416.nw > 3416.c
```

13 2008 Europe northwestern, problem G: solution I

Problem statement: http://livearchive.onlinejudge.org/external/42/4292.pdf

Given a number of matchsticks, it was asked to find out the largest and smallest number that could be created by laying the matchsticks out like a seven-segment display (i.e. like the numbers on an alarm clock). the numbers on an alarm clock.

13.1 Solution

The following table illustrates what numbers require what number of matches to make.

#	# Matches
1	2
7	3
4	4
2, 3, 5	5
0, 6, 9	6
8	7

The solution to the largest number was very straightforward, the goal being to make as many digits as possible using the matchsticks given. Thus, if an odd number of matches were given the solution was a 7 followed by as many 1's as possible. If the number was even, the solution was all 1's.

Finding the smallest number possible was more difficult. The goal was to make as little digits as possible, or put more exactly, making the smallest possible numbers that require the most matchsticks to make. Because of this, the list of possible numbers that could be used was narrowed to

$$0, 1, 2, 4, 6, 7, 8.$$

For example, there was no point in using five matchsticks to make a 5, when you could make a 2, etc. There were some exceptions however as leading zeroes are not allowed.

It was also discovered in the process of coding this problem, that the numbers 4 and 7 (taking 4 and 3 matchsticks respectively), were only used in a situation of having only 4 or 7 matchsticks in total.

The method used to solve this problem therefore was the following:

- Calculate number of digits to be used when creating smallest number. This is done by dividing number of matchsticks by 7 and rounding up,7 being the largest number of matchsticks that can be used for any one digit.
- Starting on the left hand side of the number, and starting with the number 1.
- Subtract 2 (the number of matchsticks used to make a 1) from the number of matches, then check if the following condition is satisfied,

  $$r \leq 7l,$$

  where r is the number of remaining matchsticks, and l is the number of digits left to be assigned.
- If satisfied, repeat process for next digit.
- Otherwise, increase the number, if at the first digit however increase to 2 say, otherwise, use 0.
- Repeat, until last digit, checking remainder at each iteration and making sure to have zero matchsticks remaining at the end.

13.2 Code

⟨*4292*⟩≡

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

void upper(int m, int *array) {
    int j;
    int num=(int)(m/2.0);

    for(j=0;j<num;j++) {
        array[j]=1;
    }

    if(m%2 == 1) {
        array[0]=7;
    }
}

void lower(int m, int *array) {
    int temp = 0, j, l=0, k=0, rem = m;
    int num=ceil(m/7.0);

    for(j=0;j<num;j++) {
        array[j]=0;
    }
    int poss[5]= {2,6,5, 6, 7};
    int value[5] = {1,0,2, 6, 8};

    while(l<num) {
        if((rem-poss[k])<=(num-l-1)*7 && rem-poss[k]>=0) {
            if(k==1 && l==0 && ((rem-poss[k+1])<=(num-l-1)*7)) {
                ++k;
                array[l]=value[k];
                ++l;
                rem-=poss[k];
                --k;
            }
            else {
                array[l]=value[k];
                ++l;
                rem-=poss[k];
            }
        }
        else ++k;
    }
    if(array[0]==0) array[0]=6;
    if(m==3) array[0]=7;
    if(m==4) array[0]=4;
}

int main() {
    int n, m, i;
    int a=0;

    int *array_max;
    int *array_min;
    if(scanf("%d", &n)!=1) {
        printf("not enough\n");exit(1);
    }

    int test[n];

    for(a=0;a<n;a++) {
        if(scanf("%d", &m)!=1) {
            printf("not enough\n");exit(1);
        }
        test[a] = m;
    }

    array_max = (int*)malloc(m*sizeof(int));
    array_min = (int*)malloc(m*sizeof(int));

    for(a=0;a<n;a++) {
        array_max = (int*)realloc(array_max, ((int)(test[a]/2.0))*sizeof(int));
        if(array_max==NULL) {
            printf("1:error with realloc.\n");
            exit(1);
        }

        array_min= (int*)realloc(array_min, (ceil(test[a]/7.0))*sizeof(int));
        if(array_min==NULL) {
            printf("2:error with realloc.\n");
            exit(1);
        }

        lower(test[a], array_min);
        for(i=0; i<ceil(test[a]/7.0);i++) {
            printf("%d", array_min[i]);
        }
        printf(" ");

        upper(test[a], array_max);
        for(i=0;i<(int)(test[a]/2.0);i++) {
            printf("%d", array_max[i]);
        }
        printf("\n");

        for(i=0;i<(int)(test[a]/2.0);i++) {
            array_max[i]=0;
        }
        for(i=0;i<ceil(test[a]/7.0);i++) {
            array_min[i]=0;
        }
    }

    free(array_min);
```

```
    free(array_max);
    return 0;
}
```

13.3 Bug History

This problem was not entirely without bugs after the first attempt. Initially it was tried to build the least number from the right hand side, and using the number 8 (which requires 7 matchsticks) until no more 8's could be used and splitting the difference among 1's and 7's etc. It turned out this would not work however, after discovering that it's solution to 17 matchsticks (788) was not correct (200 being the correct answer), and a different method needed to be used.

After starting on the r.h.s, the problem of leading zeroes needed to be avoided, which was done with a simple if statement.

Since the numbers 7 and 4 were only used once, an if statement was used which would output 3 and 4 respectively if either was read in.

13.4 Makefile

Memorandum

```
notangle -t8 -R4292.Makefile report.nw > Makefile
```

⟨*4292.Makefile*⟩≡

```
4292: report.nw
	/usr/bin/notangle -L -R4292 report.nw > 4292.c
	gcc -o 4292 -lm 4292.c

clean:
	rm *.c *.h *.tex *.aux *.log *.out

.SUFFIXES: .nw .tex .c

.nw.c: ;	/usr/bin/notangle 4292.nw > 4292.c
```

14 2008 Europe northwestern, problem G: solution II

Problem statement: http://livearchive.onlinejudge.org/external/42/4292.pdf

It can be shown reasonably easily that the maximum number that can be constructed is always either $7111\ldots111$ or $111\ldots111$. In particular, if the number of matchsticks m is even, then the largest number that can be constructed consists of $\frac{m}{2}$ 1's concatenated and if m is odd, the the largest number that can be constructed is 7 followed by $\frac{m-3}{2}$ 1's concatenated.

It is more difficult to construct the smallest number. Given m matchsticks, the least number of digits that can be used in constructing a number is

$$k = \lceil \frac{m}{7} \rceil$$

since we can maximally use 7 matches in constructing one digit (the case of 8). Denote by p^* the number of matches required to construct the digit p. We can determine the value of the leftmost digit of the smallest number by considering the following: if p is the left-most digit then p^* satisfies $p^* > m \bmod 7$ (if more than one p satisfies this, then we obviously choose the smallest such p). This must be so, since otherwise we would not be able to use up all the matches in filling every other position in the number.

After determining the left-most digit, we then try to use as many matches as possible towards the right of the number i.e. we try to place as many 8's there. At each stage we then check if the number of matches left is a multiple of 6 and, if so then we can insert zeroes in the remaining positions - provided this is possible, for example if there are 42 matches remaining and 6 positions to be filled, then we cannot, since we would still have 6 matches remaining. We continue this process from right-to-left until we reach the digit just preceeding the left-most-digit at which point we choose the digit that will use all remaining matches.

14.1 Globals

- `l-dig` contains the left-most digit of the smallest number that will be constructed by the program.
- `left` keeps track of how many matches are left at each stage of the program.
- `nod` is the number of digits that will appear in the smallest number constructed by the program.

⟨*4292x.variables*⟩≡
```
int i,l_dig,left=a,nod=(int)ceil((double)a/7);
unsigned long int temp=0UL;
div_t D=div(a,7);
```

- The `val[]` array contains the number of matches required to construct each digit viz. `val[i]` is the number of matches required to construct the digit `i`.
- The `ans[]` is the answer in all single digit cases - invert the above array.

⟨*4292x.globals*⟩≡
```
int val[10]={6,2,5,5,4,5,6,3,7,6};
int ans[6]={1,7,4,2,6,8};
```

14.2 Input

⟨*4292x.input*⟩≡
```
main() {
        int i,matches,cases;
        scanf("%d",&cases);

        for(i=0;i<cases;i++) {
                scanf("%d",&matches);
                printf("%ld ",min(matches));
                printmax(matches);
        }
        exit(0);
}
```

14.3 Printing the maximum

We must print the largest number digit by digit because the number becomes absolutely enormous.

⟨*4292x.print max*⟩≡
```
void printmax(int a) {
        int i;
        div_t D=div(a,2);
        printf("%d",(D.rem==0?1:7));
        for(i=0;i<D.quot-1;i++) printf("%d",1);
        printf("\n");
}
```

14.4 Finding the minimum

⟨*4292x.finding the minimum*⟩≡
```
unsigned long int min(int a) {
        if(a<=7) return ans[a-2];

        ⟨4292.variables⟩

        if(D.rem==0) i=8;
        else for(i=1;D.rem-val[i]>0;i++);
        l_dig=i;
        left-=val[i];
```

The code below takes care of the situation when the number of remaining matches is a multiple of 6 etc. as described previously.

⟨*4292x.finding the minimum*⟩+≡
```
        div_t p=div(left,6);
        if(p.rem==0 && p.quot==nod-1) return l_dig*pow(10,nod-1);
        else temp+=l_dig*pow(10,nod-1);

        for(i=1;i<nod-1;i++) {
                p=div(left,6);
                if(p.rem==0 && p.quot==nod-i) return temp;
                temp+=(8*pow(10,i-1));
                left-=7;
        }
```

The code below computes the digit just to the right of the left-most digit as outlined previously.

⟨*4292x.finding the minimum*⟩+≡
```
        for(i=0;val[i]!=left;i++);

        return temp+(pow(10,nod-2)*i);
}
```

14.5 Consolidated code

⟨*4292x*⟩≡
```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
```

⟨*4292x.globals*⟩

⟨*4292x.print max*⟩

⟨*4292x.finding the minimum*⟩

⟨*4292x.input*⟩

14.6 Debugging history

The program worked almost immediately once it was realised that the left-digit had to be dealt with separately i.e. the necessity of including

⟨*4292x.debug1*⟩≡
```
div_t p=div(left,6);
if(p.rem==0 && p.quot==nod-1) return l_dig*pow(10,nod-1);
else temp+=l_dig*pow(10,nod-1);
```

outside of the `for` loop in the `min[]` function.

14.7 Makefile

Memorandum

```
notangle -t8 -R4292x.Makefile report.nw > makefile
```

⟨*4292x.Makefile*⟩≡
```
4292x: report.nw
        /usr/bin/notangle -L -R4292x report.nw > 4292x.c
        gcc -o 4292 4292.c

clean:
        rm *.c *.h *.tex *.aux *.log *.out

.SUFFIXES: .nw .tex .c

.nw.c: ;        /usr/bin/notangle $*.nw > $*.c
```

15 2009 Europe northwestern, problem A

Problem statement: http://livearchive.onlinejudge.org/external/46/4609.pdf

We use a naive brute-force method - generate all unique permutations and test whether or not they are prime.

15.1 Globals

- `n` is the number of digits that appear in the number that is read in from the file.
- `comb[]` is used to store the combination of the digits that is currently being considered by the program.
- `per[]` vice versa the permutations.
- `buffer[]` is an array containing the integer digits of the number read-in from the file, from left to right.

⟨*4609.globals*⟩≡
```
int i,j,k,n,cases;
int comb[100],per[100],buffer[500];
char x;
FILE *fin;
```

15.2 Input

We need to preserve any leading zeroes in the input so we must take the digits in as `char`'s and then convert them back to integers using the `ctoi` function (see below).

⟨*4609.input*⟩≡
```
x=fgetc(fin);

for(i=0;x!='\n'&& x!=EOF;i++) {
  buffer[i]=ctoi(x);
  x=fgetc(fin);
}
```

15.3 Ancillary functions

The `ctoi` function converts a single digit number stored as a `char` into an `int`.

⟨*4609.ancillary*⟩≡
```
int ctoi(int a) {
  return a - '0';
}
```

The `not-in-list` function is used to ensure only unique primes are added to `plist[]`.

⟨*4609.ancillary*⟩+≡
```
int not_in_list(int D) {
  int i;
  for(i=0;i<p && plist[i]!=D;i++);

  return i==p? 1:0;
}
```

The `assem-num` function, given an array of integer digits, returns the number that would be obtained if the entries of the array were concatenated together.

⟨*4609.ancillary*⟩+≡
```
int assem_num(int* num, int v[],int n) {
  int j,temp=0;
  for (j=0;j<n;j++) temp+=(pow(10,n-j-1)*num[v[j]-1]);
  return temp;
}
```

15.4 Generating the permutations

Permutations are generated using a double pass method - the code is adapted from a description given in [CLRS]. Since we also need every permutation of each subset of digits of the number we also require a function to generate combinations of the digits. Rather than dealing with the actual digits, we generate all combinations of subsets of length m of the set $\{1, 2, 3, \ldots, n\}$ for each integer $m \leq n$, and then we assemble the numbers into a list only if they have not already been added - to ensure we don't count integers twice (this will happen with repeated digits, say with the number 889 where the number 89 will be generated twice because two 8's occur). Dealing only with the subset $\{1, 2, 3, \ldots, n\}$ makes the code significantly simpler.

⟨*4609.permutations*⟩≡
```
void swap(int v[], int i, int j) {
  int t=v[i];
  v[i]=v[j];
  v[j]=t;
}

void permute(int v[], int n, int i,int* num,int* list) {
  int j;

  if(i==n) {
     int D=assem_num(num,v,n);
     if( p_Q(D) && not_in_list(D)) {
       plist[p]=D;
       p++;
     }
  }
  else {
    for(j=i;j<n;j++) {
      swap(v,i,j);
      permute(v,n,i+1,num,list);
      swap(v,i,j);
    }
  }
}

int next_comb(int comb[], int k, int n) {
  int i=k-1;
  ++comb[i];

  while((i>=0) && (comb[i]>=n-k+1+i)) {
    --i;
    ++comb[i];
  }
```

```
    if(comb[0]>n-k) return 0;

    for(i=i+1;i<k;++i) comb[i]=comb[i-1]+1;

    return 1;
  }
```

15.5 Testing primality

We use the sieve of Eratosthenes as far as $\{2, 3, 5, 7\}$.

⟨*4609.testing primality*⟩≡

```
  int p_Q(int num) {
    int i;

    if(num==2||num==3||num==5||num==7) return 1;
    else if (num<=1||num%2==0||num%3==0||num%5==0||num%7==0) return 0;

    for(i=11;i<=sqrt(num);i+=4) {
      if(num%i==0) return 0;
      i+=2;
      if(num%i==0) return 0;
    }
    return 1;
  }
```

15.6 Consolidated code

The primes generated are inserted into the `plist[]` array. The integer `p` counts how many have been added.

⟨*4609*⟩≡

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

  #define FILENAME "input.txt"
  #define MAX 20

  int p,plist[10000];

  ⟨4609.ancillary⟩

  ⟨4609.testing primality⟩

  ⟨4609.permutations⟩

  main() {
    ⟨4609.globals⟩

    if( (fin=fopen(FILENAME,"r"))==NULL ) exit(1);

    if(fscanf(fin,"%d",&cases)!=1) exit(1);
    fgetc(fin);
```

The function of the `fgetc` above is to trash the first `\n` after the very first number (the number of cases in the file) read-in from the input file.

⟨*4609.main*⟩≡
```
  for(j=0;j<cases;j++) {
    ⟨4609.input⟩
    n=i,p=0;

    for(k=1;k<=n;k++) {
      for (i = 0; i < k; ++i) {
        comb[i] = i;
        per[i] = i+1;
      }
      permute(per,k,0,buffer,plist);

      while(next_comb(comb,k,n)) {
        for (i = 0; i < k; ++i) per[i] = comb[i]+1;
        permute (per,k,0,buffer,plist);
      }
    }
    printf("%d\n",p);
  }
  fclose(fin);
  exit(0);
}
```

15.7 Debugging history

When coding was initially finished, the program was unable to compute any correct answers because it could not deal with repeated digits. This required somewhat of a rewrite - it was required to construct an array to add all the numbers generated, if they had not been already added, and then to only output the unique numbers.

15.8 Makefile

Memorandum

```
notangle -t8 -R4609.Makefile report.nw > makefile
```

⟨*4609.Makefile*⟩≡
```
4609: report.nw
        /usr/bin/notangle -L -R4609 report.nw > 4609.c
        gcc -o 4609 4609.c

clean:
        rm *.c *.h *.tex *.aux *.log *.out

.SUFFIXES: .nw .tex .c

.nw.c: ;        /usr/bin/notangle $*.nw > $*.c
```

16 2011 Europe central, problem C

Problem statement: http://livearchive.onlinejudge.org/external/58/5880.pdf

The problem is very straight forward - we manipulate the ASCII values to make the program a little simpler. The actual ciphering is carried out in the following three lines

⟨5880.cipher⟩≡
```
shift=(int)key[i]-'A'+1;
pos=(int)plain-'A';
new='A'+((shift+pos)%26);
```

The `shift` variable contains the integer value of how much the letter must be shifted and `pos` determines the position of the plaintext letter in the alphabet. `new` is the ASCII value of the ciphered letter.

⟨5880⟩≡
```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define FILENAME "input.txt"

main() {
  char key[1000],plain;
  int i=0,keylen,pos,shift,new;
  FILE *fin;

  if( (fin=fopen(FILENAME,"r"))==NULL ) exit(1);
  fscanf(fin,"%s",key);

  while(key[0]!='0') {
    keylen=strlen(key);
    fgetc(fin);

    while((plain=fgetc(fin))!='\n') {

      ⟨5880.cipher⟩
      printf("%c",(char)new);
      i=((i+1)%keylen);
    }
    i=0;
    printf("\n");
    fscanf(fin,"%s",key);
  }
  fclose(fin);
  exit(0);
}
```

16.1 Makefile

Memorandum

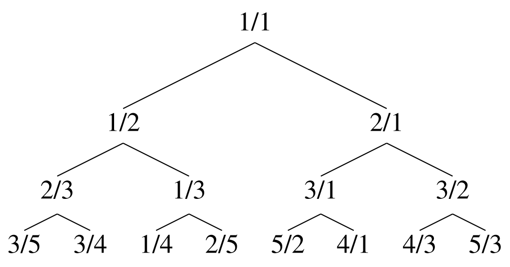
```
notangle -t8 -R5880.Makefile report.nw > makefile
```

⟨*5880.Makefile*⟩≡
```
5880: report.nw
        /usr/bin/notangle -L -R5880 report.nw > 5880.c
        gcc -o 5880 5880.c

clean:
        rm *.c *.h *.tex *.aux *.log *.out

.SUFFIXES: .nw .tex .c

.nw.c: ;        /usr/bin/notangle $*.nw > $*.c
```

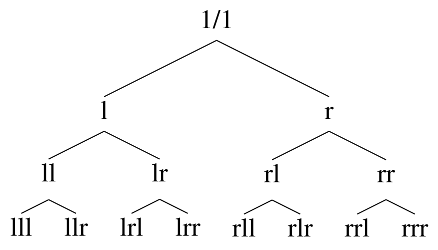
17 2011 Europe northwestern, problem B

Problem statement: http://livearchive.onlinejudge.org/external/59/5901.pdf

Define the left operation as $x \mapsto 1/(x+1)$ and the right operation as $x \mapsto \frac{1}{x} + 1$. The binary tree given to us in the problem statement is the following:



Each new child is obtained by pre-composition with either the given left or right operations. The above tree written as the iterations of the functions applied to obtain the value at that node is the following:



Notice that the operations shown at each node in the above tree are exactly the output we need, which leads to: given some input $\frac{a}{b}$ can we obtain the sequence of left/right operations that, when applied to $\frac{a}{b}$, will yield $\frac{1}{1}$? (Note that such a sequence is unique since we are told that each fraction appears exactly once in the tree.) The left, right operations and their inverses are:

$$\begin{aligned}
\text{left: } \frac{a}{b} &\mapsto \frac{b}{a+b}\\
\text{right: } \frac{a}{b} &\mapsto \frac{a+b}{a}\\
\text{left (inverse): } \frac{a}{b} &\mapsto \frac{b-a}{a}\\
\text{right (inverse): } \frac{a}{b} &\mapsto \frac{b}{a-b}
\end{aligned}$$

(Note that the inverse operations also yield reduced fractions - this is necessary to obtain a unique sequence of operations.) But since each of the denominator and numerator must be positive, there is, remarkably, only one choice at each stage! So we devise an algorithm: if $b - a$ is positive apply the inverse-left operation otherwise if $a - b$ is positive apply the inverse-right operation - continue until we obtain $\frac{1}{1}$. This process applied to $\frac{5}{7}$ is shown below

$$\frac{5}{7} \overset{l^{-1}}{\to} \frac{2}{5} \overset{l^{-1}}{\to} \frac{3}{2} \overset{r^{-1}}{\to} \frac{2}{1} \overset{r^{-1}}{\to} \frac{1}{1}$$

which yields the string LLRR.

$\langle 5901 \rangle \equiv$
```
  #include <stdlib.h>
```

```
#include <stdio.h>

#define 11ENWprobb "input.txt"

int main () {
  int i,cases;
  long int n,d,temp;
  FILE *fin;

  if( (fin=fopen(11ENWprobb,"r"))==NULL ) exit(1);
  fscanf(fin,"%d",&cases);

  for(i=0;i<cases;i++) {
    fscanf(fin,"%ld %*c %ld",&n,&d);

    while(n!=1 || d!=1) {
      if( (n-d)>0 ) {
        temp=n-d;
        n=d;
        d=temp;
        printf("R");
      }
      else {
        temp=d-n;
        d=n;
        n=temp;
        printf("L");
      }
    }

    printf("\n");
  }
  fclose(fin);
  exit(0);
}
```

17.1 Makefile

Memorandum

```
notangle -t8 -R5901.Makefile report.nw > makefile
```

⟨*5901.Makefile*⟩≡
```
5901: report.nw
        /usr/bin/notangle -L -R5901 report.nw > 5901.c
        gcc -o 5901 5901.c

clean:
        rm *.c *.h *.tex *.aux *.log *.out

.SUFFIXES: .nw .tex .c

.nw.c: ;        /usr/bin/notangle $*.nw > $*.c
```

18 2011 World finals, problem E

Problem statement: http://livearchive.onlinejudge.org/external/51/5132.pdf

This problem is to find the point in a $\mathrm{d}x \times \mathrm{d}y$ grid such that you are within a specified distance of the most possible coffee shops, the positions of which are given in the input.

A couple of methods were considered for solving this problem, such as calculating distances between all coffee shops or creating a matrix of the grid and radially inputting the distances from each coffee shop.

The method eventually decided upon sort of mirrors the minesweeper game in that after creating a matrix to simulate the grid for each element near a coffee shop i.e. within the query distance, the element is incremented by one. Then a search is made for the maximum value in that matrix following the rules outlined in the problem statement.

⟨*5132.filling the matrix*⟩≡
```
void load_shop(int x,int y,int xmax,int ymax,int q)
{
        int i,j,low_lim_x,hi_lim_x,low_lim_y,hi_lim_y,dist;
        low_lim_x=MAX(0,x-q);
        hi_lim_x=MIN(xmax-1,x+q);

        for(i=low_lim_x;i<=hi_lim_x;i++) {
                dist=q-abs(x-i);
                low_lim_y=MAX(0,y-dist);
                hi_lim_y=MIN(ymax-1,y+dist);

                for(j=low_lim_y;j<=hi_lim_y;j++) grid[i][j]+=1;
        }
}
```

18.1 Globals

- `dx` and `dy` are the x and y dimensions of the grid respectively
- `q-tot` is the total number of queries to be carried out on the data set, and `q` is the current query.
- The `grid` array contains the $\mathrm{d}x \times \mathrm{d}y$ grid of the coffee shop distances viz. if `grid[i][j]=3` say, then there are three coffee shops within distance `q` (using the taxi-cab metric) of the point (i, j).
- The `location` array is used as a container for the coffee shop locations input by the user.
- The `MAXX` variable is just the upper bound on the number of coffee shops that can be given as input.

⟨*5132.globals*⟩≡
```
#define MAXX 500000

int grid[1000][1000];
```

⟨*5132.variables*⟩≡
```
int i,j,k,l,set=1;
int x,y,max,sum;
int dx,dy,q,q_tot,n;
int location[MAXX][2];
```

18.2 Input & output

⟨*5132.main code*⟩≡
```
int main()
{
  if(scanf("%d %d %d %d", &dx, &dy, &n, &q_tot)!=4) exit(1);
  sum=dx+dy+q_tot+n;
  init_grid(dx,dy);

  while(sum>0) {
    int max,xtemp,ytemp;

    for(k=0;k<n;k++) {
      if(scanf("%d %d", &i, &j)!=2) exit(1);
      location[k][0]=i;
      location[k][1]=j;
    }

    printf("Case %d:\n",set);

    for(l=1;l<=q_tot;l++) {
      max=0;
      xtemp=dx;
      ytemp=1;

      if(scanf("%d", &q)!=1) exit(1);
      for(k=0;k<n;k++) load_shop(location[k][0]-1, location[k][1]-1, dx, dy, q);

      for(i=0;i<dx;i++) {
        for(j=0;j<dy;j++) {
          if(max<grid[i][j] || (max==grid[i][j] && ytemp>j)
                || (max==grid[i][j] && ytemp==j && xtemp<i)) {
            max=grid[i][j];
            xtemp=(i+1);
            ytemp=(j+1);
          }
        }
      }

      printf("%d (%d,%d)\n",max, xtemp, ytemp);

      init_grid(dx,dy);
    }

    if(scanf("%d %d %d %d", &dx, &dy, &n, &q_tot)!=4) exit(1);
    sum = dx+dy+q_tot+n;

    set++;
  }
  exit(0);
}
```

18.3 Consolidated code

⟨*5132*⟩≡
```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define MAX(x, y) x>y ? (x) : (y)
#define MIN(x, y) x<y ? (x) : (y)

⟨5132.globals⟩

void init_grid(dx,dy) {
        int i,j;
        for(i=0;i<dx;i++)
                for(j=0;j<dy;j++)
                        grid[i][j]=0;
}

⟨5132.filling the matrix⟩

⟨5132.main code⟩
```

18.4 Debugging report

The first instance of the program did not function correctly at all - this was due to problems with the searching part of the program. We could not identify the problem, so we simply rewrote it, and then the program worked super-fantastically.

18.5 Makefile

Memorandum

```
notangle -t8 -R5132.Makefile report.nw > makefile
```

⟨*5132.Makefile*⟩≡
```
5132: report.nw
        /usr/bin/notangle -L -R5132 report.nw > 5132.c
        gcc -o 5132 5132.c

clean:
        rm *.c *.h *.tex *.aux *.log *.out

.SUFFIXES: .nw .tex .c

.nw.c: ;        /usr/bin/notangle $*.nw > $*.c
```

Addendum: compiling this document

Modifications or additions can be made to the `report.nw` file and then compiled using the following makefile.

Memorandum

```
notangle -t8 -Rcompilation report.nw > makefile
```

⟨*compilation*⟩≡

```
dvi: report.nw
        /usr/bin/noweave -delay report.nw > report.tex
        latex report
        latex report
        rm report.log report.out

dvipdfm: dvi
        dvipdfm report

clean:
        rm *.c *.h *.tex *.aux *.log *.out

.SUFFIXES: .nw .tex .c

.nw.c: ;        /usr/bin/notangle $*.nw > $*.c
```