```
/*
 * Problem A.
 */
#include <stdio.h>
#include <stdlib.h>


int  n;
int *pic;
int check_figure(int i, int j, int k);


int main(void)
{
  int  max_half_size, i, j, k, n_minus_one, n_minus_two;
  int *row;

  while ( 1 ) {
    /* Get picture dimensions */
    scanf("%d", &n);
    if ( n == 0 )
      break;
    n_minus_one = n - 1;
    n_minus_two = n_minus_one - 1;

    /* Get picture */
    pic = (int *) malloc( sizeof(int) * n * n );
    for ( i = 0; i < n * n; i++ )
      scanf("%d", pic + i );

    max_half_size = 0;

    /* Main loop--search for the first white cell, this will be a candidate for
       an upper vertex of white square inside the figure we are looking for.  No
       use to check first and last columns and two last rows--they cannot
       contain the upper vertex we are looking for.  */
    for ( i = 0; i < n_minus_two; i++ ) {
      row = pic + i * n;
      for ( j = 1; j < n_minus_one; j++ ) {
        if ( !*(row + j ) ) {
          /* Now, go left and right from this vertex, ‘expanding’ width of the
             candidate figure, as long as we do not hit the border or we have no
             more black cells both to the left and to the right of this vertex.
             In each step check whether we got our figure.  If we did not, try
             to ‘expand’ the width more; if we did, record its width and
             break--there is no use ‘expanding’ further, we will not get any
             more figures because it would have to contain the one we found,
             which is impossible.  */
          for ( k = 1; *(row+ j-k) && *(row+ j+k) && j-k >= 0 && j+k < n; k++ )
            if ( check_figure(i, j, k) ) {
              if ( k > max_half_size )
                max_half_size = k;
              j += k; /* skip right after the figure */
              break;
            }
        }
      }
    }
    if ( max_half_size )
      printf("%d\n", max_half_size * 2 + 1);
    else
      puts("No solution");

    free(pic);
  }

  return 0;
}

int check_figure(int vertex_i, int vertex_j, int half_size)
{
  int i, j;
  int *row_vertex;

  /* Check vertical width does not violate the border (no use to check
     horizontal width--it has been done in the main loop) */
  if ( vertex_i + half_size * 2 + 1 > n )
    return 0;

  /* Check upper half of the figure (center line included).  Do not check the
     first line--it has been checked in the main loop) */
  for ( i = 1; i <= half_size; i++ ) {
    row_vertex = pic + (vertex_i + i) * n + vertex_j;
    /* Check white cells are there */
    for ( j = 0; j <= i; j++ )
      if ( *(row_vertex + j) || *(row_vertex - j) )
        return 0;
    /* Check black cells are there */
    for ( j = half_size; j > i; j-- )
      if ( *(row_vertex + j) != 1 || *(row_vertex - j) != 1 )
        return 0;
  }

  /* Check lower half of the figure (center line excluded) */
  for ( i = 1; i <= half_size; i++ ) {
    row_vertex = pic + (vertex_i + half_size + i) * n + vertex_j;
    /* Check white cells are there */
    for ( j = 0; j <= half_size - i; j++ )
      if ( *(row_vertex + j) != 0 || *(row_vertex - j) != 0 )
        return 0;
    /* Check black cells are there */
    for ( j = half_size; j > half_size - i; j-- )
      if ( *(row_vertex + j) != 1 || *(row_vertex - j) != 1 )
        return 0;
  }

  return -1;
}



/*
 * Local variables:
 * compile-command: "gcc -Wall a.c && ./a.out < a.in"
 * End:
 */
```

```c
/*
 * Problem B.
 */
#include <stdio.h>
#include <stdlib.h>


#define STACK_MEM_CHUNK      245  /* does funny things when set to anything less
                                     than 245 */
#define MAX_STACKS_COUNT    1000


typedef struct _stack {
  long *ptr;
  int   len;
  int   top;
} stack, *stackptr;

stackptr stacks;


int main(void)
{
  int  n, i;
  long a, b;
  int  tmp;
  stackptr ptrStack;

  /* Get number of operations */
  scanf("%d", &n);

  /* Set up the list of stacks */
  stacks = (stackptr) malloc( sizeof(stack) * MAX_STACKS_COUNT );
  for ( i = 0; i < MAX_STACKS_COUNT; i++ ) {
    (stacks + i)->ptr = NULL;
    (stacks + i)->len = 0;
    (stacks + i)->top = 0;
  }

  /* Main loop */
  for ( i = 0; i < n; i++ ) {
    tmp = scanf(" POP %ld", &a);
    if ( tmp == 1 ) {            /* POP */
      ptrStack = stacks + a;
      printf("%ld\n", *( ptrStack->ptr + --ptrStack->top ) );
    } else {                     /* PUSH */
      scanf("USH %ld %ld", &a, &b);
      ptrStack = stacks + a;
      /* Grow stack if needed */
      while ( ptrStack->len <= ptrStack->top ) {
        ptrStack->len += STACK_MEM_CHUNK;
        ptrStack->ptr = (long *) realloc( (void *) ptrStack->ptr, ptrStack->len
);
      }
      *( ptrStack->ptr + ptrStack->top++ ) = b;
    }
  }

  return 0;
}



/*
 * Local variables:
 * compile-command: "gcc -Wall b.c && ./a.out <<< \"7  PUSH 1 100  PUSH 1 200  P
USH 2 300  PUSH 2 400  POP 2  POP 1  POP 2\""
 * End:
 */
```

```
/*
 * Problem C.
 */
#include <stdio.h>
#include <stdlib.h>


#define MAX_RABBIT_COUNT        200


void combinations(int, int);
void check_line(void);

int sel[2]; /* A pair of special rabbits we select */
int n;
int x[MAX_RABBIT_COUNT];
int y[MAX_RABBIT_COUNT];
int max = 0;


int main(void)
{
  int i;

  /* Get the stuff */
  scanf("%d", &n);
  for(i = 0; i < n; i++)
    scanf("%d%d", x + i, y + i);

  combinations(0, 0);

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

  return 0;
}

void combinations(int i, int nsel)
{
  if ( nsel == 2 ) {
    check_line();
    return;
  }

  if ( i == n - 1 ) {
    sel[nsel] = i;
    check_line();
    return;
  }

  combinations(i + 1, nsel);

  sel[nsel] = i;
  combinations(i + 1, nsel + 1);

  return;
}

void check_line()
{
  int i;
  int r1 = sel[0], r2 = sel[1];
  int cur = 2;

  /* See how many rabbits located on the line with the two selected rabbits */
  for( i = 0; i < n; i++ )
    if( i != r1 && i != r2 )
      if( ((long) ( x[r2] - x[r1])) * ( y[i] - y[r2]) ==
          ((long) ( y[r2] - y[r1])) * ( x[i] - x[r2])    )
        cur++;

  if( cur > max )
    max = cur;
}



/*
 * Local variables:
 * compile-command: "gcc -Wall c.c && ./a.out <<< \" 6  7 122  8 139  9 156  10
173  11 190  -100 1\""
 * End:
 */
```

```
/*
 * Problem D.
 */
#include <stdio.h>
#include <stdlib.h>

#define MEM_ALLOC_CHUNK     10

int **lists;
int  *already_out;
int  *sorted;
int   sorted_count;
int   n;


void find_descendents(int martian);


int main(void)
{
  int i, j, len, temp;

  scanf("%d", &n);

  lists       = (int **) malloc( sizeof(int *) * n );
  already_out = (int *)  malloc( sizeof(int)   * n );
  sorted      = (int *)  malloc( sizeof(int)   * n );

  /* Enter the lists */
  for ( i = 0; i < n; i++ ) {
    len = 0;
    j   = 0;
    *(lists + i) = NULL;
    *(already_out + i) = 0;
    do {
      scanf("%d", &temp);
      /* Grow list if necessary */
      while ( len <= j ) {
        len += MEM_ALLOC_CHUNK;
        *(lists + i) = (int *) realloc((void *) *(lists + i), len*sizeof(**lists
));
      }
      *(*(lists + i) + j++) = temp;
    } while ( temp );
  }

  /* Main loop */
  sorted_count = 0;
  for ( i = 0; i < n && sorted_count < n; i++ )
    find_descendents(i);

  /* Output the sorted list of members (it is in reverse order) */
  for ( i = n - 1; i > 0; i-- )
    printf("%d ", *(sorted + i) + 1);
  printf("%d", *sorted + 1);

  return 0;
}
```

```
void find_descendents(int martian)
{
  int j = 0;

  /* First, recursively output all descendents, if any */
  while ( *(*(lists + martian) + j) )
    find_descendents( *(*(lists + martian) + j++) - 1 );

  /* Now, output this Martian if he (she? it?) is not yet out */
  if ( !*(already_out + martian) ) {
    *(already_out + martian)   = -1;
    *(sorted + sorted_count++) = martian;
  }

  return;
}



/*
 * Local variables:
 * compile-command: "gcc -Wall d.c && ./a.out <<< \"5   0   4 5 1 0   1 0   5 3
0   3 0\""
 * End:
 */
```

```
/*
 * Problem E.
 */
#include <stdio.h>

void s_out(int n);
void a_out(int n);

int main(void)
{
  int n;

  scanf("%d", &n);

  s_out(n);

  return 0;
}

void s_out(int n)
{
  int i;

  for ( i = 1; i < n; i++ )
    putchar('(');
  for ( i = 1; i < n; i++ ) {
    a_out(i);
    printf("+%d)", n - i + 1);
  }
  a_out(n);
  printf("+%d\n", n - i + 1);

  return;
}

void a_out(int n)
{
  int i;
  int plus;

  for ( i = 1, plus = 0; i < n; i++, plus = ~plus )
    printf("sin(%d%c", i, plus ? '+' : '-');
  printf("sin(%d", i);
  for ( i = 1; i <= n; i++)
    putchar(')');

  return;
}



/*
 * Local variables:
 * compile-command: "gcc -Wall e.c && ./a.out <<< 3"
 * End:
 */
```

```c
/*
 * Problem F.
 */
#include <stdio.h>
#include <stdlib.h>


typedef struct _TSpan {
  int s;
  int e;
} TSpan, *ptrTSpan;


int      n;
ptrTSpan ttbl;
int      count;


int main(void)
{
  int   i, j, bubbled_up;
  TSpan tspan;

  /* Get time-table length */
  scanf("%d", &n);

  /* Allocate buffers */
  ttbl = (ptrTSpan) malloc( sizeof(TSpan) * n );

  /* Enter time-table */
  for ( i = 0; i < n; i++ )
    scanf("%d%d", &(ttbl + i)->s, &(ttbl + i)->e);

  /* Sort time-table by start time */
  for ( i = 0; i < n; i++ ) {
    bubbled_up = 0;
    for ( j = i + 1; j < n; j++ )
      if ( (ttbl+i)->s > (ttbl+j)->s ) {
        tspan       = *(ttbl + i);
        *(ttbl + i) = *(ttbl + j);
        *(ttbl + j) = tspan;
        bubbled_up  = -1;
      }
    if ( !bubbled_up )
      break;
  }

  /* Main loop.  What we do is we find the sequence of ‘optimal’ time intervals.
     The idea behind the algorithm for finding this sequence is as follows
     (array of intervals is assumed to be sorted by start time):

     1. make first interval current;
     2. make current interval a candidate for next optimal interval (OptInt);
     3. search forward for an interval starting and ending before OptInt ends;
     4. if no such interval found, go to step 5; else make the interval we found
        an OptInt and go to step 3;
     5. fix current OptInt; skip all intervals which do not start after the
        fixed OptInt; go to step 2;
  */
  count = 0;
  i     = 0;
  while ( i < n ) {
    /* Find OptInt */
    for ( j = i + 1; j < n && (ttbl + j)->s < (ttbl + i)->e; j++ )
      if ( (ttbl + j)->e < (ttbl + i)->e )
        i = j;
    /* Another interval found--increment counter */
    count++;
    /* Skip until intervals starting after the interval we found ends */
    while ( j < n && (ttbl + j)->s <= (ttbl + i)->e )
      j++;
    i = j;
  }

  printf("%d\n", count);

  return 0;
}



/*
 * Local variables:
 * compile-command: "gcc -Wall f.c && ./a.out <<<\"5  3 4  1 5  6 7  4 5  1 3\""
 * End:
 */
```

```
/*
 * Problem G.
 */
#include <stdio.h>
#include <stdlib.h>

int  k;
int  len;
int *result;
int *scratch;

int main(void)
{
  int i, j, carry;

  scanf("%d", &k);

  /* Length for result buffer:  36 * 55^{k-1} < 100 * 100^{k-1} = 10^{2k} */
  len     = k + k;
  result  = (int *) malloc( sizeof(*result)  * len );
  /* We could make scratch buffer length one less than result buffer length, but
     algorithm below will write 0 to this extra buffer element, so leave it as
     it is */
  scratch = (int *) malloc( sizeof(*scratch) * len );

  *result       = 6;
  *(result + 1) = 3;
  j             = 2;

  while ( --k ) {
    /* Multiply result buffer by 5 saving it in scratch buffer */
    carry = 0;
    for ( i = 0; i < j; i++ ) {
      *(scratch + i) = *(result + i) * 5 + carry;
      if ( *(scratch + i) >= 10 ) {
        carry = *(scratch + i) / 10;
        *(scratch + i) %= 10;
      } else
        carry = 0;
    }
    if ( carry )
      *(scratch + i++) = carry;

    /* Add scratch buffer to itself shifted one digit saving it in result
       buffer */
    *(scratch + i) = 0; /* we have reserved this extra element (see above) */
    *result = *scratch;
    carry   = 0;
    for ( j = 1; j <= i; j++ ) {
      *(result + j) = *(scratch + j) + *(scratch + j - 1) + carry;
      if ( *(result + j) >= 10 ) {
        carry = *(result + j) / 10;
        *(result + j) %= 10;
      } else
        carry = 0;
    }
    if ( carry )
      *(result + j++) = carry;
  }
  /* Print out the result */
  for ( i = j - 1; i >= 0; i-- )
    putchar(*(result + i) + '0');
  putchar('\n');

  return 0;
}



/*
 * Local variables:
 * compile-command: "gcc -Wall g.c && ./a.out <<< 5"
 * End:
 */
```

```
/*
 * Problem H.
 */
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

#define INT_WIDTH   (sizeof(int) * CHAR_BIT)

int main(void)
{
  unsigned eggs, stories, skip;
  unsigned min_exp;

  while ( 1 ) {
    scanf("%ud", &eggs);
    if ( !eggs )
      break;
    scanf("%ud", &stories);

    /* Compute how many stories we gonna skip, i.e. 2^{eggs-1} */
    if ( eggs > INT_WIDTH )
      eggs = INT_WIDTH;
    skip = 1 << (eggs - 1);

    /* Calculate number of experiments */
    if ( skip <= stories )
      min_exp = stories / skip + eggs - 1;
    else {
      min_exp = eggs - 1;
      while ( (skip >>= 1) > stories )
        min_exp--;
    }

    printf("%u\n", min_exp);
  }

  return 0;
}



/*
 * Local variables:
 * compile-command: "gcc -Wall h.c && ./a.out <<<\"1 10  2 5   0\""
 * End:
 */
```

```
/*
 * Problem I.
 */
#include <stdio.h>
#include <stdlib.h>

unsigned long **blocks;
char outbuffer[15];

int main(void)
{
  int n, i, j, k;
  int carry, nonzero;
  unsigned long *ptr, *cur;

  scanf("%d", &n);

  /* Allocate a ‘triangular’ 2-D buffer.  In this buffer, i-th element is
     pointer to the vector of numbers of staircases we can build of i bricks.
     j-th element in this vector is number of staircases we can build of i
     bricks if we use up j bricks for the highest step. */
  blocks = (unsigned long **) malloc( sizeof(*blocks) * n );
  for ( i = 0; i < n; i++ )
    *(blocks + i) = (unsigned long *) malloc( sizeof(**blocks) * (i + 1) );

  /* Fill the buffer up, calculating the vectors for i bricks on the basis of
     the vectors we already have filled in for i-1, i-2, ..., 1 bricks. */
  for ( i = 0; i < n; i++ ) {  /* take from 1 to n bricks */
    **(blocks + i) = 0;        /* no staircases with step of height 1 (for the
                                  very first staircase of 1 brick, this will be
                                  changed to 1, see below) */
    for ( j = 1; j < i; j++ ) {/* use from 1 to i - 1 bricks for the highest
                                  step and calculate how many staircases we can
                                  build below this step from remaining bricks */
      cur  = *(blocks + i) + j;
      *cur = 0;                /* zero the counter */
      ptr  = *(blocks + i - j - 1); /* (i - j) is number of bricks left, so look
                                       in (i - j)-th vector */
      /* Add up the staircases we can build with remaining bricks */
      k = j - 1; /* Start with step one brick lower than the highest step but */
      if ( k > i - j - 1 ) /* make sure there are enough bricks left for step */
        k = i - j - 1;     /* of such height (we have (i-j) bricks left).     */
      for ( ; k >= 0; k-- )
        *cur += *(ptr + k);
    }
    *(*(blocks + i) + i) = 1;   /* Though we cannot build a staircase with just
                                   one stair of i bricks, make this 1 so that
                                   future staircases would include such step */
  }

  /* Add up numbers in n-th vector */
  i = n - 1;
  k = 0;
  carry = 0;
  do {
    nonzero = 0;
    for ( j = n - 2; j >= 0; j-- )    /* don’t count last element (see above) */
      if ( *(*(blocks + i) + j) ) {
        nonzero = -1;
        carry += *(*(blocks + i) + j) % 10;
        *(*(blocks + i) + j) /= 10;
      }
    *(outbuffer + k++) = carry % 10;
    carry /= 10;
  } while ( nonzero && carry );
  while ( carry ) {
    *(outbuffer + k++) = carry % 10;
    carry /= 10;
  }

  /* Print out the total */
  for ( j = k - 1; j >= 0; j-- )
    printf("%d", (int) *(outbuffer + j));
  putchar('\n');

  return 0;
}



/*
 * Local variables:
 * compile-command: "gcc -Wall i.c && ./a.out <<< 212"
 * End:
 */
```