DESIGN &
ANALYSIS OF
ALGORITHM

Analysis of Game of Life V.1

DESIGN &
ANALYSIS OF
ALGORITHM

```
#include    "common.h"   /* common include files and definitions */
#include    "life.h"     /* Life's defines, typedefs, */
void main(void)
{
    int  row, col;
    Grid map;                      /* current generation   */
    Grid newmap;                   /* next generation      */

    Initialize(map);
    WriteMap(map);
    printf("This is the initial configuration you have chosen.\n"
           "Press <Enter> to continue.\n");
    while(getchar() != '\n')
        ;
    do {
        for (row = 1; row <= MAXROW;    row++)
            for (col = 1; col <= MAXCOL; col++)
                switch(NeighborCount(map, row, col)) {
                case 0:
                case 1:
                    newmap[row][col]= DEAD;
                    break;
                case 2:
                    newmap[row][col]= map[row][col];
                    break;
                case 3:
                    newmap[row][col]= ALIVE;
                    break;
                case 4:
                case 5:
                case 6:
                case 7:
                case 8:
                    newmap[row][col]= DEAD;
                    break;
                }
        CopyMap(map, newmap);
        WriteMap(map);
        printf("Do you wish to continue ");
    } while (UserSaysYes());
}
```

Main.c

We will ignore I/O and count statements.

1200x(12+2)+1200=1800

Neighbor Count

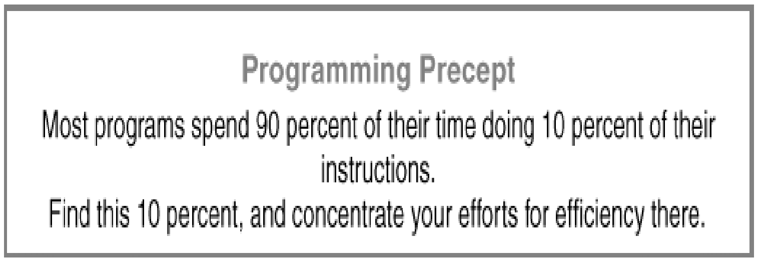
DESIGN &
ANALYSIS OF
ALGORITHM

```
int NeighborCount(Grid map, int row, int col)
{
    int i;          /*row of a neighbor */
    int j;          /*column ofaneighbor */

    int count = 0;

    for (i = row - 1; i <= row + 1; i++)
        for (j=col - 1; j<= col + 1; j++)
            if (map[i][j] == ALIVE)
                count++;
    if (map[row][col] == ALIVE)
        count--;
    return count;
}
```

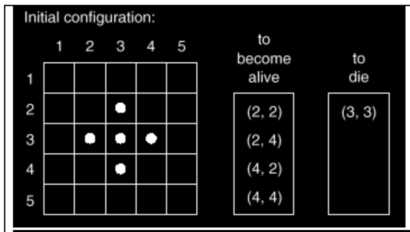
1 +2 + 9 =12

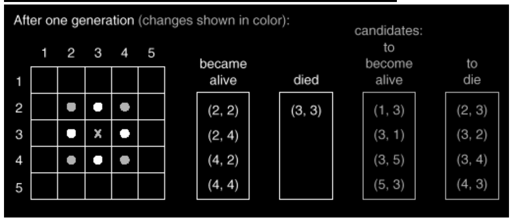


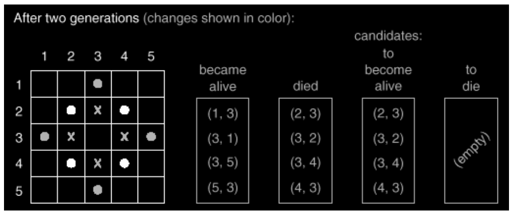
Programming Precept

Most programs spend 90 percent of their time doing 10 percent of their instructions.
Find this 10 percent, and concentrate your efforts for efficiency there.

How to Improve?







Use Scratch paper..

DESIGN & ANALYSIS OF ALGORITHM

LECT-02, S-5
ALG00S, javed@kent.edu
Javed I. Khan@1999

Game of Life 2

Get the initial configuration of living cells and use it to calculate an array holding the neighbor counts of all cells. Determine the cells that will become alive and that will become dead in the first generation;

Repeat the following steps as long as desired:

Vivify each cell that is ready to become alive;

Kill each cell that is ready to die;

Write out the map for the user;

Increase the neighbor counts for each neighbor of each cell that has become alive; If a neighbor count reaches the appropriate value, then keep track of the cell as a candidate to be made alive or dead in the next generation;

Decrease the neighbor counts for each neighbor of each cell that has become dead; If a neighbor count reaches the appropriate value, then keep track of the cell as a candidate to be made alive or dead in the next generation.

DESIGN & ANALYSIS OF ALGORITHM

LECT-02, S-6
ALG00S, javed@kent.edu
Javed I. Khan@1999

void AddNeighbors(ListEntry cell);

Pre: cell has just become alive.

Post: Array numbernbrs has increased counts for all the cells neighboring cell. If the cell thereby becomes a candidate to be vivified [*resp.* killed] then the cell has been added to list maylive [*resp.* maydie].

Uses: Function AddList; changes array numbernbrs and lists maylive and maydie as global variables (side effects).

void Vivify(ListEntry cell);

Pre: The cell is a candidate to become alive.

Post: Checks that cell meets all the requirements to become alive. If not, no change is made. If so, then cell is added to the list newlive, and array map is updated.

Uses: Function AddList, array numbernbrs, changes array map and list newlive as global variables (side effects).

At the beginning of the main loop, list maylive contains only dead cells, and list maydie contains only living cells. These lists may contain duplicate entries or spurious entries with wrong neighbor counts. The lists newlive and newdie are empty.

DESIGN & ANALYSIS OF ALGORITHM

Main2.c

```
#include "common.h"
#include "life2.h"

Grid map;           /* global: square array holding cells
*/
Gridcount numbernbrs;/* global: square array holding neighbor counts
*/
List newlive,  /* global: the cells that have just been vivified
*/
     newdie,       /* global: the cells that have just died
*/
     maylive,  /* global: candidates to vivify in the next generation
*/
     maydie;       /* global: candidates to kill in the next generation
*/
int maxrow, maxcol; /* global: user defined grid size
*/
int main(void)
{
     Initialize(map, numbernbrs, &newlive, &newdie, &maylive, &maydie);
     WriteMap(map);
     printf("Proceed with the demonstration");
     while (UserSaysYes()) {
          TraverseList(&maylive,Vivify);/* uses numbernbrs, changes map
and newlive */
          TraverseList(&maydie, Kill);  /* uses numbernbrs, changes map
and newdie */
          WriteMap(map);
          ClearList(&maylive);
          ClearList(&maydie);
          TraverseList(&newlive, AddNeighbors); /* changes numbernbrs,
maylive, maydie */
          TraverseList(&newdie, SubtractNeighbors);
          ClearList(&newlive);
          ClearList(&newdie);
          printf("Do you want to continue viewing new generations");
     }
     return 0;
}
```

adds them to the **newlive** array

adds some of them to **maylive** or **maydie** array

DESIGN & ANALYSIS OF ALGORITHM

DESIGN &
ANALYSIS OF
ALGORITHM

Vivify

```
void Vivify(ListEntry cell)
{
    if (map[cell.row][cell.col] == DEAD &&
        numbernbrs[cell.row][cell.col] == 3)
        if (cell.row >= 1 && cell.row <= maxrow &&
/* not on hedge */
            cell.col >= 1 && cell.col <= maxcol) {
            map[cell.row][cell.col] = ALIVE;
            AddList(cell, &newlive);
        }
}
```

DESIGN &
ANALYSIS OF
ALGORITHM

AddNeighbors

```
void AddNeighbors(ListEntry cell)
{
    int nbrrow,     /* loop index for row of neighbor loops */
        nbrcol;     /* column loop index    */
    Cell neighbor;  /* structure form of a neighbor */

    for (nbrrow = cell.row-1; nbrrow <= cell.row+1; nbrrow++)
        for (nbrcol = cell.col-1; nbrcol <= cell.col+1; nbrcol++)
            if (nbrrow != cell.row || nbrcol != cell.col) { /* Skip cell itself. */
                numbernbrs[nbrrow][nbrcol]++;
                switch (numbernbrs[nbrrow][nbrcol]) {

                    case 0:
                         Error("Impossible case in AddNeighbors.");
                         break;


                    case 3:
                         if (map[nbrrow][nbrcol] == DEAD) {
                              neighbor.row = nbrrow; /* Set up a coordinate record. */
                              neighbor.col = nbrcol;
                              AddList(neighbor, &maylive);
                         }
                         break;


                    case 4:
                         if (map[nbrrow][nbrcol] == ALIVE) {
                              neighbor.row = nbrrow; /* Set up a coordinate record. */
                              neighbor.col = nbrcol;
                              AddList(neighbor, &maydie);
                         }
                         break;
                }   /* switch statement */
            }
}
```

Analysis of Version 2

11

Analysis of Version 2

DESIGN & ANALYSIS OF ALGORITHM

- Ignore I/O
- All most all work is done in the 4 traversing statements.
- The amount of work is no longer dependant on the size of the grid but on the number of state changes.
- If there are 50 cells occupied for a typical configuration, then on the average there will be 25 state changes.

LECT-02, S-12
ALG00S, javed@kent.edu
Javed I. Khan@1999

Main2.c

```
#include "common.h"
#include "life2.h"

Grid map;           /* global: square array holding cells                  */
Gridcount numbernbrs;/* global: square array holding neighbor counts        */
List newlive,  /* global: the cells that have just been vivified       */
     newdie,       /* global: the cells that have just died                */
     maylive,  /* global: candidates to vivify in the next generation  */
     maydie;       /* global: candidates to kill in the next generation    */
int maxrow, maxcol; /* global: user defined grid size                      */
int main(void)
{
     Initialize(map, numbernbrs, &newlive, &newdie, &maylive, &maydie);
     WriteMap(map);
     printf("Proceed with the demonstration");
     while (UserSaysYes()) {
          TraverseList(&maylive,Vivify);/* uses numbernbrs, changes map and
newlive */
          TraverseList(&maydie, Kill);  /* uses numbernbrs, changes map and
newdie */
          WriteMap(map);
          ClearList(&maylive);
          ClearList(&maydie);
          TraverseList(&newlive, AddNeighbors); /* changes num
maydie */
          TraverseList(&newdie, SubtractNeighbors);
          ClearList(&newlive);
          ClearList(&newdie);
          printf("Do you want to continue viewing new generations");
     }
     return 0;
}
```

We will ignore I/O and count statements.

DESIGN & ANALYSIS OF ALGORITHM

Vivify

```
void Vivify(ListEntry cell)
{
    if (map[cell.row][cell.col] == DEAD &&
        numbernbrs[cell.row][cell.col] == 3)
        if (cell.row >= 1 && cell.row <= maxrow
&&      /* not on hedge */
            cell.col >= 1 && cell.col <= maxcol)
{
            map[cell.row][cell.col] = ALIVE;
            AddList(cell, &newlive);
        }
}
```

≈4

DESIGN & ANALYSIS OF ALGORITHM

AddNeighbors

```
void AddNeighbors(ListEntry cell)
{
    int nbrrow,     /* loop index for row of neighbor loops */
        nbrcol;     /* column loop index    */
    Cell neighbor;  /* structure form of a neighbor */

    for (nbrrow = cell.row-1; nbrrow <= cell.row+1; nbrrow++)
        for (nbrcol = cell.col-1; nbrcol <= cell.col+1; nbrcol++)
            if (nbrrow != cell.row || nbrcol != cell.col) { /* Skip cell itself. */
                numbernbrs[nbrrow][nbrcol]++;
                switch (numbernbrs[nbrrow][nbrcol]) {

                    case 0:
                         Error("Impossible case in AddNeighbors.");
                         break;


                    case 3:
                         if (map[nbrrow][nbrcol] == DEAD) {
                              neighbor.row = nbrrow; /* Set up a coordinate record. */
                              neighbor.col = nbrcol;
                              AddList(neighbor, &maylive);
                         }
                         break;


                    case 4:
                         if (map[nbrrow][nbrcol] == ALIVE) {
                              neighbor.row = nbrrow; /* Set up a coordinate record. */
                              neighbor.col = nbrcol;
                              AddList(neighbor, &maydie);
                         }
                         break;
                }   /* switch statement */
            }
```

9*(1+1+4)≈54

DESIGN & ANALYSIS OF ALGORITHM

Comparisons

- Time complexity
  - Total= 25*(4+54+4+54)≈2900 as opposed to 18,000 statements!
  - 6 Times saving!
- How about Space Complexity?
- V.1 requires about 2x20x60 bits= 300 bytes.
- V.2 needs
  - 150 bytes for one map.
  - 2400 bytes numbernbrs for 1200 grid points.
  - 4x600=2400 bytes for four lists (assuming 25% per list)
  - A total of 5000 bytes!
- Plus the extra programming effort!

DESIGN & ANALYSIS OF ALGORITHM