Valid input
1 Tetriminos = 4x4 char array
# of Tertiminos range = {1, 26}
Valid Chars = {'.', '#'}
All lines end in '\n'
All Tetriminos 4x4 matricies must be seperated by a '\n'
Tetriminos Must resemble classic tetris pieces.
Each block ('#') must touch at least one other block on any of it's 4 sides.

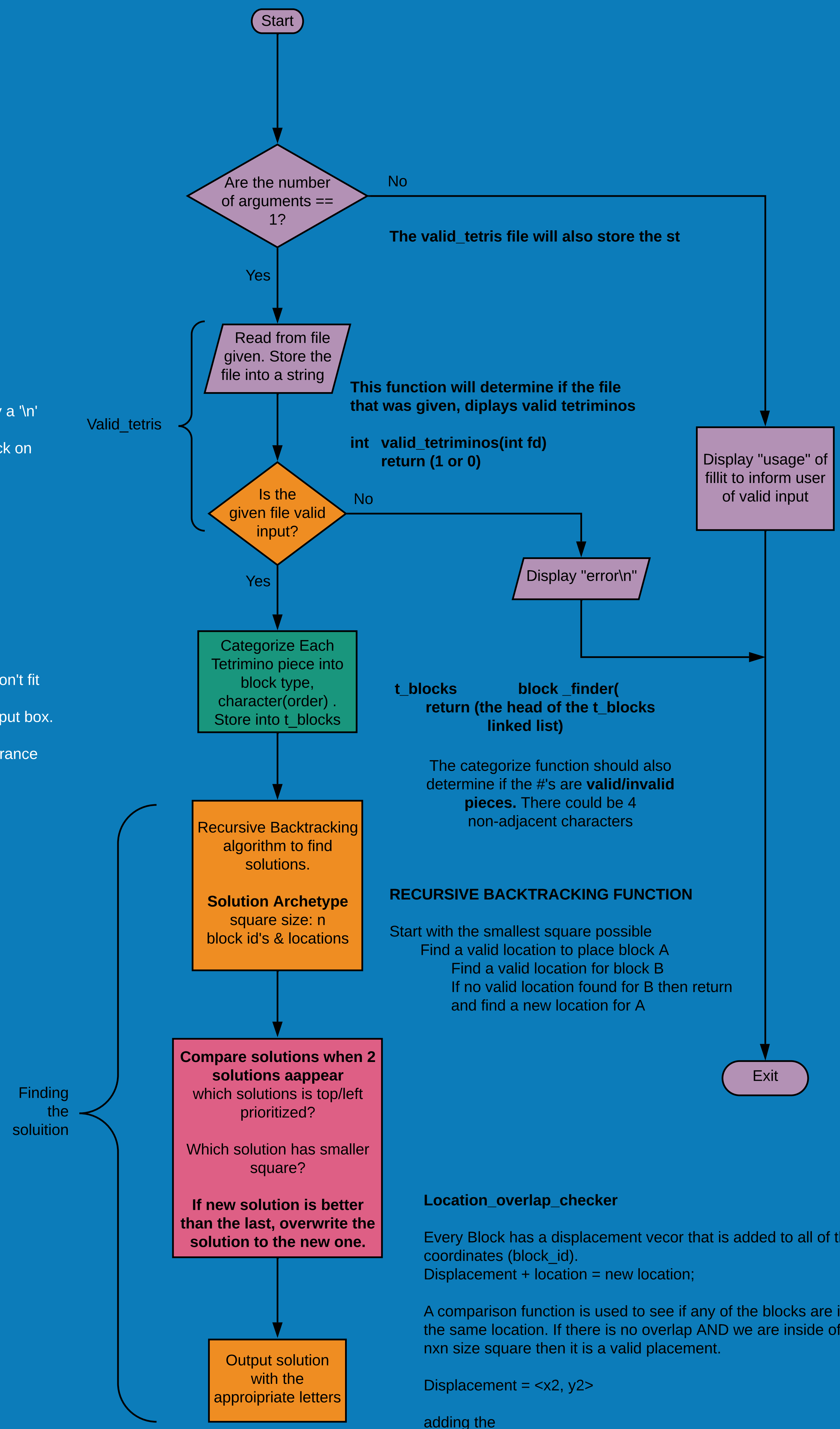
Overall programming stretegy
Using recursive backtracking
DONT ROTATE the tetriminos when finding solutions

Finding the smallest square
It's okay to have "holes" in the squares if the pieces don't fit together perfectly.
Peices must be assembled into the **top left** of the output box.
The "square" must contain all of the tetriminos given.
The Terminos must be ordered by the order of appearance given in the file { A, B, ..., Z}

```
typedef struct    s_blocks
{
      void                  *block_id;
      char                  c;
      struct s_blocks  *next;
}                     t_blocks

struct for the solution?
      void             *block_id;
      char             c;

      location      (
```



Start
Are the number of arguments == 1?
No
Yes
Read from file given. Store the file into a string
Valid_tetris
Is the given file valid input?
No
Yes
Display "error\n"
Display "usage" of fillit to inform user of valid input
Categorize Each Tetrimino piece into block type, character(order) . Store into t_blocks
Recursive Backtracking algorithm to find solutions.
Solution Archetype
square size: n
block id's & locations
Finding the solution
Compare solutions when 2 solutions aappear
which solutions is top/left prioritized?
Which solution has smaller square?
If new solution is better than the last, overwrite the solution to the new one.
Output solution with the appropriate letters
Exit

The valid_tetris file will also store the st

This function will determine if the file that was given, diplays valid tetriminos

int valid_tetriminos(int fd)
return (1 or 0)

t_blocks   block _finder(
return (the head of the t_blocks linked list)

The categorize function should also determine if the #'s are **valid/invalid pieces.** There could be 4 non-adjacent characters

RECURSIVE BACKTRACKING FUNCTION

```
Start with the smallest square possible
      Find a valid location to place block A
            Find a valid location for block B
            If no valid location found for B then return
            and find a new location for A
```

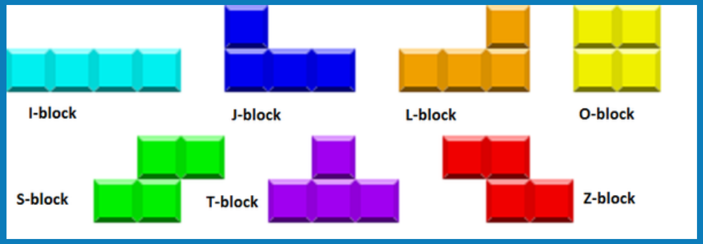
Location_overlap_checker

Every Block has a displacement vecor that is added to all of the coordinates (block_id).
Displacement + location = new location;

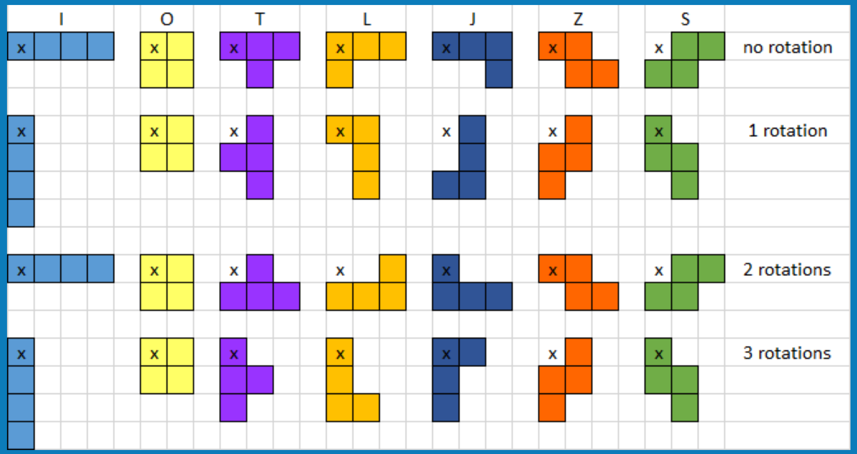
A comparison function is used to see if any of the blocks are in the same location. If there is no overlap AND we are inside of the nxn size square then it is a valid placement.

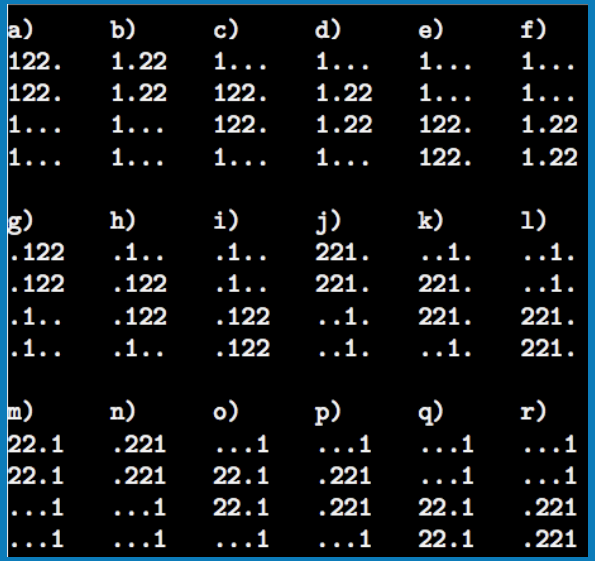
Displacement = <x2, y2>

adding the



Store the tetriminos into the structure relative to the coordinate system that is created. (top left X)





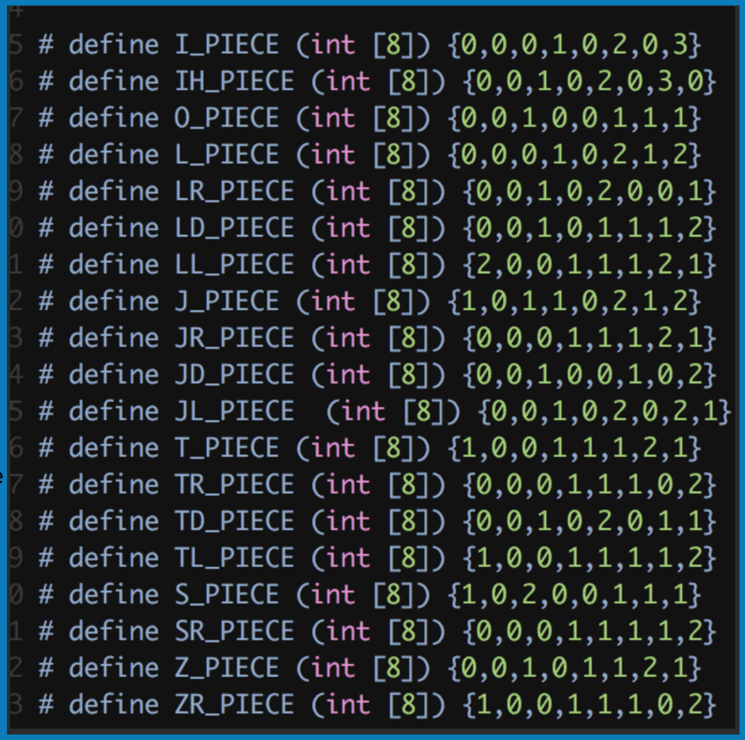
```
a)     b)     c)     d)     e)     f)
122.   1.22   1...   1...   1...   1...
122.   1.22   122.   1.22   1...   1...
1...   1...   122.   1.22   122.   1.22
1...   1...   1...   1...   122.   1.22

g)     h)     i)     j)     k)     l)
.122   .1..   .1..   221.   ..1.   ..1.
.122   .122   .1..   221.   221.   ..1.
.1..   .122   .122   ..1.   221.   221.
.1..   .1..   .122   ..1.   ..1.   221.

m)     n)     o)     p)     q)     r)
22.1   .221   ...1   ...1   ...1   ...1
22.1   .221   22.1   .221   ...1   ...1
...1   ...1   22.1   .221   22.1   .221
...1   ...1   ...1   ...1   22.1   .221
```

a > j
Because of the top left priority of the solution??

Medium article says that there are multiple solutions for the same input.



```c
# define I_PIECE (int [8]) {0,0,0,1,0,2,0,3}
# define IH_PIECE (int [8]) {0,0,1,0,2,0,3,0}
# define O_PIECE (int [8]) {0,0,1,0,0,1,1,1}
# define L_PIECE (int [8]) {0,0,0,1,0,2,1,2}
# define LR_PIECE (int [8]) {0,0,1,0,2,0,0,1}
# define LD_PIECE (int [8]) {0,0,1,0,1,1,1,2}
# define LL_PIECE (int [8]) {2,0,0,1,1,1,2,1}
# define J_PIECE (int [8]) {1,0,1,1,0,2,1,2}
# define JR_PIECE (int [8]) {0,0,0,1,1,1,2,1}
# define JD_PIECE (int [8]) {0,0,1,0,0,1,0,2}
# define JL_PIECE  (int [8]) {0,0,1,0,2,0,2,1}
# define T_PIECE (int [8]) {1,0,0,1,1,1,2,1}
# define TR_PIECE (int [8]) {0,0,0,1,1,1,0,2}
# define TD_PIECE (int [8]) {0,0,1,0,2,0,1,1}
# define TL_PIECE (int [8]) {1,0,0,1,1,1,1,2}
# define S_PIECE (int [8]) {1,0,2,0,0,1,1,1}
# define SR_PIECE (int [8]) {0,0,0,1,1,1,1,2}
# define Z_PIECE (int [8]) {0,0,1,0,1,1,2,1}
# define ZR_PIECE (int [8]) {1,0,0,1,1,1,0,2}
```