Thank you for taking the time to take this test. The problems are written in C++ or C#, but aside from problem 1 you can use any relatively mainstream language you like. You'll want to have your development environment(s) at hand, obviously. We've allotted six hours for this, but won't be tracking you to the exact second on the end time. If you're confident in your answers sooner than that feel free to send them sooner. ***You should send in answers to the core questions before attempting any of the "extra" problems.*** If you send something in early and follow up with "hey, I found a bug or thought of something new", we'll look at the later version without any penalty.

We'll be evaluating your answers on the following criteria, from most to least important:

1. Number of answers which work completely and correctly for all valid inputs
2. Number of answers which mostly work but have some bugs or edge cases
3. Style and readability of code
4. Performance and efficiency of working code
5. Partial credit for incomplete/non-working answers that point in the right direction. Please write a short bit with your thoughts on where you're going, how you're trying to get there, and what challenges remain.
6. Credit for speed if you finish everything in less time than allotted.
7. All of the above, for any of the "extra credit" problems.

In other words, as in real product development, it's better to succeed at being good than to fail at being great, and it's better to deliver core features on time than to ship late with more. If you have a choice between getting a simple robust solution done in less time, or implementing a faster but more complex algorithm that may have bugs or cut into your time for the other problems, then you should go for simple but working. You can always revisit the problem later if time allows.

All of the problems have an "Extras" section. **We don't expect (or want) anyone to do all of the Extras. Pick the one(s) that will give you the best chance to show off your unique strengths and the area(s) that make you special.**

As a general note, because we're recruiting for a limited number of spots it won't be in your interest to discuss the test with others.

Problem 1

Note: If you're applying for the web/UI/UX developer position, we don't expect you to know or use C++. You can skip this problem entirely without penalty. If you really want to do this we'll treat it as "extra credit", but please don't try to Google your way to C++ knowledge in an afternoon.

Implement a simple class in C++ to encapsulate a two-dimensional array of integers. The constructor should take two integers specifying the initial dimensions, and there should be a way to get or set any element of the array. Provide a `Resize(int sizeX, int sizeY)` method that grows or shrinks the array. The class must clean up all memory it allocates and safely handle attempts to make out-of-bounds accesses or set negative sizes.

You should use standard, unmanaged, non-garbage-collected C++. You should implement all key functions yourself without using the STL. (In other words, you'll need to call new or malloc, but you can't just wrap a std::vector.) Error reporting and external test code may use any standard libraries you like. This *must* be in C++; it's specifically a test of your ability to manually manage memory and pointers without the benefit of a garbage collector.

Extras, Problem 1

Worry about this only if you're completely sure of your success (or failure) at the core parts of all the problems! If you choose to do this, get a working answer to the core question first and submit that separately before you play with this one.

1A) Implement operator overloading for this class. Given an instance of this class, you should be able to get/set elements like this:

```
Array2D arr(5, 7);
arr[3][3] = 9;
```

You *do NOT* need to handle bounds checking. So this code...

```
Array2D arr(5, 7);
arr[3][8] = 9;        // error, 8 is out of bounds
```

...is an error, but one that is allowed to crash, fail, or otherwise show undefined behavior in your implementation. Your accessors can be row-major or column-major, your choice.

1B) As in 1A, but with bounds checking. The incorrect example should detect and somehow handle the error. (You can throw an exception, quietly redirect the write someplace safe, or other approaches.)

Problem 2

Suppose you have a singly-linked list of integers. In C++, this would be represented by:

```
struct Node { int val; Node * next; };
```

In C# or Java, where Node is implicitly a reference type, it might be declared more like this:

```
class Node { public int val; public Node next; };
```

A list is represented by a pointer to the head `Node`. The end of the list is represented by a `Node` with a null `next` field. An empty list is valid, and is represented by a null head pointer. A sample list would look like:

```
headPtr--> ( 4 )--> ( 7 )--> ( 6 )--> ( 2 )-->NULL
```

Write a function to sort the list. By "sort", you must choose the node with the smallest value to be the head node, and rearrange the "next" pointers of each node to reference the node with the next highest value: For the sample list, the result would look like:

```
headPtr--> ( 2 )--> ( 4 )--> ( 6 )--> ( 7 )-->NULL
```

Your function should take a pointer to the unsorted head node as input, and return as output a pointer to the node that became the head after sorting. You must modify the `next` pointers of the Nodes in place; do not modify the values within the nodes. Do not allocate extra memory other than local variables on the stack: In particular, do not allocate an array, put all the nodes into it, and sort the array. You may write and call any additional sub-functions as needed, but you cannot use your language's container types or built-in sorting functions. You may use any mainstream language you like, as long as it supports a nullable reference type for the next and head pointers. In JavaScript, a general test harness might look like this:

```
// example linked list. Could also build this as:
//  var headNode = { val:4,
//                   next:{ val:7,
//                          next:{ val:6,
//                                 next: { val: 2,
//                                         next:null } } }
var lastNode = { val : 2, next: null };
var thirdNode = { val : 6, next : lastNode };
var secondNode = { val : 7, next : thirdNode };
var headNode = { val : 4, next : secondNode };
```

```
// function to walk through the nodes of a linked list
//   and add them to an HTML <ul>
function writeList( headNode, parentElementID ) {
    var n = headNode;
    var parentElement = document.getElementById(parentElementID);
    while (n) {
        var listItem = document.createElement('li');
        listItemText = document.createTextNode('Val: '+ n.val);
        listItem.appendChild(listItemText);
        parentElement.appendChild(listItem);
        n = n.next;
    }
}

// Test out the sort function, showing the linked list
// before and after
writeList(headNode,'problem2Input');
var newHeadNode = sortList(headNode);
writeList(newHeadNode,'problem2Output');
```

Extras, Problem 2

Worry about this only if you're completely sure of your success (or failure) at the core parts of all the problems! If you choose to do this, get a working answer to the core question first and submit that separately before you play with this.

2A) Tell us about the average running time of your function, in terms of the number of items in a random list. (i.e. If there are N items in the list, does it take N or N^2 operations to sort? Will a list that's 10 times as long take 10, 20, 100, or 1000 times as long to sort?) You can ignore effects from being over/under your particular CPU's cache size.

2B) If you didn't already, write a version that, without allocating memory, sorts a singly-linked list in at most O(N * log N) time.

Problem 3

Your task here is to mark filled circles in a 2D array. This could be used for drawing a picture, marking cells covered by fog of war, or determining the area hit by an explosion. You're working in a two-dimensional array with one byte per cell. This array will always have 100x100 elements (most languages will number them 0..99). You need to write a function which, given a floating-point center position and radius, will set all points within that radius to a provided value. In C++, your function would look like this:

```
void FillCircle( unsigned char cells[100][100],
                 float xCenter,
                 float yCenter,
                 float radius,
                 unsigned char markValue );
```

You should assume your array is in row-major (i.e. cells[y][x]) order, and that the array coordinates represent a point exactly at the position of the array indices. Points exactly on the circle boundary should be included in the circle. So if someone called `FillCircle(cells, 5.25f, 20.0f, 2.25f, 150)` then you *would* set cells[20][3] to 150 because (3.0,20.0) is exactly 2.25 units away from (5.25, 20.0). In that row you would also set cells[20][4] through cells[20][7] to 150 because they are within the circle. You would *not* change cells[20][8], because (8.0,20.0) is 2.75 units away from the center, which is outside the circle.

You can implement whatever subfunctions you want to. You can make use of your language's core math library for functions like square roots, exponents, trigonometry, etc. (But if you find a language with "draw a circle" in its library, that function is off-limits!)

Circles that extend outside the array should be clipped cleanly. The parts that are within the array should be marked, and the parts that are outside the array should be silently ignored. FillCircle(image, 99, 99, 49.5, 27) should mark a quarter-circle centered on one corner. FillCircle(image, 1000, 40, 40, 5) should safely do nothing.

If you're working in a language that doesn't have good support for bytes (JavaScript, etc.), you can substitute any other numeric type.

Extras, Problem 3

Worry about these only if you're completely sure of your success (or failure) at the core parts of all the problems! These are three separate challenges, and you can take on any or all of them. Get a working answer to the core question first and submit that separately before you play with these. We suggest you submit a separate answer for each of these you attempt, rather than trying to do all of them in one change.

3A) How many mathematical operations of each type are you using? Tell us in a short

paragraph, and describe how that number scales up as you increase the radius.

Now, can you write a version that uses fewer operations? Show us how fast you can make this thing go! Can you optimize it even further by writing more than one cell at a time, or even using SSE or other vector operations? We'll give you credit for explaining how, and more credit for showing us running code.

3B) Let's add anti-aliasing. Assume each cell is a box centered on its array indices. So image[4][9] represents a 1x1 box whose boundaries range from 8.5 to 9.5 in the X direction, and 3.5 to 4.5 in the Y direction. Instead of completely setting or not setting each pixel, you should blend each cell from its current value to the desired value based on what percentage of the pixel's box overlaps the circle.

If someone called FillCircle(image, 5, 10, 3, 100), you'd still set image[10][5] to 100 because it's completely within the circle. But now you'd blend image[10][2] to a value about halfway between its current value and 100, because the edge of the circle runs through the center of that pixel's box. You'd still leave image[10][1] alone, because its box is completely outside the circle.

The blend amount can be an approximation of the exact analytical value; the only thing that matters here is to run well and please the eye. What are the tradeoffs with the method you've chosen?

You can change the cells array and markValue to floating point for this if you prefer. Assume you're working with linear, non-sRGB intensities.

3C) Instead of drawing flat circles, let's draw illuminated spheres. We'll treat the array as an orthographic view of a 3D world. Your X and Y coordinates are the same pixel indices we've been using, but now there's a Z coordinate which increases in a positive direction as we move out towards the viewer. There's an infinitely far light source shining in from the (+1,+1,+1) direction, and everything else is darkness.

Draw the circles as if they're spheres whose Z center coordinate is 0, using the lighting model of your choice to make them pretty. Ignore any depth testing or intersection with other spheres; you can just overdraw whatever pixels are in the image.

You can change the cells array and markValue to floating point for this if you prefer. (Your lighting model will presumably interpret markValue as some sort of albedo or reflectance.) Assume you're working with linear, non-sRGB intensities.

Problem 4

Every classic RPG had a dungeon. It's a tradition as old as pencils and Cheeto-stained graph paper. For this problem, you'll be working with a 2D maze on a **22x22 square grid**. Every grid square is either completely filled or completely empty, and is represented by a 2D **row-major** array of integers telling you whether the cell is **walkable (1)** or **blocked (0)**. You may assume that the maze is sealed; that is, all the cells along the four edges (those with an X or Y coordinate of 0 or 21) will be set to zero. You can rely on this, and you do not need to check for it in your code. You can move in the four cardinal directions (up, down, left, and right) between any pair of adjacent walkable cells, but you cannot move diagonally or through blocked cells. The maze is not necessarily connected; there may be walkable cells that can never be reached from certain other walkable cells. A smaller 9x7 maze would be declared like this in C#:

```
var mazeDef = new int[7,9] { { 0,0,0,0,0,0,0,0,0 },
                             { 0,1,1,1,1,1,1,0,0 },
                             { 0,0,0,0,0,0,1,1,0 },
                             { 0,1,1,1,1,1,1,0,0 },
                             { 0,1,0,1,0,0,1,0,0 },
                             { 0,1,0,1,1,1,1,0,0 },
                             { 0,0,0,0,0,0,0,0,0 } };
```

Your task is to write a function that will take as parameters a 22x22 2D array and (X,Y) coordinates for a start and end cell. You must return a single integer indicating the **number of cells in the longest possible non-overlapping path** through the two points, or else **-1 if no such path is possible**. In C# your function will look something like this:

```
int longestPath(int[,] maze, int startX, int startY,
                             int endX, int endY);
```

If you were working with the example maze above instead of a full 22x22 maze, some sample inputs and their expected outputs would be:

longestPath(mazeDef, 1,1, 6,1) == 6 *(straight across the top corridor)*
longestPath(mazeDef, 1,1, 1,2) == -1 *(can't reach a blocked cell)*
longestPath(mazeDef, 3,3, 4,3) == 10 *(go around the loop through 6,5 in the bottom right)*
longestPath(mazeDef, 3,3, 3,3) == 1 *(can't go around the loop because we'd overlap our path at the end)*

You may write any subfunctions, declare any data structures, and allocate any memory that you like. You may use system library functions freely, including the C++ STL, C#'s System.Collections.Generic and LINQ, or any other container and/or algorithm classes that come with the latest version of your language. You may destructively modify the passed-in maze data if you want.

Extras, Problem 4

Worry about these only if you're completely sure of your success (or failure) at the core parts of all the problems! These are separate challenges, and you can take on any or all of them. Get a working answer to the core question first and submit that separately before you play with these.

4A) Write a function to find the **shortest** path between two squares in a 22x22 maze. Are there different algorithms or data structures that will let you do this more efficiently than finding the longest path? Use them where possible.

4B) As mentioned, the maze may not be fully connected. There may be walkable squares that are not connected to each other by any path. Write a function that will take a 22x22 maze and return the number of disconnected walkable areas. In the small sample maze above, there is one area because everything is connected. However, if we changed the single cell at 6,2 from walkable to blocked, there would be three areas: One for the top hallway, one for the single room on the far right, and one for the bottom area.

```
var disconnectedMazeDef = new int[7,9]
                          { { 0,0,0,0,0,0,0,0,0 },
                            { 0,1,1,1,1,1,1,0,0 },
                            { 0,0,0,0,0,0,0,1,0 },
                            { 0,1,1,1,1,1,1,0,0 },
                            { 0,1,0,1,0,0,1,0,0 },
                            { 0,1,0,1,1,1,1,0,0 },
                            { 0,0,0,0,0,0,0,0,0 } };
```