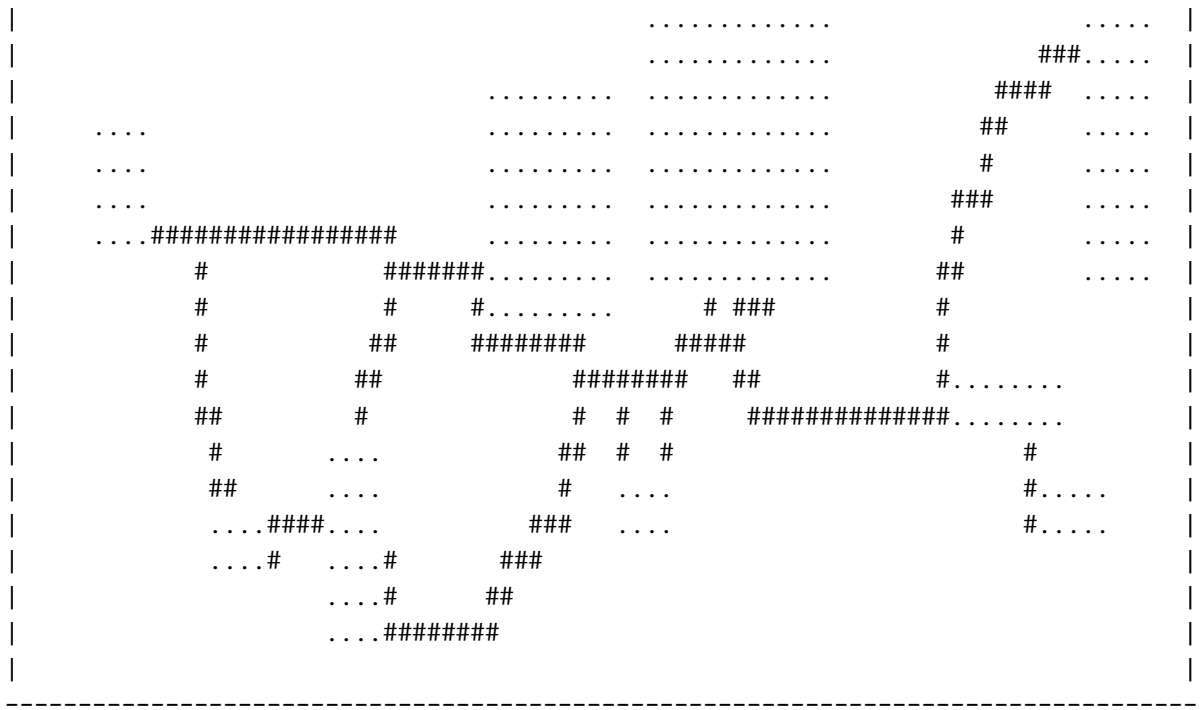
COM S 327, Fall 2016
Programming Project 1.01
Dungeon Generation

It's time to start developing our Roguelike game. We'll begin by generating a dungeon. Some details are specified below, but generating a good-looking dungeon is largely a matter of taste. The figure shows an example generated by my game.

Because it's so much a matter of taste, you may decide how you want your dungeon too look, and come up with your own heuristics to generate that look. Things to keep in mind are playability and realism. Yeah, it's a fantasy game where we fight monsters in underground dungeons, but that doesn't mean there's no realism! Corridors that are too straight are probably unrealistic; after all, veins of harder minerals occur in the rock, and miners will probably tunnel around them. On a more practical note, large open areas, wide corridors, and overly straight corridors give advantages to monsters in play because the player does not have architectural features to use to tactical advantage. A poor dungeon leads to an unplayable game. Players need places to rest and hide out of sight of fire-breathing dragons and balrogs, spell-casting necromancers, and arrow-loosing orc hordes. On the other hand, sometimes line of sight is useful for your own attacks, so you don't want your passages too twisty.

Your dungeon generator should be written with an eye toward extensibility. Remember that you will be adding functionality on top of this next week, and for the rest of the semester. For this step, you generate a dungeon, draw it on the standard output, and exit. Here are the requirements:



An example randomly-generated dungeon from my code. Rock is represented by spaces, room floor by periods, and corridor floor by hashes. The border was added artificially to demarcate the map from the page.

- All code is in C.
- Dungeon measures 80 units in the x (horizontal) direction and 21 units in the y (vertical) direction. A standard terminal is 80×24, and limiting the dungeon to 21 rows leaves three rows for text, things like gameplay messages and player status.
- Require at least 6 rooms per dungeon
- Each room measures at least 4 units in the x direction and at least 3 units in the y direction.
- Rooms need not be rectangular, but neither may they contact one another. There must be at least 1 cell of non-room between any two different rooms.
- The outermost cells of the dungeon are immutable, thus they must remain rock and cannot be part of any room or corridor.
- Room cells should be drawn with periods, corridor cells with hashes, and rock with spaces.
- The dungeon should be fully connected.
- Corridors should not extend into rooms, e.g., no hashes should be rendered inside rooms.

Here is an informal description of a dungeon generator that I wrote to produce the figure; I've played with dungeon generation enough to know that much simpler methods can do the job, but this method does a nice job of balancing aesthetically-pleasing (to me) dungeons and having a small, straightforward implementation. There are certain tunable parameters, for example, determining the size of a new room, that I leave out of the description. You may use this algorithm, something you find online, or something of your own devising.

To implement my algorithm, you will need an array of rooms and an 80×21 matrix of cells representing the dungeon. I initialize the dungeon by setting an immutable flag on the outermost cells and assigning a hardness to the material in every cell. I then attempt to randomly place random rooms in the available space, checking that the room can be placed legally each time, until some termination criterion is reached. Example criteria: the dungeon is at least 7% open; there were 2000 failed placement attempts in a row; a `create_new_room_p()` predicate failed; etc.

After placing rooms, move through the room array of n rooms, connecting room 1 with room 2, then room 3 with rooms 1–2, ... until you've connected room n with rooms 1–$(n-1)$. Okay, so how do we make that connection? Find the closest room in the already connected set using Euclidean distance to its centroid then carve a path to it by changing rock to open space; this can always be done with zero or one change of direction. If you get that working, then add some random changes of direction in there to make it look a little more exciting.

So why do I have that rock hardness? Well, our adventurer and monsters will eventually be able to tunnel through the dungeon, so hardness will be a useful notion to influence tunneling, but for now I use it to drive a corridor heuristic. I use Dijkstra's algorithm where my edge weights are rock hardnesses and modified with penalties for direction changes to find shortest paths between 1–2, 2–3, ..., $(n-1)$–n. If you implement this, you'll find that your code will draw corridors directly to nearby rooms too often—the dungeon ends up connected, but there are never any loops, and I think that's kind of boring—so you need to weight empty space close to the hardness of the softest rock to get more satisfying results. You may be thinking that this is an interesting method, but it requires a priority queue, so it's too much work. Maybe that's true—up to you to decide—however: 1) A priority queue is easy to implement; and 2) You'll need one eventually, anyway, in order for monsters to move around the dungeon without getting stuck in corners.