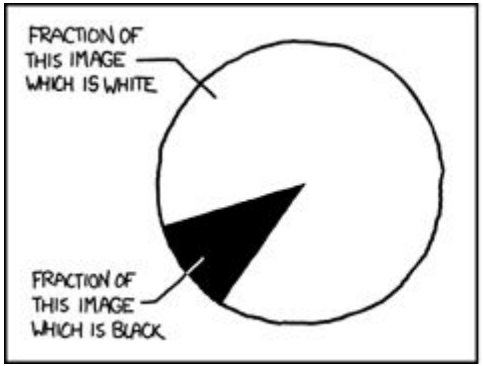
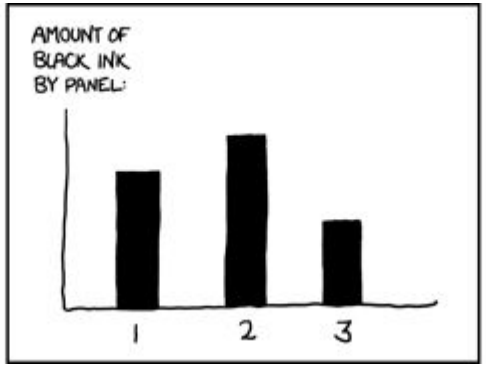
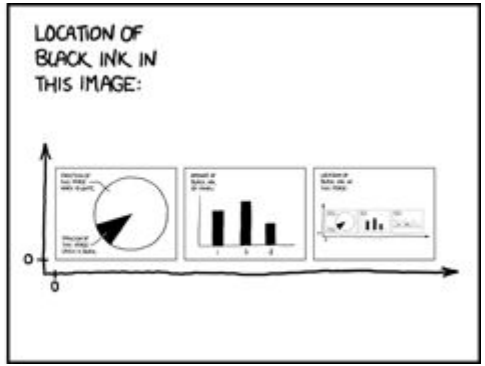
YEAH - Recursion!

Anton Apostolatos



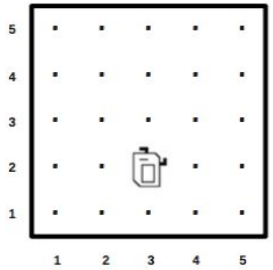




Source: XKCD

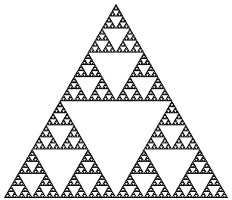
A3: Recursion!

Karel Goes Home

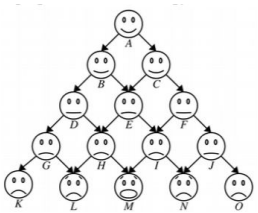


Subsequences

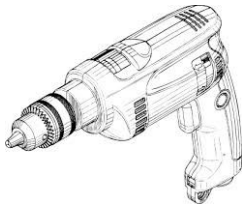
Sierpinksi



Human Pyramids



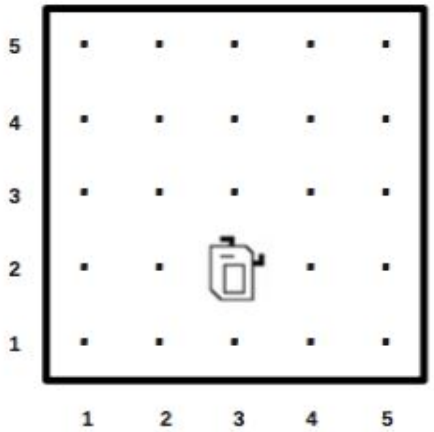
Drill, Baby, Drill!

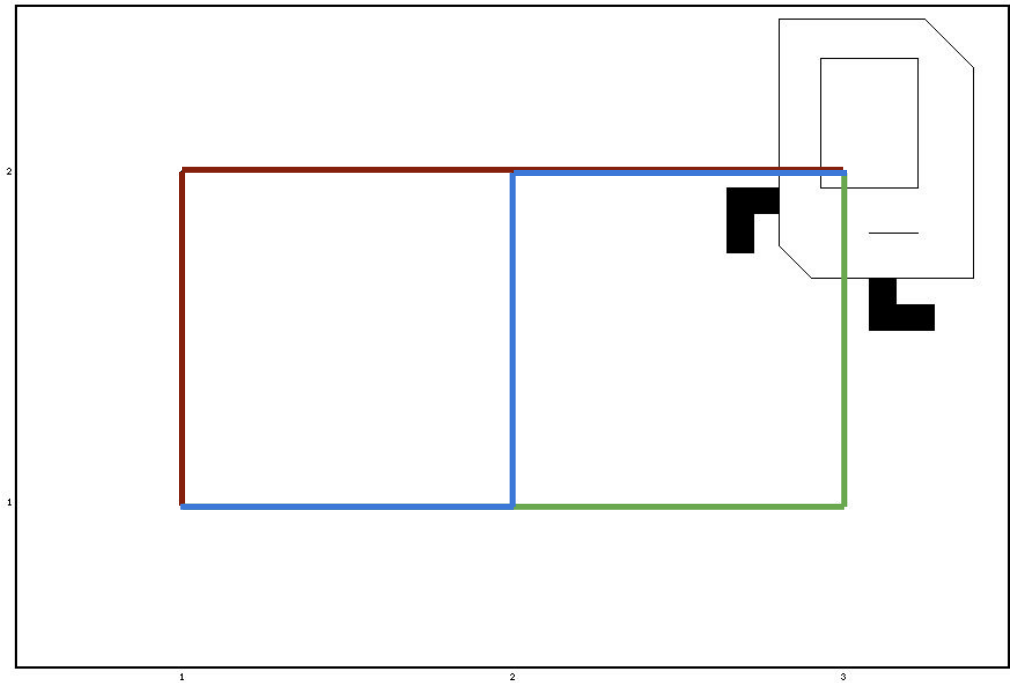


Universal Health Care

```
if (problem is sufficiently simple) {
   Directly solve the problem.
   Return the solution.
 } else {
   Split the problem up into one or more smaller
      problems with the same structure as the original.
   Solve each of those smaller problems.
   Combine the results to get the overall solution.
   Return the overall solution.
}
```

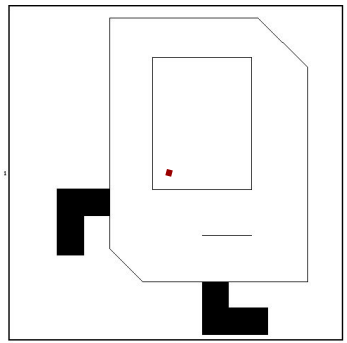
Karel Goes Home

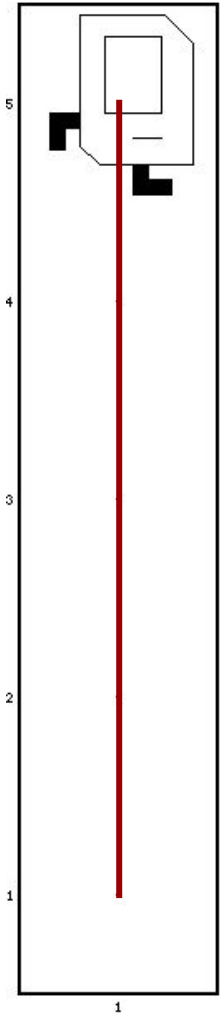




3 paths!

1 path!





1 path!

Write the recursive function

```
int numPathsHome(int street, int avenue)
```

returns the number of shortest paths Karel could take back to the origin from the specified starting position

***Note**: Karel wants the shortest path, so she should only move south or west!*

Subsequences

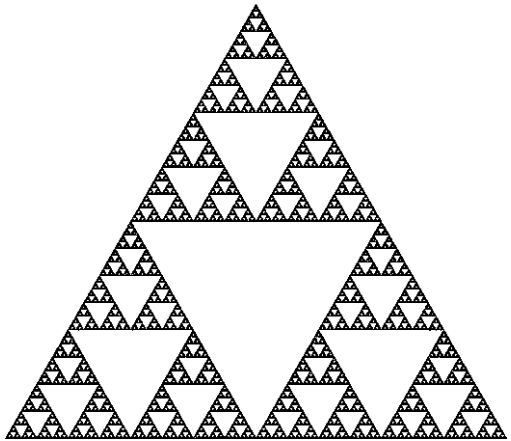
Write the recursive function

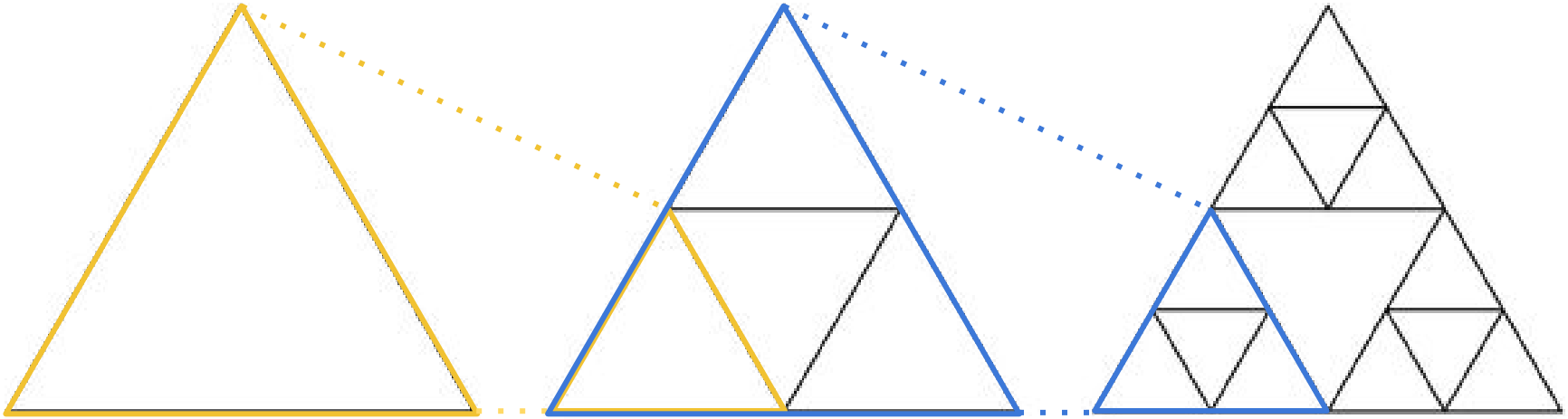
```
bool hasSubsequence(string text, string subseq)
```

returns whether the second string is a subsequence of the first

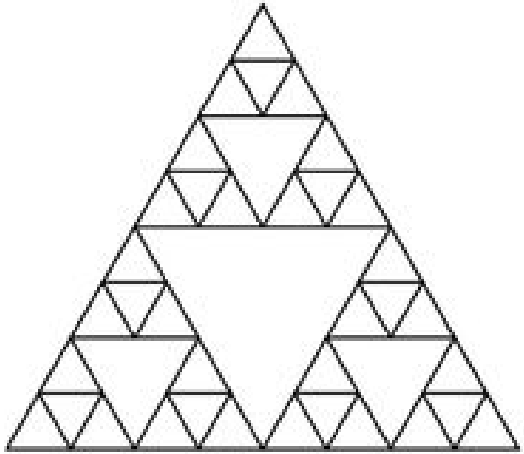
```
 hasSubsequence("abcde", "bd")                    →  true
 hasSubsequence("I love the water", "I hate")     →  true
hasSubsequence("", "bd")                          →  false
hasSubsequence("I AM THE ALPHA AND OMEGA", "man") →  false
```

Sierpinski

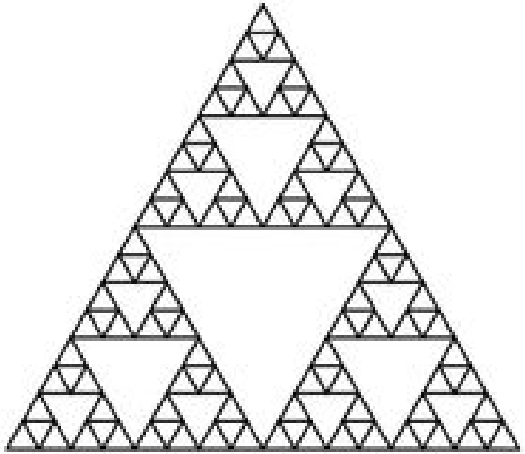




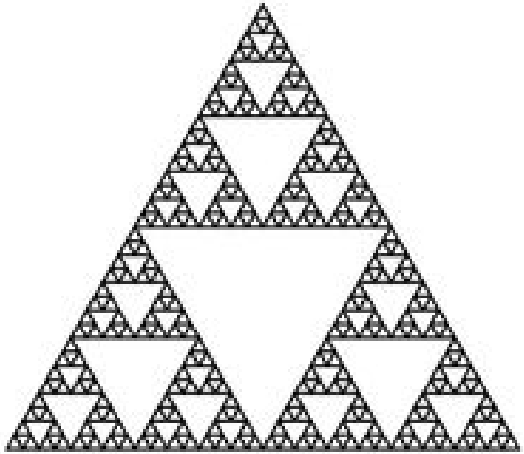
Order 0 Order 1 Order 2



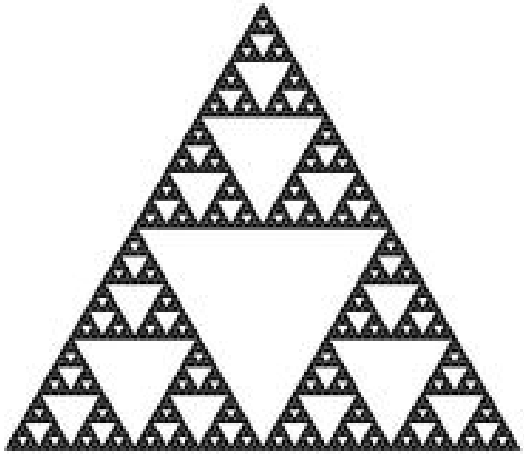
Order 3



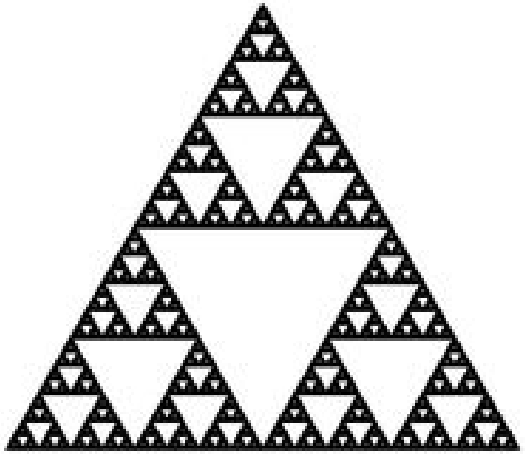
Order 4



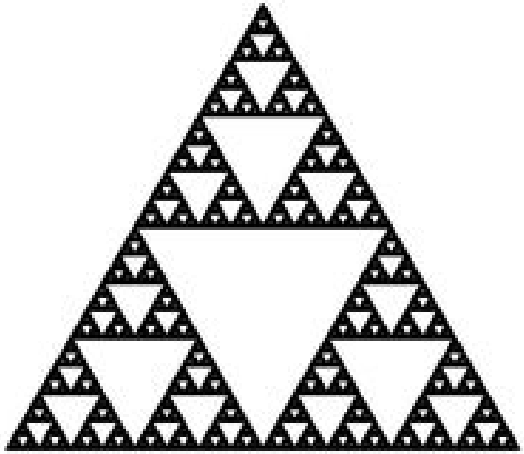
Order 5



Order 6



Order 7



Order 8

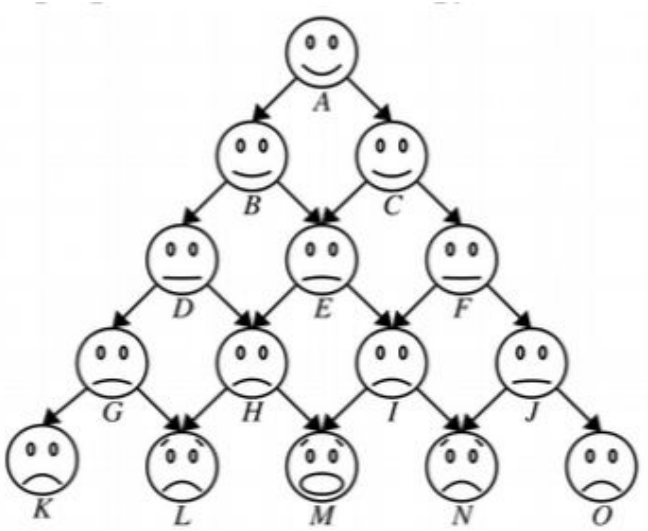
Write the recursive function

```
void drawSierpinskiTriangle(GWindow& window,
   double x, double y, double sideLength, int order)
```

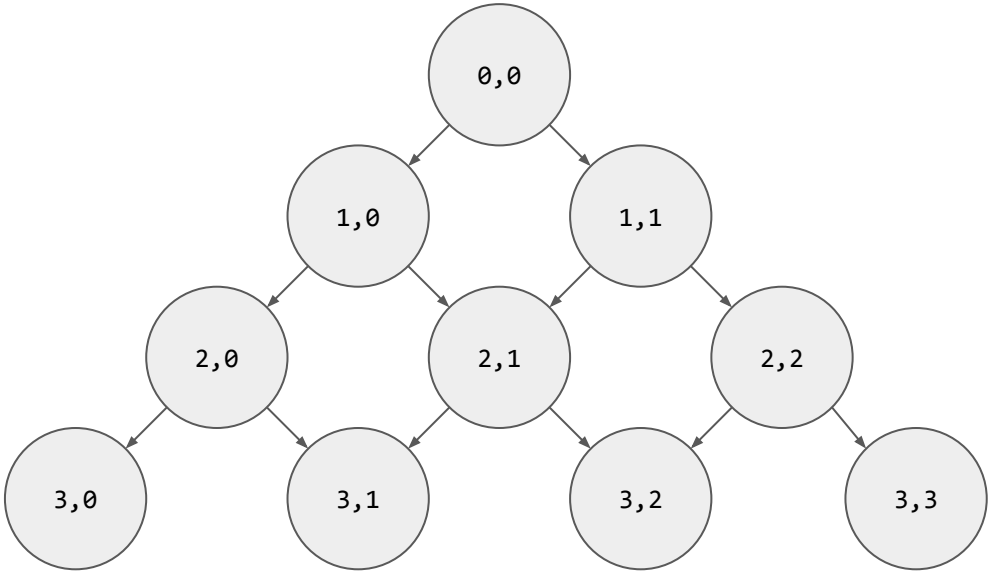
window: where to draw the triangle (see C++ docs!)
(x, y): bottom-left corner of the triangle
sideLength: length of triangle side
order: the order of the triangle to draw

***Note**: using **drawPolarLine** will make your life easier!*

Human Pyramids

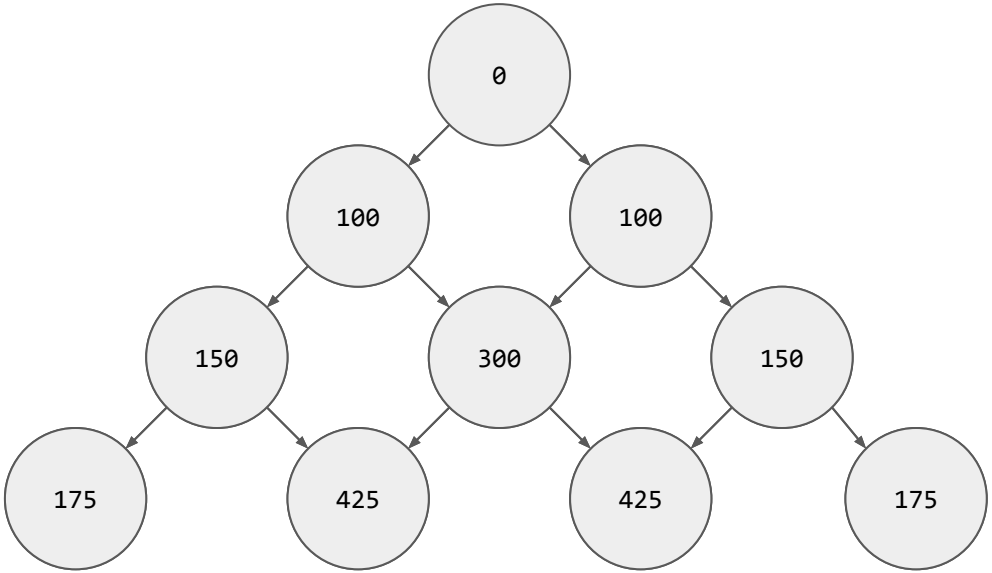


Consider a human pyramid:



Consider a human pyramid, **where every person weighs 200lbs**

What's the weight on a certain person's knees?



Write the recursive function

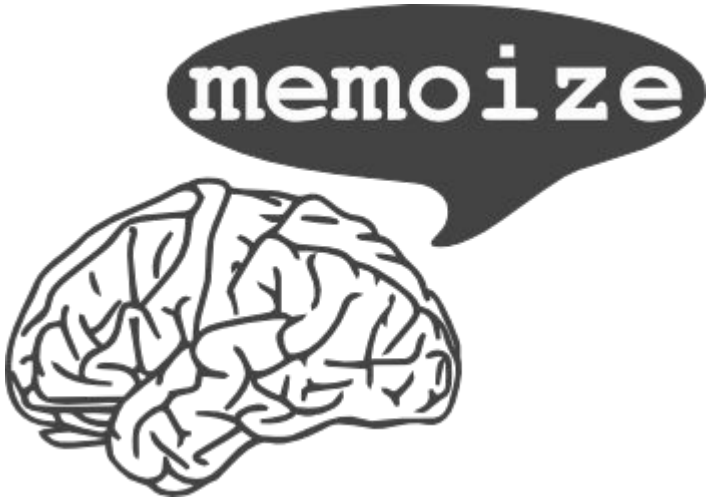
```
double weightOnBackOf(int row, int col)
```

(row, col): row and col of person we're interested in

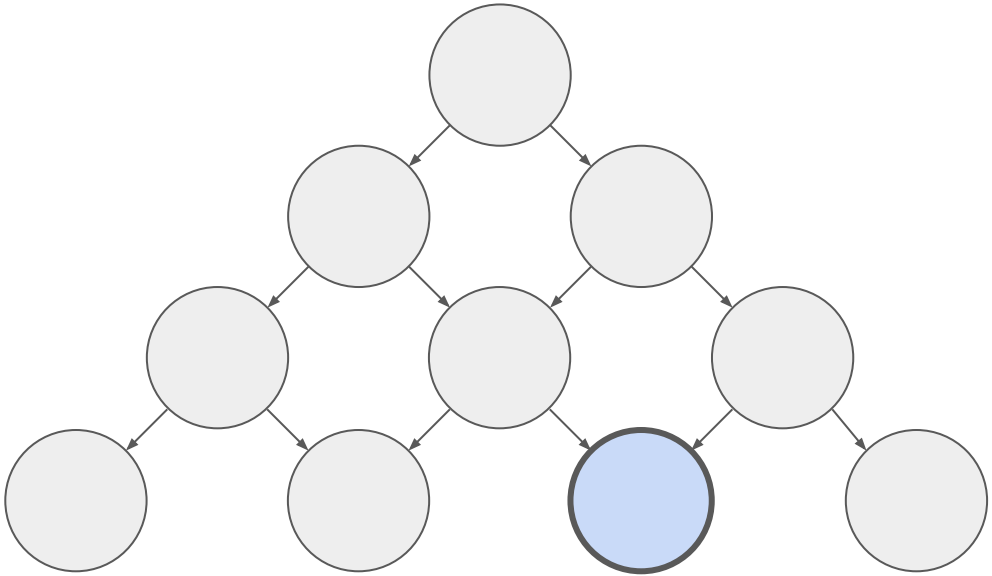
Note:

- *We only care about the weight on their back, without their own weight.*
- *Consider edge cases (e.g. negative rows or cols)!*

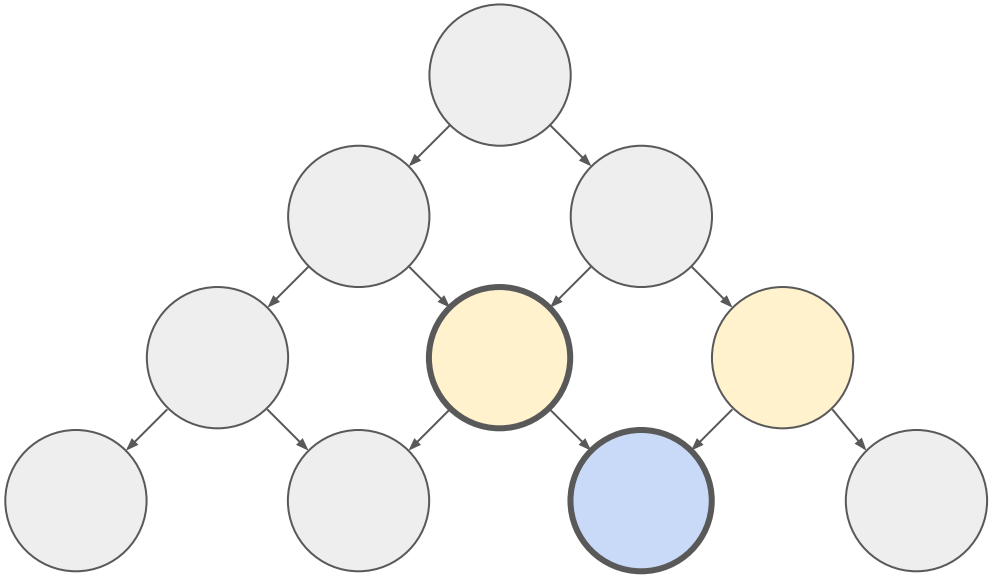
Memoized Human Pyramids



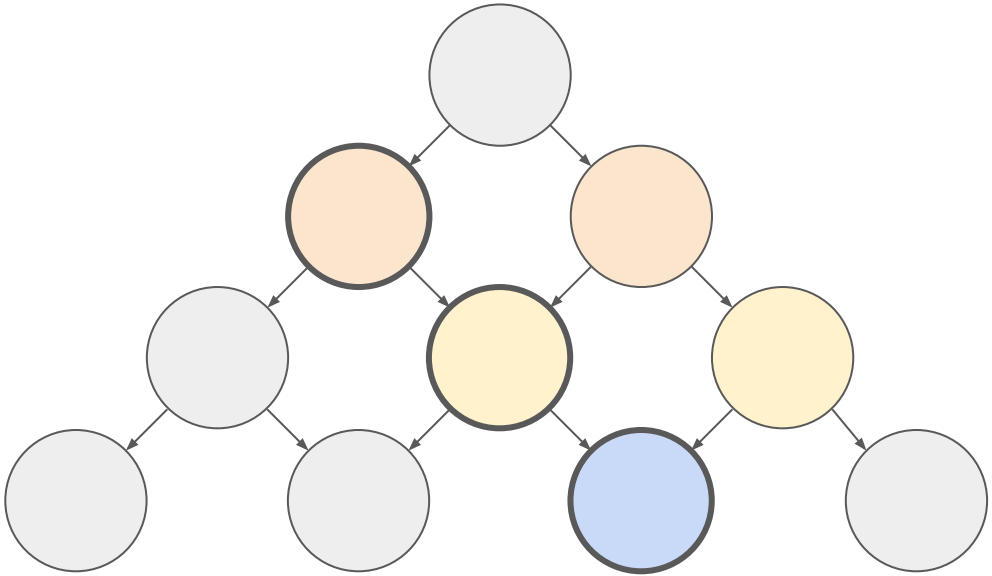
Consider wanting the weight on the back of **(3,2)**



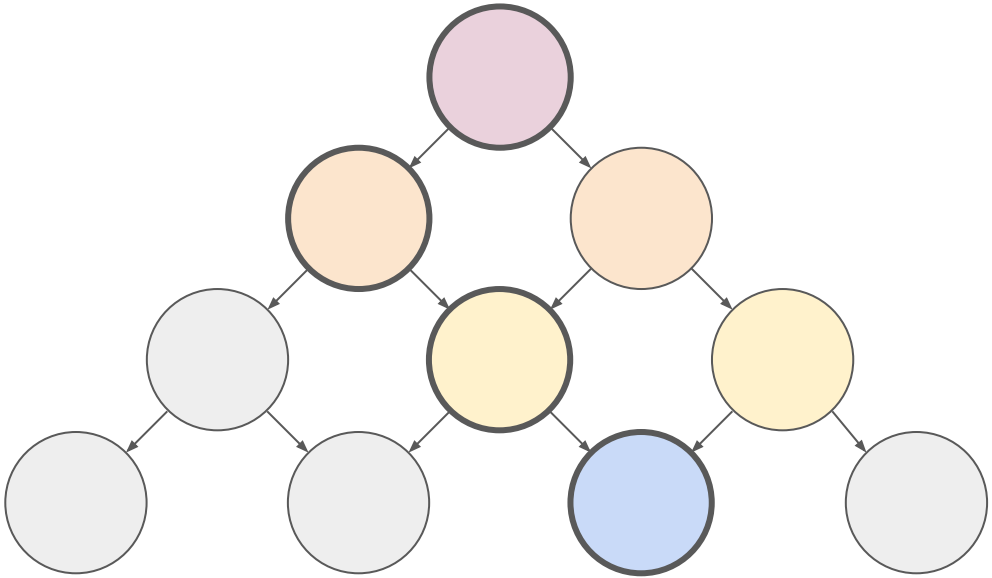
Consider wanting the weight on the back of **(3,2)**



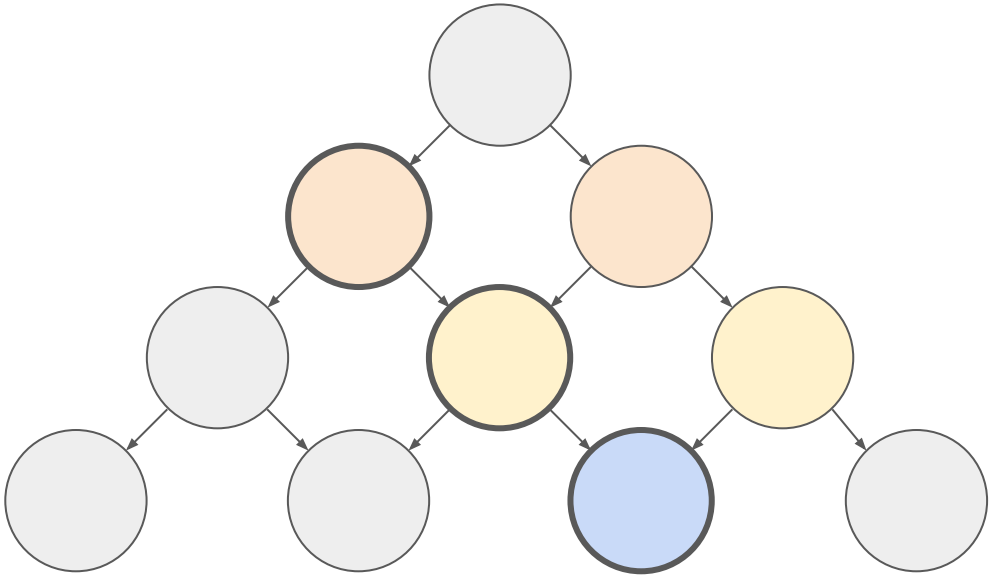
Consider wanting the weight on the back of **(3,2)**



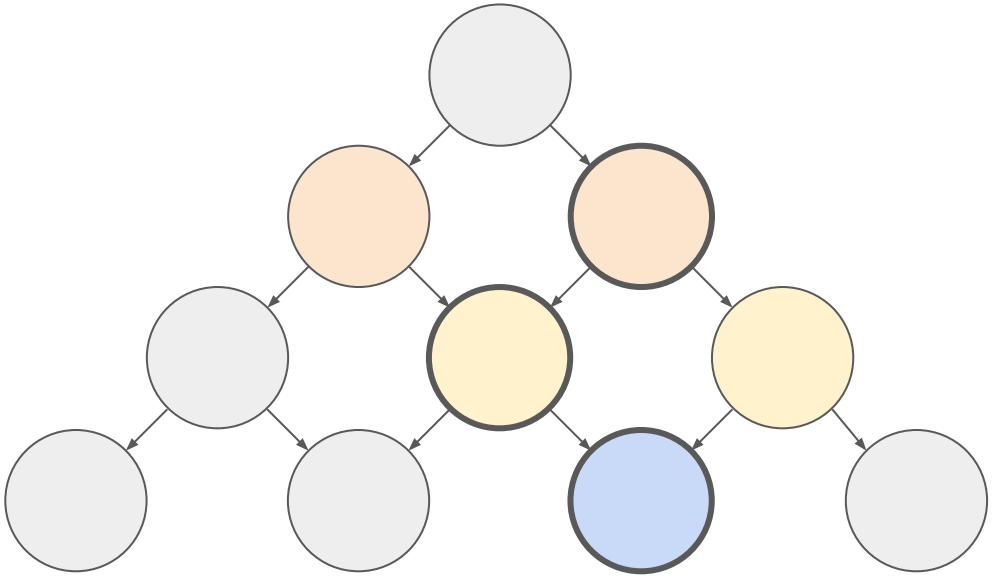
Consider wanting the weight on the back of **(3,2)**



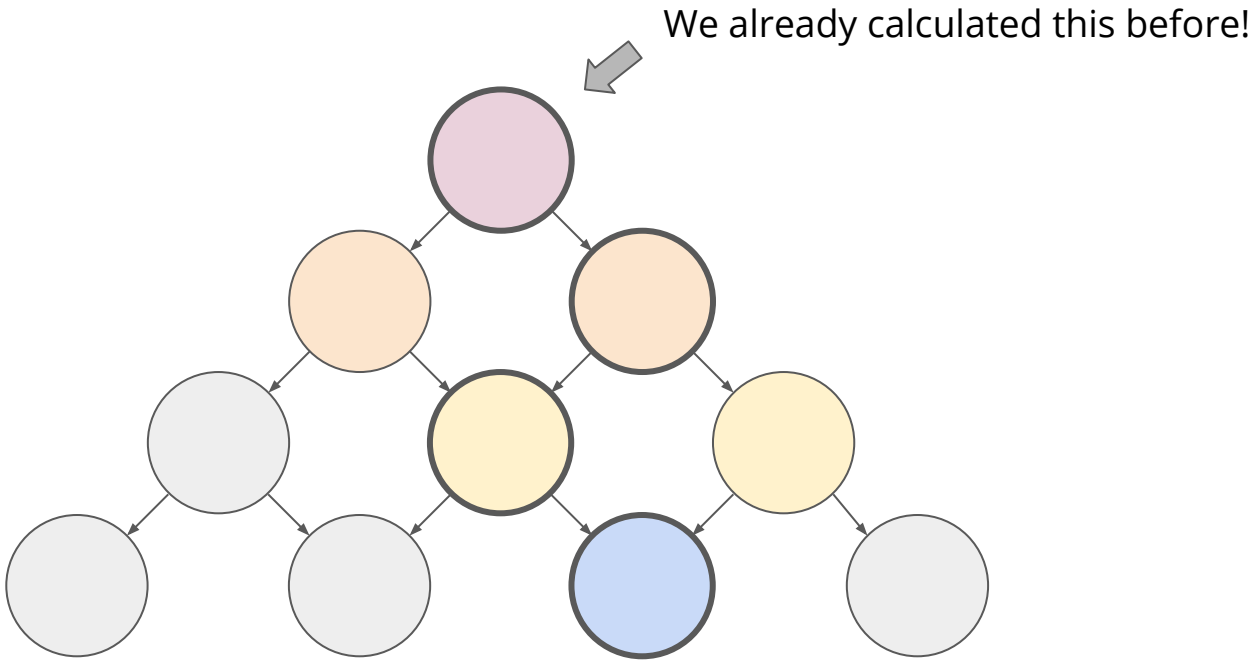
Consider wanting the weight on the back of **(3,2)**



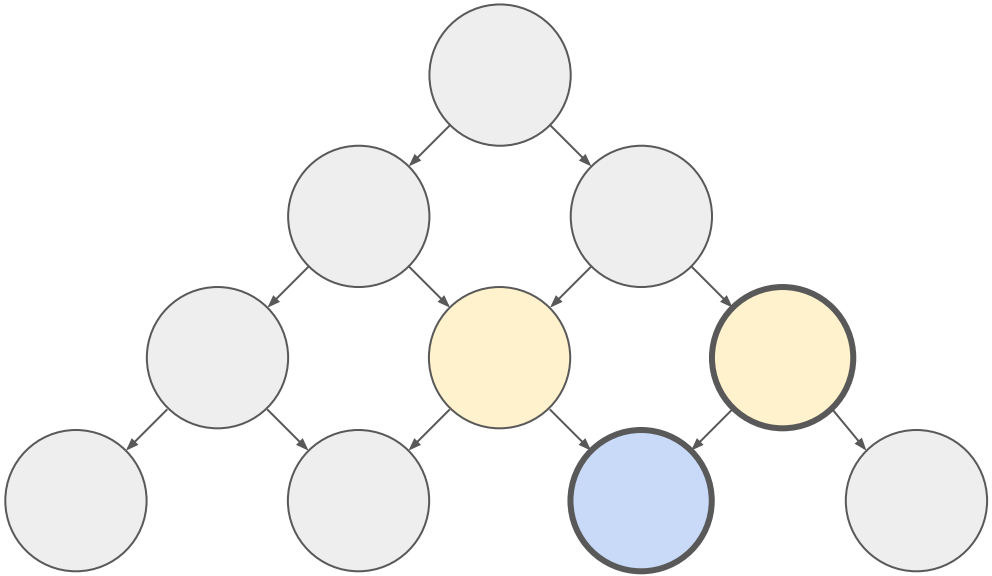
Consider wanting the weight on the back of **(3,2)**



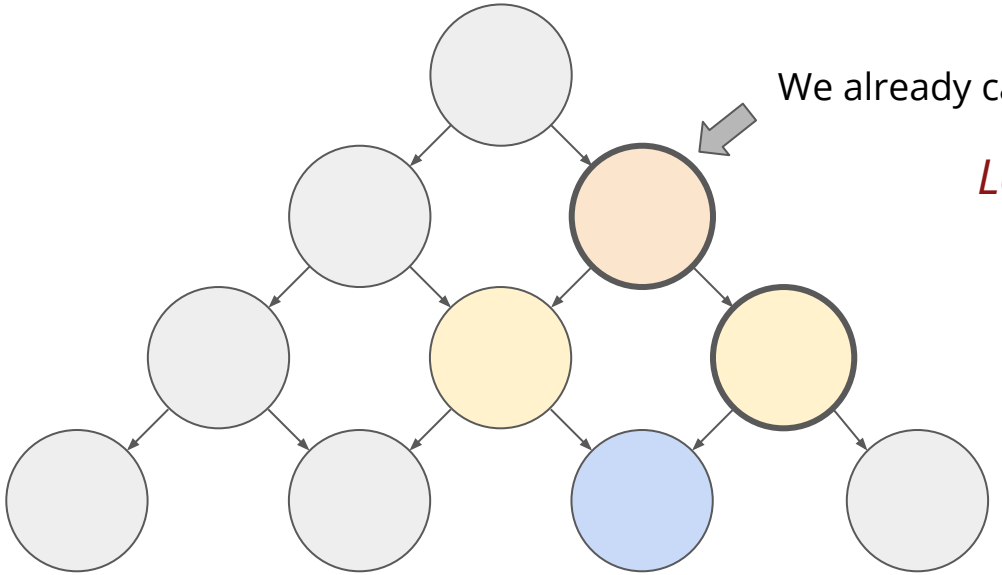
Consider wanting the weight on the back of **(3,2)**



Consider wanting the weight on the back of **(3,2)**



Consider wanting the weight on the back of **(3,2)**



We already calculated this before!

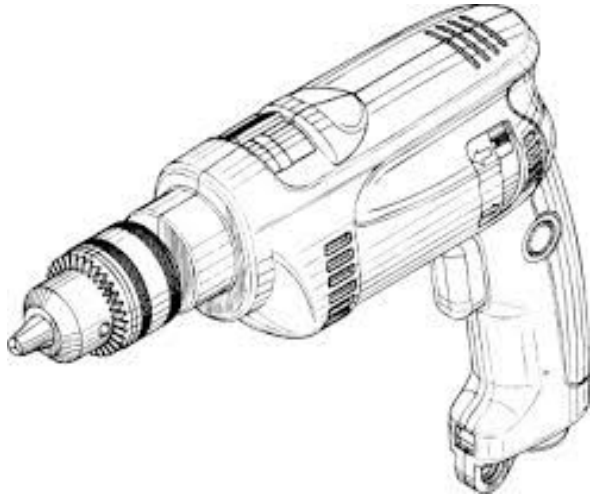
Let's store it!

Memoization speeds things up!

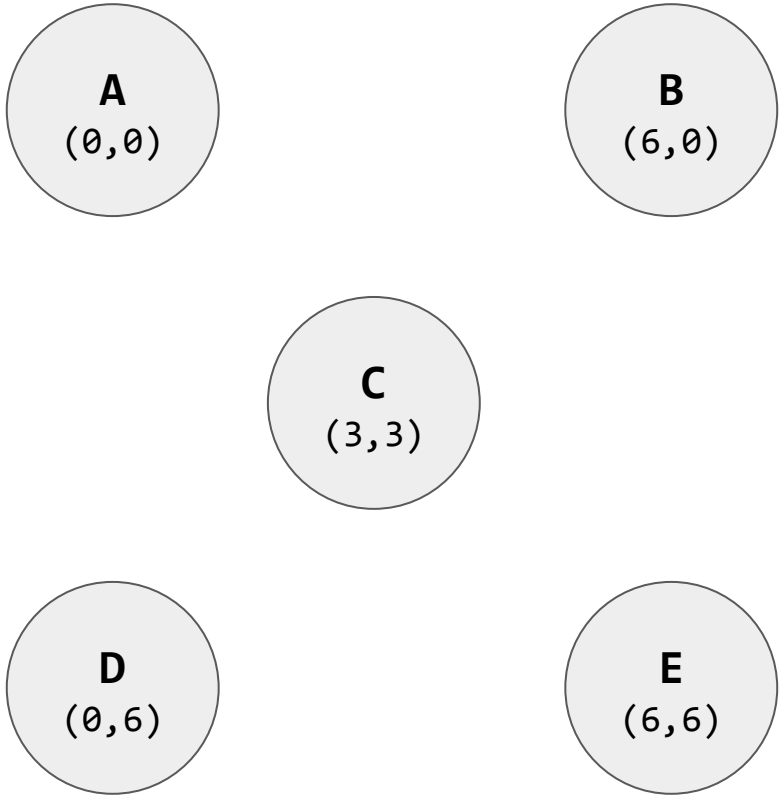
```
// ========== Before ========== //
Ret recursiveFunction(Arg a) {
   if (base-case-holds) {
      return base-case-value;
   } else {
      do-some-work;
      return recursive-step-value;
   }
}
```

```
// ================ After ================ //
Ret recursiveFunction(Arg a, Table& table) {
   if (base-case-holds) {
      return base-case-value;
   } else if (table contains a) {
      return table[a];
   } else {
      do-some-work;
      table[a] = recursive-step-value;
      return recursive-step-value;
   }
}
```

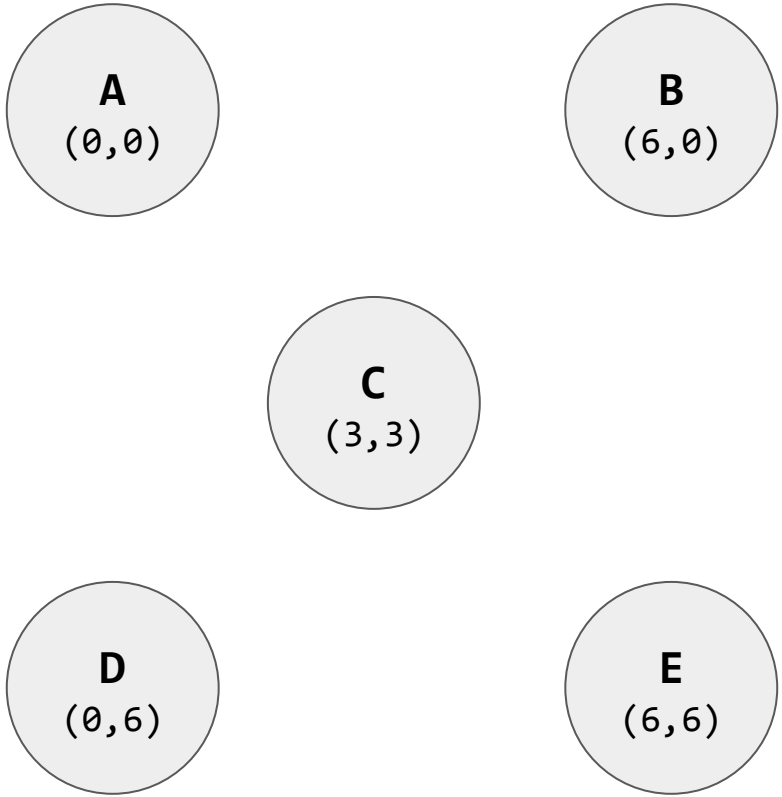
Drill, Baby, Drill!



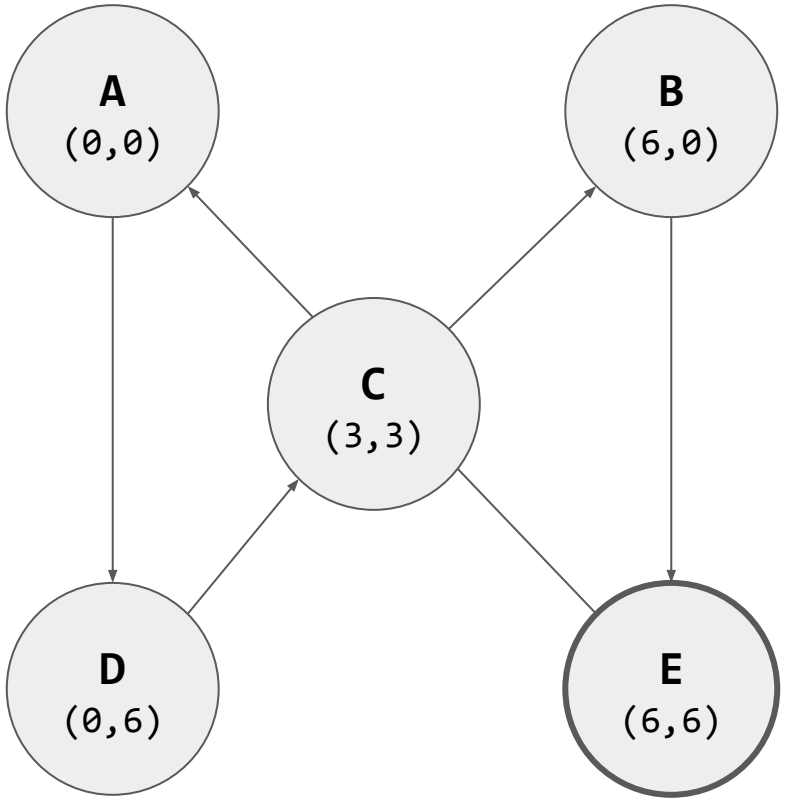
Consider having a drill and a number of drill sites with name and `(x,y)` location



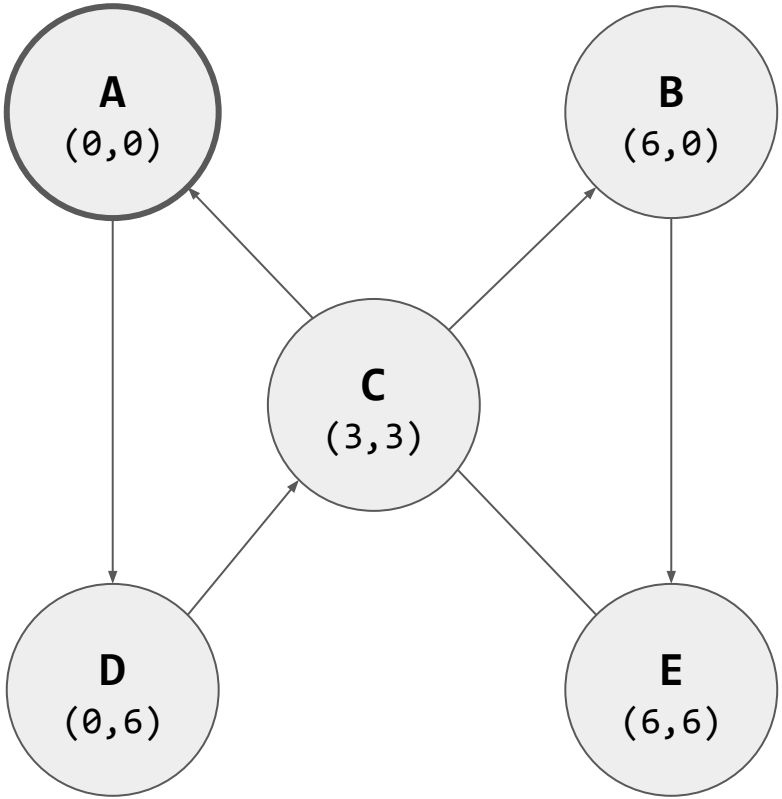
Consider having a drill and a number of drill sites with name and `(x,y)` location



Q: What's the fastest way to go through all drills, starting and ending in the same spot?



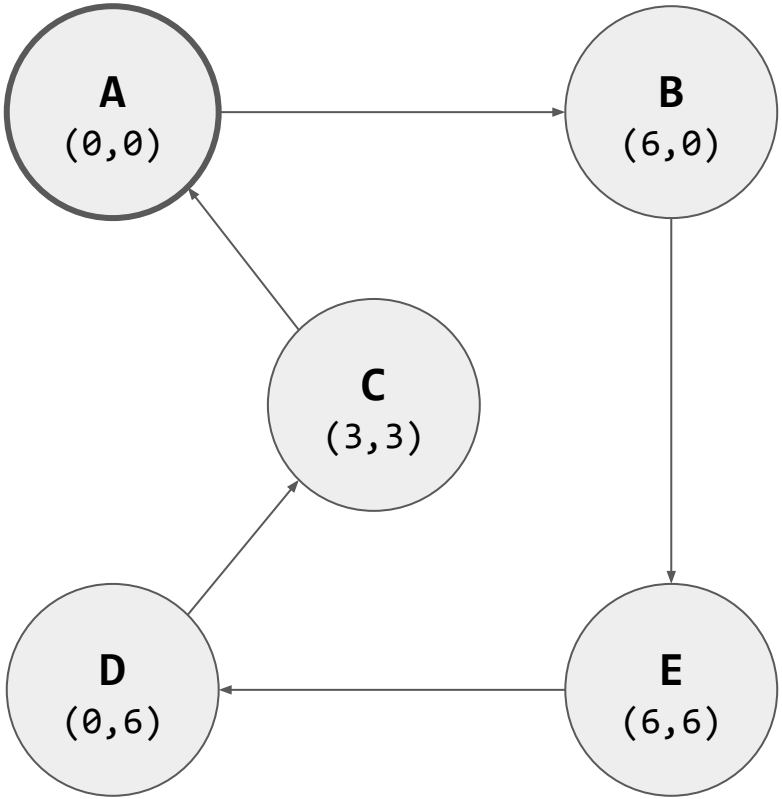
8.48 + 6 + 6 + 6 +
4.24 + 4.24 + 6 =
40.96



6 + 6 + 6 + 4.24 + 4.24 + 6 + 8.48 =

40.96

(specific start point doesn't matter)



Quickest:

4.24 + 6 + 6 + 6 + 4.24 = **26.5**

Drill site:

```
struct DrillSite {
    string name;          // The name of the drill site
    GPoint pt;            // Where the drill site is
}
```

Write the recursive function

```
Vector<DrillSite> bestDrillRouteFor(Vector<DrillSite> sites)
```

returns the optimal order in which the robot should drill holes.

We provide:

```
/**
* Helper function that, given two drill sites, returns the distance
* between them.
*/
double drillDistance(const DrillSite& a, const DrillSite& b) {...}


/**
* Helper function that, given a list of drill sites in order, returns
* the cost associated with drilling all of them in order and returning
* to the start point.
*/
double drillRouteLength(const Vector<DrillSite>& path) {...}
```

You have to create two test files!

drill_handout.txt

```
// FORMAT:
//
// Name (x_coordinate, y_coordinate)

A (0, 0)
B (6, 0)
C (3, 3)
D (0, 6)
E (6, 6)
```

Include in each test case:

- Why you chose that test
- What the test is testing for
- What the optimal answer is

Universal Health Care

You are the new Minister of Health of **Recursia**!

You are tasked to build hospitals to cover as many cities as possible, within a certain budget.

Each potential hospital is represented as such:

```
struct Hospital {
    string name;                    // Name of hospital
    int cost;                       // How much it costs to build
    Set<string> citiesServed;       // Cities it covers
}
```

You want to provide coverage to as many cities as possible.

Imagine you are given **$50,000,000** as a budget.

Consider the following hospital sites:

```
Site 1: Covers Bazekas, Suburb Setz, and Cambinashun.                  Price: $40,000,000
Site 2: Covers Bazekas, Frak Tell, Suburb Setz, and Perumutation City. Price: $50,000,000
Site 3: Covers Hanoi Towers, Jenuratif, and Hooman Pyramids.           Price: $10,000,000
Site 4: Covers Suburb Setz, Permutation City and Baktrak Ing.          Price: $10,000,000
```

Optimal coverage: [Site 3, Site 4] (covers 6 cities)

or

Optimal coverage: [Site 1, Site 3] (also covers 6 cities)

***Note**: you only optimize for city coverage, not for money!*

Write the recursive function

```
Vector<Hospital> bestCoverageFor(Vector<Hospital> sites,
                                 int fundsAvailable)
```

returns list of hospitals that provide coverage to greatest number of cities

Tips and Tricks:

- You can break ties arbitrarily (it doesn't have to be the cheapest one)
- The order of returned hospitals is irrelevant
- If a city is covered twice it can only be counted once

You have to create two test files!

hospital_simple.txt

```
// FORMAT:
//
//[Cities]: City 1, City 2, ...
//[Funds available]: Funds
//[Site]: Cost - City Covered 1, City Covered 2, ...
//[Site]: Cost - City Covered 1, City Covered 2, ...
// ...

[Cities]: Bazekas, Leapofayt, Frak Tell, Hanoi Towers
[Funds available]: 50

[Site]: 13 - Bazekas, Leapofayt, Frak Tell
[Site]: 27 - Hanoi Towers, Frak Tell
[Site]: 62 - Bazekas, Leapofayt, Frak Tell, Hanoi Towers
```

Questions?