Assignment 7: Calligraphy

Assignment by Ashley Taylor and Keith Schwarz

The word "calligraphy" comes from the Greek roots "kallos" and "graphein," literally meaning "beautiful writing." But with a little bit of word play, we can think of "calligraphy" as meaning "beautiful graphs," and that's what this assignment is all about.

Graphs appear in so many contexts that to give even a partial list here wouldn't do them sufficient justice. This assignment is designed to give you a taste of just how powerful and flexible an abstraction they are. We hope that in working through it, you learn how fundamental graph algorithms behave and that you emerge on the other side with some interesting insights about the human condition.

This assignment consists of two parts, which may be completed in any order.

- ***Rising Tides:*** Historical records, backed by satellite data, indicate that global sea levels are slowly rising. You sometimes hear claims like "experts forecast that ocean levels will rise by around two meters over the next century." Those numbers in the abstract don't tell you much. What, exactly, will be underwater if that happens?
- ***Process Optimization:*** If you're trying to perform a multistep task in a group – whether it's making pancakes or running a political campaign – certain steps will hold others back from completing on time. Focusing your efforts at improving the efficiency of those tasks will help get your breakfast ready or your buddy elected in less time. Focusing on other steps in the process won't have any impact.

This is the final assignment of the quarter, and we hope that it's a fitting way to round out your whirlwind ten-week introduction to recursive problem-solving, linked structures, algorithmic efficiency, and abstractions.

Once you've finished this assignment, take a look back at all you've accomplished. Think about where you were at the beginning of the quarter on entry to CS106B. Did you think you'd be able to do all this much in ten weeks?

Due Friday, March 15th at the start of lecture.
You are welcome to work on this assignment in pairs.

☞ ***No late days may be used*** ☜

and

☞ ***no late submissions will be accepted.*** ☜

The Starter Files

The starter files for this final assignment are very similar to the ones from Data Sagas. Here's a breakdown of what the buttons in the demo app do:

- ***Run Tests:*** Yep, there's a testing harness provided for this assignment. Choose this button to run all the tests defined via the `ADD_TEST` macro. You've become such an expert on this at this point that we probably didn't need to tell you that. 😄
- ***Rising Tides:*** Runs the rising tides demo. If you try to load any of the worlds, the program will crash because you haven't implemented anything. After you've finished this part of the assignment, the terrains will load with the water level set to 0m. Changing the water height in the input box and clicking the "Go!" button will recompute where the water will be, which can take a second or two to complete on the larger terrains.
- ***Process Optimizer:*** This demo lets you load processes and see which parts can be optimized. Initially, you should be able to see the processes you load, but after hitting the "Optimize!" button nothing will happen. Once you've finished this part of the assignment, the optimization candidates will appear in bright gold, with processes that don't need to be improved shown in regular white.

There are two we expect you to modify in the course of this assignment:

- `RisingTides.cpp`: You'll put your implementation of the `floodedRegionsIn` function here, along with custom tests. You should not edit this header file, unless you're doing extensions (in which case, everything is fair game!)
- `ProcessOptimizer.cpp`: This is where your code for `optimizationCandidatesFor` will go, plus your custom tests. Again, you should not edit the header file unless you're adding in extensions to the base assignment.

You are welcome to peek around at the demo files if you'd like, though you're not expected to do so.

There is one other file you might want to look at, `Duration.h`, which contains the definition of the `Duration` type needed in the process optimizer. However, don't modify this file; other parts of the code rely on it.

Problem One: Rising Tides

Global sea levels have been rising, and the most recent data suggest that the rate at which sea levels are rising is increasing. This means that city planners in coastal areas need to start designing developments so that an extra meter of water doesn't flood people out of their homes.

Your task in this part of the assignment is to build a tool that models flooding due to sea level rise. To do so, we're going to model terrains as grids of `doubles`, where each `double` represents the altitude of a particular square region on Earth. Higher values indicate higher elevations, while lower values indicate lower elevations. For example, take a look at the three grids to the right. Before moving on, take a minute to think over the following questions, which you don't need to submit. Which picture represents a small hill? Which one represents a long, sloping incline? Which one represents a lowland area surrounded by levees?

0	1	2	3	4	2	1
1	2	3	4	5	4	2
3	4	5	6	6	5	4
2	4	5	7	5	3	2
1	2	4	5	3	2	1
0	1	2	3	1	1	1
0	0	1	2	1	1	1

-1	0	0	4	0	0	1
0	0	4	0	-1	-1	0
0	4	0	0	0	0	0
4	0	-1	-1	0	0	3
0	-1	-2	-1	0	3	0
0	0	-1	0	3	0	0
0	0	0	3	0	0	-1

0	1	2	3	4	5	6
0	1	2	3	4	5	5
0	1	2	3	3	4	4
0	0	1	2	3	3	3
0	0	1	1	2	2	2
-1	-1	0	0	1	1	1
-2	-1	0	0	0	0	0

We can model the flow of water as follows. We'll imagine that there's a water source somewhere in the world and that we have a known height for the water. Water will then flow anywhere it can reach by moving in the four cardinal directions (up/down/left/right) without moving to a location at a higher elevation than the initial water height. For example, suppose that the upper-left corner of each of the three above worlds is the water source. Here's what would be underwater given several different water heights:

Water source at top-left corner

Height: 0m

0	1	2	3	4	2	1
1	2	3	4	5	4	2
3	4	5	6	6	5	4
2	4	5	7	5	3	2
1	2	4	5	3	2	1
0	1	2	3	1	1	1
0	0	1	2	1	1	1

-1	**0**	**0**	4	0	0	1
0	**0**	4	0	-1	-1	0
0	4	0	0	0	0	0
4	0	-1	-1	0	0	3
0	-1	-2	-1	0	3	0
0	0	-1	0	3	0	0
0	0	0	3	0	0	-1

0	**1**	**2**	**3**	**4**	**5**	**6**
0	**1**	**2**	**3**	**4**	**5**	5
0	**1**	**2**	**3**	**3**	4	4
0	**0**	**1**	**2**	**3**	**3**	**3**
0	**0**	**1**	**1**	**2**	**2**	**2**
-1	**-1**	**0**	**0**	**1**	**1**	**1**
-2	**-1**	**0**	**0**	**0**	**0**	**0**

Height: 1m

0	**1**	2	3	4	2	1
1	2	3	4	5	4	2
3	4	5	6	6	5	4
2	4	5	7	5	3	2
1	2	4	5	3	2	1
0	1	2	3	1	1	1
0	0	1	2	1	1	1

-1	**0**	**0**	4	0	0	1
0	**0**	4	0	-1	-1	0
0	4	0	0	0	0	0
4	0	-1	-1	0	0	3
0	-1	-2	-1	0	3	0
0	0	-1	0	3	0	0
0	0	0	3	0	0	-1

0	**1**	**2**	**3**	**4**	**5**	**6**
0	**1**	**2**	**3**	**4**	**5**	5
0	**1**	**2**	**3**	**3**	4	4
0	**0**	**1**	**2**	**3**	**3**	**3**
0	**0**	**1**	**1**	**2**	**2**	**2**
-1	**-1**	**0**	**0**	**1**	**1**	**1**
-2	**-1**	**0**	**0**	**0**	**0**	**0**

Height: 2m

0	**1**	**2**	3	4	2	1
1	**2**	3	4	5	4	2
3	4	5	6	6	5	4
2	4	5	7	5	3	2
1	2	4	5	3	2	1
0	1	2	3	1	1	1
0	0	1	2	1	1	1

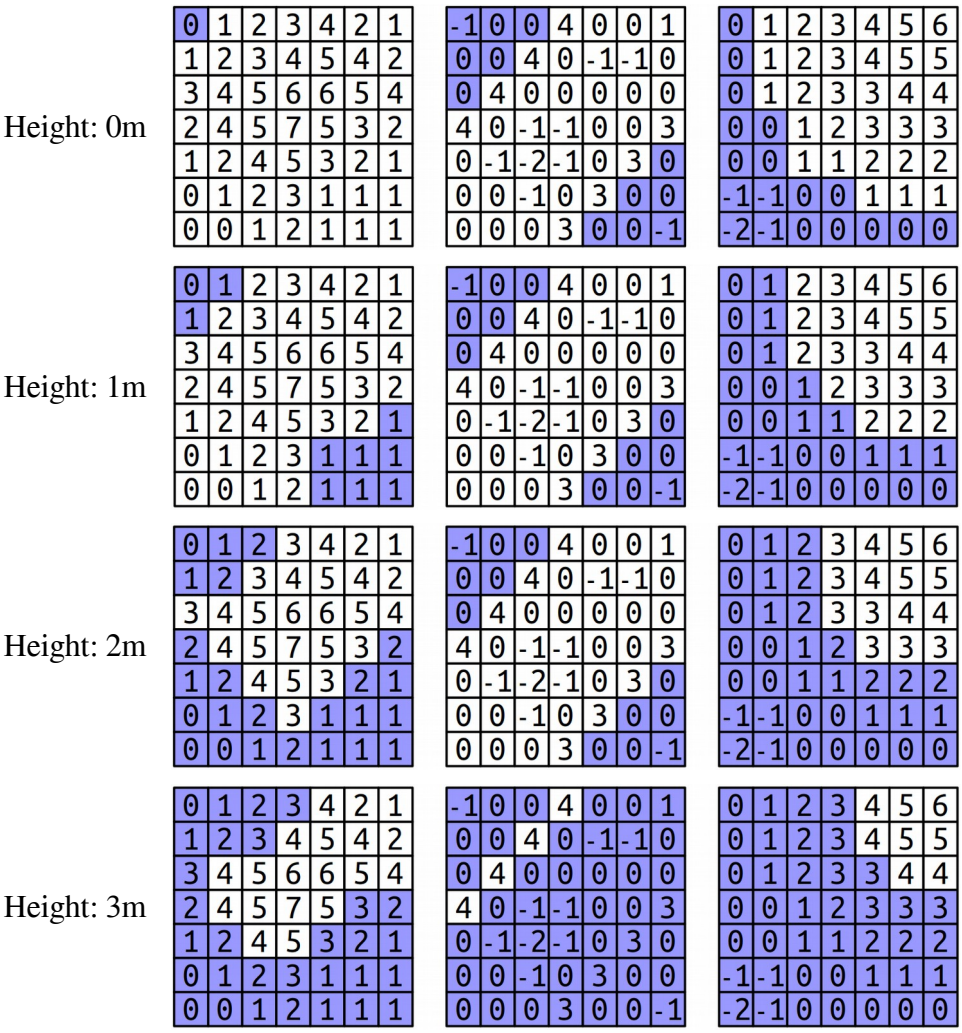
-1	**0**	**0**	4	0	0	1
0	**0**	4	0	-1	-1	0
0	4	0	0	0	0	0
4	0	-1	-1	0	0	3
0	-1	-2	-1	0	3	0
0	0	-1	0	3	0	0
0	0	0	3	0	0	-1

0	**1**	**2**	3	4	5	6
0	**1**	**2**	3	4	5	5
0	**1**	**2**	3	3	4	4
0	**0**	**1**	**2**	3	3	3
0	**0**	**1**	**1**	**2**	**2**	**2**
-1	**-1**	**0**	**0**	**1**	**1**	**1**
-2	**-1**	**0**	**0**	**0**	**0**	**0**

A few things to notice here. First, notice that the water height is independent of the height of the terrain at its starting point. For example, in the bottom row, the water height is always two meters, even though the terrain height of the upper-left corner is either 0m or -1m, depending on the world. Second, in the terrain used in the middle column, notice that the water stays above the upper diagonal line of 4's, since we assume water can only move up, down, left, and right and therefore can't move diagonally through the gaps. Although there's a lot of terrain below the water height, it doesn't end up under water until the height exceeds that of the barrier.

It's possible that a particular grid has multiple different water sources. This might happen, for example, if we were looking at a zoomed-in region of the San Francisco Peninsula, we might have water to both the east and west of the region of land in the middle, and so we'd need to account for the possibility that the water level is rising on both sides. Here's another set of images, this time showing where the water would be in the sample worlds above assume that both the top-left and bottom-right corner are water sources. (We'll assume each water source has the same height.)

Water sources at top-left and bottom-right corners



Notice that the water overtops the levees in the central world, completely flooding the area, as soon as the water height reaches three meters. Water can't flow uphill, so it can't move from a lower elevation to a higher elevation, but it can move across cells at the same height as the water line.

Your task is to implement a function

```
Grid<bool> floodedRegionsIn(const Terrain& terrain, double height);
```

that takes as input a `Terrain` object (described in the `RisingTides.h` header file) and the height of the water level, then returns a `Grid<bool>` indicating, for each spot in the terrain, whether it's under water (`true`) or above the water (`false`).

You know the drill at this point – test your code extensively! You need to write at least ***four*** custom tests for your code here. Think of it this way – if you're trying to make actionable recommendations to municipality about what to expect if the seas rise, you'd better be confident that your code is correct! It would be terrible if a faulty model led to millions – or billions – of dollars spent on unneeded infrastructure projects or on developments ruined by flooding.

The demo app we've included will let you simulate changes to the sea levels for several geographical regions of the United States. The data here is taken from the National Oceanographic and Atmospheric Administration (NOAA), which provides detailed topological maps for the US. Play around with the data sets – what do you find?

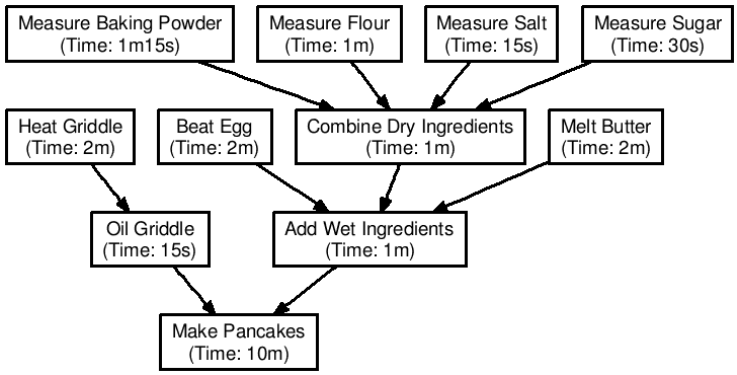
Some notes on this problem:

- Your solution should never trigger a stack overflow. As a heads-up, some of the sample grids bundled with the demo app contain worlds with millions of cells, and the call stack maxes out in the thousands of stack frames.
- Need a refresher on the `Grid` type? Check the Stanford C++ Library Documentation link up on the course website.
- The initial height of the water at a source may be below the level of the terrain there. If that happens, that water source doesn't flood anything, including its initial terrain cell. (This is an edge case where both the "flood it" and "don't flood it" options end up being weird, and for consistency we decided to tiebreak in the "don't flood it" direction.)
- The heights of adjacent cells in the grid may have no relation to one another. For example, if you have the topography of Preikstolen in Norway, you might have a cell of altitude 0m immediately adjacent to a cell of altitude 604m. If you have a low-resolution scan of Mount Whitney and Death Valley, you might have a cell of altitude 4,421m next to a cell of altitude -85m.
- The examples in the previous pages might be good to take a look at if you're curious about how water behaves in various edge cases.
- Be strategic with how you test this function. ***Don't copy the examples from this handout*** – manually entering all the entries in a 7×7 grid and annotating each space with a `bool` is going to take a while and doesn't require any creativity on your part. Instead, see if you can find smaller examples that test important edge cases, along with some more typical cases.
- Your solution to this problem must be efficient. We aren't going to ask you to squeeze every ounce of efficiency out of your code, but you should avoid slow and unnecessary operations. For reference, it's possible to get this one working in time $O(mn)$ on a grid of dimensions $m \times n$, and for full credit you should aim to hit this runtime.

Problem Two: Process Optimizer

In this problem, you will develop code to solve a common issue faced by managers: which tasks should a manager focus on to decrease the total time of creating a product? This problem is general enough that it could be applied to the release workflow for a software product, to a factory assembly-line, or even to making breakfast. The commonality is that all workflows are comprised of individual tasks, each of which requires a specific amount of time to finish, and some of those tasks cannot be started until other tasks have finished. Those tasks can be done on in parallel with one another provided you don't violate any prerequisites, and you can assume you have as many people available to work on tasks as you need.

As an example, let's take the (delicious) process of making pancakes. The specific tasks that need to be done are shown to the right. Arrows represent dependencies; for example, you can't combine dry ingredients until you've finished measuring flour and measuring salt. The time mentioned on each task denotes how long it will take that task to complete. For example, "Add Wet Ingredients" takes a full minute, and "Melt Butter" takes two minutes.



Here's an initial question – assuming you have lots of friends to help you out, how long is it going to take us to finish making pancakes? With a lot of people working in parallel, the answer is thirteen minutes and fifteen seconds. Here's why. (We recommend annotating the diagram above with timing marks to make it easier to follow along.)

- Four people to each measure one dry ingredient. One minute and fifteen seconds later, all the dry ingredients will have been measured.
- Once the dry ingredients are measured, someone combines them together. That takes one minute, so all the dry ingredients are ready at the two minute and fifteen second mark.
- Starting at the same time folks are measuring dry ingredients, three other people beat the egg, melt the butter, and heat the griddle. By the two minute mark, all the wet ingredients will be prepared and the griddle will be hot.
- At the two minute and fifteen second mark, when the dry ingredients have been mixed and the wet ingredients are all prepared, someone stirs the wet ingredients into the dry ingredients to make the batter. It takes a minute to do this, so the batter will be ready three minutes and fifteen seconds after we start.
- Right after the griddle is heated, someone can oil it. The griddle will therefore be heated and oiled by two minutes and fifteen seconds.
- At the three minute and fifteen second mark, we can start making pancakes with our assembled batter and prepared griddle. Ten minutes later, when we've used up the last of the batter, we're done! The total time needed was thirteen minutes and fifteen seconds.

Thirteen minutes and fifteen seconds might not seem like a long time to make pancakes, but ideally we'd like to get the pancakes made faster than that so that we don't have a bunch of hungry people crowding a kitchen. Given the process outlined above, which steps could we speed up to improve the overall completion time? This is more subtle than it looks. For example, take a look at the "Beat Egg" step, which takes a full two minutes. That sounds like a long time – would we get our pancakes made any faster if, say, we could speed up the egg beating to take only one minute?

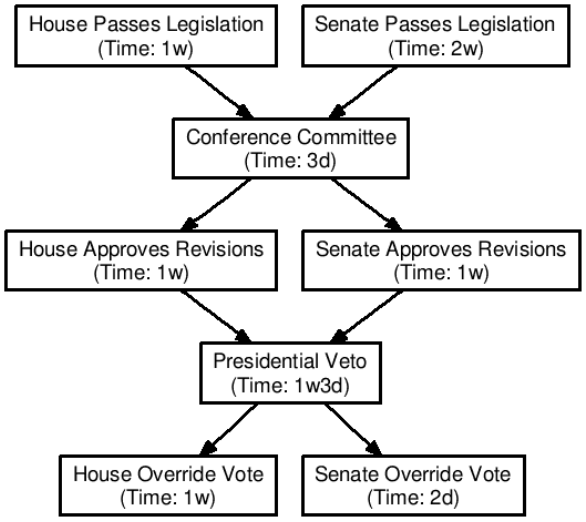
Surprisingly, the answer is no, and here's why. Notice that we only need the egg to be ready at the two minute and fifteen second mark when it's time to stir in all the wet ingredients. Since the egg will be ready after two minutes, with fifteen whole seconds to spare, no matter how much faster we finish things, the pancakes will still get completed at the same time.

On the other hand, there are tasks we could speed up to make things go faster. For example, it takes a full minute and fifteen seconds to measure out the baking powder, which is so long that it holds up the step of combining the dry ingredients. If we were to speed that step up, it would indeed reduce the total time required to make pancakes. Similarly, look at the time required to combine the dry ingredients. That step actually *does* hold up the entire process, since, as mentioned earlier, the step of combining the wet ingredients is held up on the dry ingredients rather than the beaten egg.

Extending this analysis, we can see that there are four tasks that each, independently of the other, could be improved to reduce the overall wait time for pancakes: measuring the baking powder, mixing the dry ingredients, combining the wet ingredients, and actually making the pancakes. Optimizing any one of those tasks reduce the overall completion time. On the other hand, optimizing any individual other step in the process won't actually improve the completion time.

Let's take a look at another example. Here's the procedure that has to happen for a bill to become law in the United States after a presidential veto. First, the House and Senate each need to pass a version of the bill. Once that's done, the two chambers come together at a conference committee to resolve any differences between the versions, and the bill is sent back to both chambers for a follow-up vote. After that, the President vetoes the bill. Both the House and Senate then need to override the veto by a two-thirds vote, at which point the executive grumbles and then begins implementing the new law.

Let's imagine that each step takes the amount of time shown to the right. How long will it take for the bill to become a law? The answer is 41 days, and we encourage you to annotate the diagram showing why this is. Now, which tasks, individually, could be sped up to reduce the time before the bill becomes law? Notice that the Senate takes two weeks to pass the original bill, compared to the House's one week. That holds up the conference committee, so improving the Senate's turnaround time there would be helpful. That conference committee as well is holding things up – neither chamber can vote until it finishes – so speeding that up would improve things. Similarly, the Presidential veto itself takes a long time. So that would be a spot where we could improve things. Finally, the House's one week to pass the override bill, as opposed to the Senate's two days, holds up final passage of the bill.



Notice that that the House and Senate each take one week to pass the revised bill after the conference committee. This means that improving either the House or Senate turnaround time in isolation wouldn't actually change the completion time. We'd need both tasks to speed up in tandem with one another.

To recap:

- We have a process broken down into tasks.
- Any number of tasks can be worked on in parallel.
- Tasks can't start until all the tasks they depend on have finished.
- We want to find tasks that, in isolation, would reduce the completion time if optimized.

How can you tell whether optimizing a particular ask will actually speed things up? There are several strategies you could use, but we recommend the following approach, which we've scouted out and can confirm works in all cases:

1. Determine how long the entire process takes to complete. (We'll consider the process done when the very last task finishes.)
2. For each task, independently of the others, set that task's duration to zero, then recompute how long the process takes to complete. If the completion time is the same, that task isn't holding anything up. If the completion time decreased, then optimizing that task will indeed make a difference.

Your task is to write a function

```
HashSet<Task *> optimizationCandidatesFor(const HashSet<Task *>& process);
```

that takes in a process, represented as a collection of the tasks that make it up, and then returns a `HashSet<Task *>` containing each task in that process that, if optimized, would unilaterally improve the overall completion time of the process.

Some notes on this problem:

- You may want to use topological sorting here. Remember that to implement topological sort, you need to use the recursive version of DFS, not the iterative one.
- There are several custom types you're going to need to wrap your head around here. Read over the documentation for `Task` and `Duration` before you start writing any code.
- The algorithm we've suggested says to set the duration of a particular task to zero for the purposes of checking whether it's an optimization candidate. Do you *actually* need to set its duration to zero? Or can you *simulate* setting its duration to zero while leaving its actual duration unmodified?
- You don't need to worry about the case where the process in question contains circular dependencies. You can assume that it is actually possible to complete the tasks in some order. Otherwise, it's going to take a long time for everything to finish. 😀
- Some tasks might not have any dependencies and might not have anything that depends on them.
- There can be any number of tasks, including zero.
- Your solution to this problem needs to be efficient, in the sense that minor changes to the size of the process shouldn't lead to huge blowups in runtime. Our solution to this problem runs in time $O(n(m + n))$, where n is the number of tasks and m is the number of arrows in the dependency diagram. You don't need to exactly match this runtime, but you should aim for efficiency. As a reminder, the graph algorithms we described in class are all extremely fast.

Test your code extensively! You are required to write at least ***four*** test cases for this part of the assignment. This will require you to assemble some processes one task at a time, and that will require you to allocate some memory and manually wire some pointers together. Make sure not to leak any memory in your tests.

The demo for this part of the assignment will let you load various processes and see what your algorithm says to do for those processes. These demos aren't designed for testing purposes, so if you find that you're getting back answers that seem incorrect, be sure to head back to your test cases to tune up your program.

(Optional) Extensions!

There are a great many ways you could extend these solutions – some theoretical, some practical. Here are some ideas to help get you started:

- ***Rising Tides:*** The sample worlds here have their data taken from the NOAA, which is specific to the United States, but we'd love to see what other data sets you can find. Get some topographical data from other sources, modify it to fit our file format, and load it into the program. (You can find information about the file format in the `res/terrains/README` file.) What information – either for good or for ill – does that forecast?

  Let's imagine that you want to ensure that some specific region on the map doesn't get flooded, and to do so, you decide build a collection of levees to keep the water out. You could build levees all the way along the coastline, but that's totally impractical and much too expensive. What's the cheapest set of levees you could build to accomplish this? Although this problem might seem quite challenging, there are some beautiful algorithms to help you solve it. Look up the ***Ford-Fulkerson algorithm*** and the ***max-flow/min-cut theorem*** for details.

- ***Process Optimizer:*** In the process optimization problem, we assumed that you have as many people available to work on the task in question as you need. But what if you only have a few people available to help out at once? For example, let's say you want to make pancakes, but you only have three people available. Does that change which tasks you might want to optimize to improve the overall efficiency? If so, how? Write code that lets you cap the number of workers when determining what can be optimized.

  What happens if you make some tasks optional? Or if you have pairs of tasks where only one of the two needs to be completed? How does that change things?

Submission Instructions

Once you've finished this assignment, visit Paperless and submit your `RisingTides.cpp` and `ProcessOptimizer.cpp` files, along with any other files you modified in the course of completing this assignment. Before doing so, make sure you've written the requisite number of tests for each part.

And that's it! You're done!

Good luck, and have fun!