Before you turn this problem in, make sure everything runs as expected. First, **restart the kernel** (in the menubar, select Kernel→Restart) and then **run all cells** (in the menubar, select Cell→Run All).

Make sure you fill in any place that says `YOUR CODE HERE` or "YOUR ANSWER HERE", as well as your name and collaborators below:

In [ ]:
```
NAME = ""
COLLABORATORS = ""
```

CSE 30 Spring 2022 - Homework 14

Cooking Times and Dynamic Programming

Instructions

Please disregard the YOUR NAME and COLLABORATORS above. They are put there atomatically by the grading tool. You can find instructions on how to work on a homework on Canvas. Here is a short summary:

Submitting your work

To submit your work:

- First, click on "Runtime > Restart and run all", and check that you get no errors. This enables you to catch any error you might have introduced, and not noticed, due to your running cells out of order.
- Second, download the notebook in .ipynb format (File > Download .ipynb) and upload the .ipynb file to this form. This homework is due by **11:59pm on Thursday, 26 May 2022**.

You can submit multiple times; the last submission before the deadline is the one that counts.

Homework format

For each question in this notebook, there is:

- A text description of the problem.

- One or more places where you have to insert your solution. You need to complete every place marked:

  `# YOUR CODE HERE`

  and you should not modify any other place.

- One or more test cells. Each cell is worth some number of points, marked at the top. You should not modify these tests cells. The tests pass if no error is printed out: when there is a statement that says, for instance:

  `assert x == 2`

  then the test passes if `x` has value 2, and fails otherwise. You can insert a `print(x)` (for this case!) somewhere if you want to debug your work; it is up to you.

Notes:

- Your code will be tested both according to the tests you can see (the `assert` statements you can see), *and* additional tests. This prevents you from hard-coding the answer to the particular questions posed. Your code should solve the *general* intended case, not hard-code the particular answer for the values used in the tests.
- **Please do not delete or add cells!** The test is autograded, and if you modify the test by adding or deleting cells, even if you re-add cells you delete, you may not receive credit.
- **Please do not import modules that are not part of the standard library.** You do not need any, and they will likely not available in the grading environment, leading your code to fail.
- **If you are inactive too long, your notebook might get disconnected from the back-end.** Your work is never lost, but you have to re-run all the cells before you continue.
- You can write out print statements in your code, to help you test/debug it. But remember: the code is graded on the basis of what it outputs or returns, not on the basis of what it prints.
- **TAs and tutors have access to this notebook,** so if you let them know you need their help, they can look at your work and give you advice.

Grading

Each cell where there are tests is worth a certain number of points. You get the points allocated to a cell only if you pass *all* the tests in the cell.

The tests in a cell include both the tests you can see, and other, similar, tests that are used for grading only. Therefore, you cannot hard-code the solutions: you really have to solve the

essence of the problem, to receive the points in a cell.

Code of Conduct

- Work on the test yourself, alone.
- You can search documentation on the web, on sites such as the Python documentation sites, Stackoverflow, and similar, and you can use the results.
- You cannot share your work with others or solicit their help.

In the previous chapter, on a particular run of the scheduler for pasta carbonara, I got:

```
['dice pancetta',
 'fill pot with water',
 'put eggs in bowl',
 'put pancetta in pan',
 'dice onions',
 'put oil and butter in pan',
 'bring pot of water to a boil',
 'add salt to water',
 'put pasta in water',
 'beat eggs',
 'put onions in pan',
 'colander pasta',
 'cook pancetta',
 'serve']
```

This not only makes little sense; it violates the first axiom of making pasta, which states that pasta must be served as soon as fully cooked. Above, we take our sweet time to cook pancetta *after* the pasta has been colandered! Indeed, one of the challenges of true cooking is to ensure that things are ready at exactly the right times. The pancetta has to be done cooking exactly when the pasta is colandered, so that all is united when it's just hot and right. The difficulty is that it's hard to decide precisely when do we need to start slicing onions, or heating water, to achieve this synchronicity.
We will help by designing a *timed scheduler*.

The idea behind the timed scheduler is to endow each task with an *execution time*. We specify then that the whole schedule has to be done at time 0, and we then go back and compute when we need to start the various tasks, or rather, when is the last possible moment to start the individual tasks, to be done at time 0.

The times at which we start tasks will be negative; if you find it strange to use negative time, you might want to think at it as a countdouwn, where we finish with -3, -2, -1, and at 0, not quite liftoff, but a call that dinner is ready; more prosaic perhaps, but also more commestible.

Computing start times for the tasks

How do we annotate each task with its starting time?

The idea is as follows. For a task x, denote by $t(x)$ its start time, denote by $d(x)$ its duration, and denote by $s(x)$ its set of successors, that is, the set of tasks that depend on x directly.

First, consider a task with no successors. The task can be done last, and so we simply need to set $t(x) = -d(x)$ to ensure we finish at time 0.

Second, consider a task x with successors $s(x) = \{y_1, y_2, \ldots, y_n\}$. In order to finish x on time for the start of y_i, for $1 \leq i \leq n$, it must be $t(x) + d(x) \leq t(y_i)$. As we wish to start x as late as possible, to be done "just in time", $t(x)$ must be the *largest* value such that $t(x) \leq t(y_i) - d(x)$, for all $1 \leq i \leq n$. Thus, it is easy to see that we must choose

$$t(x) = -d(x) + \min_{y \in s(x)} t(y) \ .$$

An abstract algorithm for starting times

All we need to do is to ensure that the constraint

$$t(x) = -d(x) + \min_{y \in s(x)} t(y)$$

holds for all tasks (with the convention that $\min \emptyset = 0$). How can we do this?

We can solve this following an approach known as *dynamic programming*. If you have a set of elements (in our case, tasks), along with a constraint on the elements (such as our formula for $t(x)$ for a task x), you can sometimes treat the constraint as an update rule, rather than an equality. To do so, we start by setting the start times to be as late as possible, that is, to $-d(x)$. We then consider all task x in turn, and we *update* the start time of x via:

$$t(x) := -d(x) + \min_{y \in s(x)} t(y) \ .$$

Once we cannot perform any update, the constraint will hold for all tasks.

Our algorithm can thus be expressed as follows:

1. Set $t(x) = -d(x)$ for all tasks x.
2. Repeat:

> For all tasks x, do: $t(x) := -d(x) + \min_{y \in s(x)} t(y)$

- until no change occurs (that is, until $t(x)$ is unchanged for all tasks x).

The "until no change occurs" above means that we have reached a *fixed point* of the update rule: the update rule, when applied, causes no change in the values.

Informally, the algorithm works because initially we set $t(x) = 0$ for every task, thus setting the start time to be as late as possible; we then anticipate the start time only if we have a

good reason to do so, to leave enough time for a dependent task to also execute before time 0.

Correctness of the algorithm

This section is optional, and can be skipped at a first reading.

How can we prove precisely that the algorithm works? Denote by $t^*(x)$ the *true* latest starting time for x.
The proof proceeds in the following steps.

1. **Termination:** first, we will prove that the algorithm terminates.
2. **Overapproximation:** Second, we will prove that throughout the algorithm, $t(x)$ is always later than the latest starting time, or $t(x) \geq t^*(x)$.
3. **Underapproximation:** Then, we show that if the relation $t(x) = -d(x) + \min_{y \in s(x)} t(y)$ holds, then we have enough time to do all tasks, that is, $t(x) \leq t^*(x)$.
4. From 2 and 3 above, we conclude that at the fixpoint we have $t(x) = t^*(x)$, QED.

We prove 1, 2, 3 in the following. Let $D(x)$ be the set of tasks that depend on x; of course, $s(x) \subseteq D(x)$, but $D(x)$ also contains the tasks that depend on x via intermediate tasks.

Termination

The one difficult step consists in proving termination, because, well, the algorithm may actually *not* terminate!

If there are cycles in the dependency graph, the algorithm never terminates. Consider tasks a, b, c, with a dependent on b, b on c, and c on a; assume that all tasks take 1 second. If we set $t(a) = 0$, we update then $t(c) = -1$, then $t(b) = -2$, then $t(a) = -3$, and so on and so forth.

If there are no cycles in the dependency graph, then every path in the graph is of finite length (finite number of edges), because no node can be repeated along the path. Let $n(x)$ be the length of the longest dependency path ending at x, so that $n(x) = 0$ if no task depends on x. We prove by induction that the value of $t(x)$ is determined after at most $n(x)$ iterations of step 2 above. This shows that the algorithm terminates in at most as many iterations of step 2 as there are tasks.

Precisely, the inductive assertion is that $t(x)$ does not change after $n(x)$ iterations of the algorithm; the induction is over $n(x)$.

For the base case, consider tasks x with no dependents, so with $n(x) = 0$. Initially, we set $t(x) = -d(x)$, and as the set $D(x)$ of successor of x is empty, the update reduces to $t(x) := -d(x)$, which leaves $t(x)$ unchanged.

For the induction step, consider a task x with $n(x) = k$. Obviously, $n(y) < k$ for all $y \in D(x)$, and by induction hypothesis, the values $t(y)$ of $y \in D(x)$ are determined after $k - 1$ iterations. At iteration k, we set $t(x) := -d(x) + \min_{y \in s(x)} t(y)$; from then on, as there will not be any change in the right hand side of the update rule, the value of $t(x)$ will also not change.

Overapproximation

We want to show that for all tasks x, we have $t(x) \geq t^*(x)$. This is proved by induction on the number of iterations of the repeat (step 2) of the algorithm.

The base case, for zero iterations, follows because initially we have $t(x) = 0$ and $t^*(x) \leq 0$.

In every subsequent iteration of the algorithm, denote with $t(x)$ the start time of x before the iteration (at the end of the previous iteration), and with $t'(x)$ the start time of x at the end of the iteration.
If $t'(x) \geq t(x)$, the results follows immediately by induction.

Assume that $t'(x) < t(x)$. Then, there is a task $y \in s(x)$ such that $t'(x) = -d(x) + t(y)$. By induction hypothesis, we have $t(y) \geq t^*(y)$, and so

$$t'(x) \geq -d(x) + t^*(y)$$

We have $t^*(x) \leq -d(x) + t^*(y)$, to give enough time for task x to run. This leads to $t'(x) \geq t^*(x)$, completing the induction case.

Underapproximation

For the proof of the under-approximation, it helps to define an *update operator* A. Let $\vec{t}$ be the vector consisting of all starting times. Then, our rule $t(x) := -d(x) + \min_{y \in s(x)} t(y)$ can be written as $\vec{t} = A(\vec{t})$, where A is an update operator operating on the vector of start times.

The operator A is monotonic, that is, $\vec{t} \leq \vec{u}$ implies $A(\vec{t}) \leq A(\vec{u})$, where the inequalities are interpreted in pointwise fashion. This is simply because the functions $\min$ and $+$ are monotonic.

Furthermore, let $\vec{t}_0$ be defined by $t_0(x) = -d(x)$, and let $\vec{t}_1$ be defined by $t(x) = -d(x) + \min y \in s(x) t_0(y)$; in other words, let $\vec{t}_0$ be the initial vector, and $\vec{t}_1 = A(\vec{t}_0)$ be the first iterate.
In general, let $\vec{t}_{n+1} = A(\vec{t}_n)$ for $n \geq 0$. Since $\min y \in s(x) t_0(y) \leq 0$, we have $\vec{t}_1 \leq \vec{t}_0$, and thus, $A(\vec{t}_1) \leq A(\vec{t}_0)$, or $\vec{t}_2 \leq \vec{t}_1$. This in turn, again by monotoninicy of A, leads to $\vec{t}_3 \leq \vec{t}_2$, so that the sequence $\vec{t}_0, \vec{t}_1, \vec{t}_2, \ldots$, is monotonically decreasing. In particular, we have that $t_n(x) \leq -d(x)$ for all $n \geq 0$.

If $t(x) = -d(x) + \min_{y \in s(x)} t(y)$ holds for all tasks x at the end of the iterations, then also $t(x) + d(x) \leq t(y)$ holds for all tasks x and their dependent tasks y. Thus, if we started every task x at $t(x)$, we would be able to complete every task x by $t(x) + d(x)$. Since $t(x) + d(x) \leq 0$, $t(\cdot)$ is a feasible schedule, and it must be earlier or equal to the latest feasible schedule, so that $t(x) \leq t^*(x)$ for every task x.

A naive implementation

A naive implementation of the algorithm is as follows.

In [ ]:
```
class Task(object):
    """We represent a task, with a name, duration, and start time."""

    def __init__(self, name=None, duration=0.):
        assert duration > 0, "The task duration needs to be positive."
        self.name = name
        self.duration = duration
        self.start_time = None # Not known initially.

    def __repr__(self):
        """This is used by our functions before to print tasks."""
        return "%s[%f][s=%r]" % (self.name, self.duration, self.start_time)

    def __hash__(self):
        """We need to define this, so that tasks can be used as keys
        to dictionaries."""
        return hash(self.name)

    def __eq__(self, other):
        """Used for comparisons."""
        return (self.name == other.name and self.duration == other.duration
                and self.start_time == other.start_time)
```

In [ ]:
```
from collections import defaultdict

class TimedScheduler(object):

    def __init__(self):
        # Once the tasks are scheduled, they will appear in this list in
        # order of increasing start time.
        self.tasks = set()
        self.predecessors = defaultdict(set) # Not really needed.
        self.successors = defaultdict(set)
        self.sorted_tasks = None

    def add_task(self, t, dependencies):
        """Adds a task t with given dependencies."""
        # Makes sure we know about all tasks mentioned.
        self.tasks.add(t)
        self.tasks.update(dependencies)
        # Adds to the predecessors of t the tasks in the dependencies.
        self.predecessors[t] = self.predecessors[t] | set(dependencies)
        for u in dependencies:
```

```
            self.successors[u].add(t)

    def compute_schedule(self):
        """Computes a schedule"""
        # Initialization step.
        for t in self.tasks:
            t.start_time = - t.duration
        # Iterations.
        # If the graph is acyclic, we can do at most n-1 updates, as n-1 is
        # the length of the longest path.  So we try to do n updates, and
        # if the n-th update still shows change, we stop with an error,
        # reporting that the graph contains a cycle.
        for num_iteration in range(len(self.tasks)):
            changed = False
            for x in self.tasks:
                new_time = - x.duration + min([0.] + [y.start_time
                                                      for y in self.successors[x]])
                changed = changed or new_time != x.start_time
                x.start_time = new_time
            if not changed:
                break
        assert not changed, "The graph contains a cycle."
        self.sorted_tasks = list(self.tasks)
        self.sorted_tasks.sort(key = lambda t : t.start_time)

    def get_schedule(self):
        return self.sorted_tasks
```

We have now reached the apex of our intellectual achievement: we can finally know when to start the various tasks involved in making pasta carbonara. Such effort, but such reward, also!

In [ ]:

```
tcarbonara = TimedScheduler()

# Let's define the tasks first.
t_onions = Task('dice onions', 2.)
t_pancetta = Task('dice pancetta', 10.)
t_oilbutter = Task('put oil and butter in pan', 1.)
t_pan = Task('cook pancetta', 35.)

t_fill = Task('fill pot with water', 1.)
t_boil = Task('bring water to a boil', 10.)
t_salt = Task('add salt to water', 1.)
t_pasta = Task('cook pasta', 13.)

t_eggs = Task('put eggs in bown', 2.)
t_beat = Task('beat eggs', 1.)

t_serve = Task('serve pasta', 2.)

# First, the part about cooking the pancetta.
tcarbonara.add_task(t_onions, [])
tcarbonara.add_task(t_pancetta, [])
tcarbonara.add_task(t_oilbutter, [])
tcarbonara.add_task(t_pan, [t_onions, t_pancetta, t_oilbutter])
```

```
tcarbonara.add_task(t_fill, [])
tcarbonara.add_task(t_boil, [t_fill])
tcarbonara.add_task(t_salt, [t_boil])
tcarbonara.add_task(t_pasta, [t_salt])

tcarbonara.add_task(t_eggs, [])
tcarbonara.add_task(t_beat, [t_eggs])

tcarbonara.add_task(t_serve, [t_beat, t_pasta, t_pan])
```

In [ ]:
```
tcarbonara.compute_schedule()
```

In [ ]:
```
for t in tcarbonara.get_schedule():
    print(t.name, ":", t.start_time)
```

```
dice pancetta : -47.0
dice onions : -39.0
put oil and butter in pan : -38.0
cook pancetta : -37.0
fill pot with water : -27.0
bring water to a boil : -26.0
add salt to water : -16.0
cook pasta : -15.0
put eggs in bown : -5.0
beat eggs : -3.0
serve pasta : -2.0
```

Let us check that this will not run forever in case of loops.

In [ ]:
```
ta = Task('a', 1.)
tb = Task('b', 1.)
tc = Task('c', 2.)
s = TimedScheduler()
s.add_task(ta, [tb])
s.add_task(tb, [tc])
s.add_task(tc, [ta])
try:
    s.compute_schedule()
except AssertionError:
    print("Detected a loop")
```

```
Detected a loop
```

An efficient implementation of the algorithm

Let us consider again the scheduling method of our TimedScheduler class.

In [ ]:
```
def compute_schedule(self):
    """Computes a schedule"""
    # Initialization step.
    for t in self.tasks:
        t.start_time = - t.duration
    # Iterations.
    # If the graph is acyclic, we can do at most n-1 updates, as n-1 is
```

```
    # the length of the longest path.  So we try to do n updates, and
    # if the n-th update still shows change, we stop with an error,
    # reporting that the graph contains a cycle.
    for num_iteration in range(len(self.tasks)):
        changed = False
        for x in self.tasks:
            new_time = - x.duration + min([0.] + [y.start_time
                                                  for y in self.successors[x]])
            changed = changed or new_time != x.start_time
            x.start_time = new_time
        if not changed:
            break
    assert not changed, "The graph contains a cycle."
    self.sorted_tasks = list(self.tasks)
    self.sorted_tasks.sort(key = lambda t : t.start_time)
```

This implementation is correct, but it does waste quite a bit of unnecessary work. For a task x, we note that the start time $t(x)$ of x changes only when the start time of some successor of x has changed. Thus, we can obtain amore efficient algorithm by propagating change, updating the starting time of a task only when the starting time of some of its successors changed.

To this end, we maintain a *changed* set, which keeps track of the tasks whose starting times have changed. Let L be the set of tasks that have no successors; these are the tasks that can be done last. Initially, we set $t(x) = -d(x)$ for $x \in L$, and $t(x) = 0$ otherwise; we also set *changed* to L. Then, we iteratively pick a task x in *changed*, and for all its predecessors y, we set $t(y) = min\{t(y), -d(y) + t(x)\}$; if this update changes the value of $t(y)$, we put y in *changed*. The process continues until there is no task in *changed*, so that all changes have been propagated.

The algorithm can be written as follows.

In [ ]:
```
def fast_compute_schedule(self):
    """Computes a schedule for a timed scheduler."""
    # Initialization step.
    for t in self.tasks:
        t.start_time = - t.duration
    changed = {x for x in self.tasks if len(self.successors[x]) == 0}
    while len(changed) > 0:
        x = changed.pop()
        for y in self.predecessors[x]:
            new_start_time = min(y.start_time, -y.duration + x.start_time)
            if new_start_time < y.start_time:
                changed.add(y)
                y.start_time = new_start_time
    self.sorted_tasks = list(self.tasks)
    self.sorted_tasks.sort(key = lambda t : t.start_time)

TimedScheduler.fast_compute_schedule = fast_compute_schedule
```

In [ ]:
```
tcarbonara.fast_compute_schedule()
for t in tcarbonara.get_schedule():
```

```
    print(t.name, ":", t.start_time)
```

```
dice pancetta : -47.0
dice onions : -39.0
put oil and butter in pan : -38.0
cook pancetta : -37.0
fill pot with water : -27.0
bring water to a boil : -26.0
add salt to water : -16.0
cook pasta : -15.0
put eggs in bown : -5.0
beat eggs : -3.0
serve pasta : -2.0
```

Let us check that the two schedules computed are indeed the same.

In [ ]:
```
tcarbonara.compute_schedule()
# We need to produce a static dictionary: the schedule is formed
# by tasks with a .start_time defined, and if we call the scheduler
# one more time, these .start_time will be modified.
d1 = {t.name: t.start_time for t in tcarbonara.get_schedule()}
tcarbonara.fast_compute_schedule()
d2 = {t.name: t.start_time for t in tcarbonara.get_schedule()}
d1 == d2
```

Out[ ]:
```
True
```

Writing two implementations of the same algorithm, one simple, the other efficient, and testing them by checking that they give the same results, is a useful and common testing technique.

Exercise: the fast_compute_schedule method does not check for loops. Modify it so that it terminates with an assertion error if the dependency graph contains a loop. There are various ways in which this can be done, but one of the simplest is as follows. This is a rough idea; you will need to take care of some corner cases (the tests will help).

If there is a loop, the algorithm will keep updating the starting times of the tasks in the loop, so that the starting times will become smaller and smaller. If X is the set of all tasks, clearly the worst case for a task x is that all others have to be done before it, in which case $t(x) = -\sum_{x \in X} d(x)$. So if you ever assign a starting time to a task that is *smaller than* $-\sum_{x \in X} d(x)$, there must be a loop.

Note that this relies on the assumption that all tasks have strictly positive durations.

In [ ]:
```
#@title Importing nose

# Let us ensure that nose is installed.
try:
    from nose.tools import assert_equal, assert_true
    from nose.tools import assert_false, assert_almost_equal
except:
    !pip install nose
```

```
    from nose.tools import assert_equal, assert_true
    from nose.tools import assert_false, assert_almost_equal
```

In [ ]:
```
### Definition of `fast_compute_schedule` that checks for loops

class DependencyLoop(Exception):
    pass

def safe_fast_compute_schedule(self):
    """Computes a schedule for the TimedScheduler.  Raises DependencyLoop
    if the dependency graph contains a loop."""
    # YOUR CODE HERE
    sum = 0.
    for t in self.tasks:
        t.start_time = - t.duration
        sum += abs(t.duration)
    changed = {x for x in self.tasks if len(self.successors[x]) == 0}
    if len(changed) == 0:
      raise DependencyLoop
    while len(changed) > 0:
        x = changed.pop()
        for y in self.predecessors[x]:
            new_start_time = min(y.start_time, -y.duration + x.start_time)
            if new_start_time < y.start_time:
                if new_start_time < -1 * sum:
                    raise DependencyLoop
                changed.add(y)
                y.start_time = new_start_time
    self.sorted_tasks = list(self.tasks)
    self.sorted_tasks.sort(key = lambda t : t.start_time)

TimedScheduler.fast_compute_schedule = safe_fast_compute_schedule
```

As usual, let us test this.

In [ ]:
```
### Tests with loop

# First, a graph consisting only of a loop.
ta = Task('a', 1.)
tb = Task('b', 1.)
tc = Task('c', 2.)
s = TimedScheduler()
s.add_task(ta, [tb])
s.add_task(tb, [tc])
s.add_task(tc, [ta])
loop = False
try:
    s.fast_compute_schedule()
except DependencyLoop:
    loop = True
assert_true(loop)
```

In [ ]:
```
### More tests with loop

# Then, a more complex example with a loop.
```

```
ta = Task('a', 1.)
tb = Task('b', 1.)
tc = Task('c', 2.)
td = Task('d', 4.)
te = Task('e', 3.)
s = TimedScheduler()
s.add_task(ta, [tb, td])
s.add_task(tb, [tc])
s.add_task(tc, [ta])
s.add_task(td, [])
s.add_task(te, [tb])
loop = False
try:
    s.fast_compute_schedule()
except DependencyLoop:
    loop = True
assert_true(loop)
```

In [ ]:
```
### Tests without loop

# An example without a loop.
ta = Task('a', 1.)
tb = Task('b', 1.)
tc = Task('c', 2.)
td = Task('d', 4.)
te = Task('e', 3.)
s = TimedScheduler()
s.add_task(ta, [tb, td])
s.add_task(tb, [tc])
s.add_task(tc, [td])
s.add_task(td, [])
s.add_task(te, [tb])
loop = False
try:
    s.fast_compute_schedule()
except DependencyLoop:
    loop = True
assert_false(loop)
```

Epilogue

Scheduling is a complex problem, with many variants. Consider for instance our timed scheduling problem: it models only partially how one can cook. For instance, people generally cannot slice two things at once: if there's only one person, some tasks, such as slicing, become mutually exclusive. One may also have limitations on how many things can be cooked at the same time. These are known, in general, as resource constraints on a schedule. Schedules can have other types of constraints too: particular tasks may be able to be started only after predetermined times, and so forth. For the two problems we have seen, scheduling with dependencies, and with duration constraints, efficient optimal solutions are known. For complex scheduling problems, it is often the case that such efficient solutions are

not available, and one must write algorithms that search for a solution, often with the help of heuristics.