```
In [ ]: # Initialize Otter
        import otter
        grader = otter.Notebook()
```

```
In [ ]: # Please do not change this cell because some hidden tests might depend on i
        import os

        # Only install packages if not in cs187-env
        if 'cs187-env' not in os.environ.get('CONDA_PREFIX', ''):
            import subprocess
            import sys
            subprocess.run([sys.executable, '-m', 'pip', 'install', '-q', '-r', 'rec
                           check=True)
```

Many of the data-heavy approaches to NLP are enabled by advances in parallel processing that make what were once intractable computations practical. This notebook demonstrates the issue and the `torch` technologies that apply to it, especially the "tensor" data type.

New bits of Python used for the first time in this lab, and which you may therefore want to review, include:

- `assert`
- `globals`
- `len`
- `math.inf`
- `random.choices`
- `timeit.timeit`
- `torch.tensor`
- `torch.equal`
- `torch.rand`
- `torch.shape`
- `torch.transpose`
- `torch.view`
- `torch.zeros`
- `torch.zeros_like`

Recall that in this and all labs, we will have you carry out several exercises in notebook cells. The cells you are to do are marked '`#TODO`'. They will typically have a `...` where your code or answer should go. Where specified, you needn't write code to calculate the answer, but instead, simply work out the answer yourself and enter it.

```
In [ ]: import torch
        import random

        from timeit import timeit
```

Numeric vectors can be implemented in Python in many ways. Most directly, Python provides a built-in `list` data type, which we could use to implement a vector. Here, we generate a couple of example vectors as lists each containing 1000 integers between 0 and 99.

In [ ]:
```
a1 = random.choices(range(100), k=1000)
a2 = random.choices(range(100), k=1000)
```

An example: dot product

The dot product of two vectors v and w is the sum of their componentwise product, $\sum_i v_i \cdot w_i$, which can be calculated with a simple for-loop.

In [ ]:
```
def dotproduct(v, w):
    assert len(v) == len(w)
    sum = 0
    for i in range(len(v)):
        sum += v[i] * w[i]
    return sum
```

In [ ]:
```
dotproduct_result = dotproduct(a1, a2)
dotproduct_result
```

We can test the efficiency of this approach to implementing vectors by computing a large dot product many times. (We use the `timeit` function that we imported from the `timeit` library above to return the time in seconds to perform 100,000 repetitions of the dot product computation.)

In [ ]:
```
example_time = timeit('dotproduct(a1, a2)', number=100000, globals=globals())
print(f"It took {example_time:.3} seconds.")
```

As it turns out, performing this vector computation is quite slow -- it probably took several seconds -- because the `for` loop over the list data structure computes sequentially. Instead, we can use a data type engineered especially for such vector and array computations to improve performance. Such data types include Python arrays, `numpy` arrays, and `torch` tensors. The latter are especially designed for the kinds of computations found in machine learning algorithms, so we will use them throughout the course. You can read (a lot) more about tensors in the official documentation.

We construct a couple of one-dimensional tensors for the examples above.

In [ ]:
```
t1 = torch.tensor(a1)
t2 = torch.tensor(a2)
```

Tensor properties

Tensors have four characteristics that are especially useful (but potentially confusing when you first start working with them):

- Componentwise operation: Many operations on tensors work component by component instead of all at once.
- Broadcast: Operations can broadcast individual elements to each component of a tensor.
- Reshaping: Tensors can be reshaped to present the same elements in different configurations.
- Special operations: Tensors have methods implementing certain operations especially efficiently.

We give examples of each.

Componentwise operation

Many operations on tensors work *componentwise*, that is, separately for each component of the tensor, rather than on the tensor all at once.

For instance, when we add two tensors of the same shape with the `+` operator, the summation percolates down to the individual comonents. For example,

```
In [ ]: a3 = [1, 2, 3]
        a4 = [4, 5, 6]
        t3 = torch.tensor(a3)
        t4 = torch.tensor(a4)

        t3 + t4
```

This is quite different from, say, lists, which perform a completely different operation – concatenation – when summed with the `+` operator.

```
In [ ]: a3 + a4
```

Similarly, we can compute the elementwise product of two tensors of the same shape.

```
In [ ]: t3 * t4
```

Broadcast

Related, adding a scalar to a tensor "broadcasts" the scalar addition operation to each element.

In [ ]:
```
print(t3 + 5)
print(5 + t3)
```

Again, compare with how lists work (or actually, don't work). Try uncommenting and running the cell below.

In [ ]:
```
# a3 + 5
```

Reshaping

Finally, tensors can be reshaped so that their elements appear in a different configuration. The `view` method is often used to carry out the reshaping. For instance, we start with the following 3 by 4 tensor.

In [ ]:
```
t5 = torch.tensor([[11, 12, 13, 14],
                   [21, 22, 23, 24],
                   [31, 32, 33, 34]])
```

We can tell that `t5` is a 3 by 4 tensor using the `shape` method.

In [ ]:
```
t5.shape
```

We can view the elements as a 4 by 3 tensor, or a 2 by 2 by 3 tensor, or a 3 by 1 by 4 tensor.

In [ ]:
```
print(t5.view(4, 3))
print(t5.view(2, 2, 3))
print(t5.view(3, 1, 4))
```

Special operations

Tensors have a large set of methods defined on them that work especially efficiently, for instance, taking the sum of the elements, or the minimum.

In [ ]:
```
t5.sum()
```

In [ ]:
```
t5.min()
```

The `min` method takes the minimum over all the elements in the tensor. But we often want to take the minimum or maximum with respect to a particular dimension, returning a tensor of these optima. For example, given a two-dimensional tensor, to find the minimum for each row, we can take the `min` with respect to the second dimension. (We refer to the row dimension as `1` since dimensions are 0-indexed.)

For some intuition (especially helpful as the number of dimensions gets higher and you lose the simple notion of "rows" and "columns"), think of the `.min` by dimension operation as collapsing that dimension - for example, if you take the `.min(1)` of a $4 \times 5 \times 6$ tensor, your output will be a 4×6 tensor.

The `min` method called in this way returns a tensor of the various minimum values, along with the indices at which these minima occurred. We can extract the `values` if we are interested only in those. (Feel free to take a look at the `indices` part of the return value to see what that looks like. Does its value make sense?)

```
In [ ]: t5.min(1).values
```

Define a function to find the maximum values for each column of a two-dimensional tensor.

```
In [ ]: # TODO -- Implement a function to return the max value of each column, store
        def max_col(v):
            ...
```

```
In [ ]: grader.check("max_val")
```

Vectorized dot product

Using these tensor techniques, and noting the examples above, reimplement a version of `dotproduct` that has no `for` loops. **Hint:** Your code should be *very short*.

```
In [ ]: # TODO -- Implement a vectorized dot product, which should be much faster.
        def dotproduct_v(v1, v2):
            ...
```

```
In [ ]: grader.check("dotprod")
```

This vectorized version should be *much* faster, perhaps a couple of orders of magnitude.

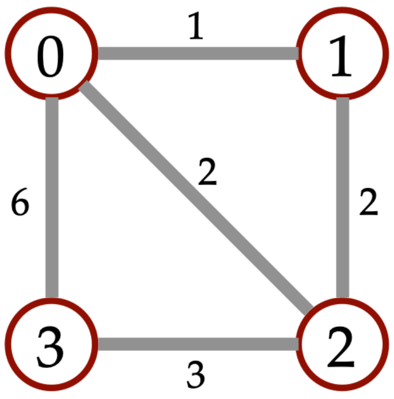
```
In [ ]: example_time_v = timeit('dotproduct_v(t1, t2)', number=100000, globals=globa
        print(f"It took {example_time_v:.3} seconds.")
```

Vectorized computations over multidimensional tensors

As you saw above, tensors aren't limited to one dimension, and these same vectorization tricks apply to multidimensional tensors. In fact, because of vectorization, we can specify computations over multidimensional tensors that look like the kinds of things you've seen in linear algebra.

An example: all-paths shortest path

As a concrete example to give you some practice, consider the algorithm for computing shortest paths in a graph of n nodes. We'll represent the graph as an $n \times n$ matrix A where A_{ij} is the distance from node i to node j. (Thus, this is a directed graph, and the distances needn't be symmetric.) We'll assume that the distance from a node to itself is zero. Here's an example, with distances that happen to be symmetric:



In [ ]:
```
from math import inf
distances = torch.tensor(
              [[0, 1,   2, 6],
               [1, 0,   2, inf],
               [2, 2,   0, 3],
               [6, inf, 3, 0]])
```

(We use `inf` for an infinite distance, that is, for nodes that are not connected with an edge.)

In this graph, the distance from node 0 to node 3 is 6, but by going through node 2, we can shorten the path to 5. In general, we define the *minplus* operation ($\star$) on two square matrices A (of shape $m \times n$) and B (of shape $n \times p$):

$$(A \star B)_{ij} = \min_k A_{ik} + B_{kj}$$

As a special case, if A and B are two distance graphs over the same nodes (but perhaps with different distances), then $A \star B$ is the graph that shows the best way to get from one node to another by traversing an edge from the first graph A and then an edge from the second graph B.

Here's an implementation of this operation `minplus` using `for` loops.

In [ ]:
```
def minplus_loop(A, B):
    (arows, acols), (brows, bcols) = A.size(), B.size()
    assert acols == brows

    R = torch.zeros(arows, bcols)
    for i in range(arows):
        for j in range(bcols):
            min = torch.tensor(inf)
            for k in range(acols):
                if A[i,k] + B[k,j] < min:
                    min = A[i,k] + B[k,j]
            R[i,j] = min
    return R
```

Let's test it out on a small rectangular matrix.

In [ ]:
```
test = torch.tensor([[1, 2, 3], [4, 5, 6]])
minplus_loop(test, test.transpose(0, 1))
```

Using this, we can compute some better ways of getting among the nodes in the `distances` graph. For paths of length at most 2, we can compute

In [ ]:
```
minplus_loop(distances, distances)
```

Notice that in this graph, the distance from node 0 to node 3 is now only 5 (not 6), and there are also now paths between nodes 1 and 3.

We can compute the minimum distance between any two nodes by repeating this minplus process until no further distance reductions are possible and the graph has reached a stable point (the so-called "fixpoint"). We return as a Python tuple both the fixpoint graph and the number of rounds of `minplus` that were needed to reach it.

In [ ]:
```
def minplus_fp(X):
    rounds = 0
    lastY = torch.zeros_like(X)
    Y = X
    while not(torch.equal(Y, lastY)):
        lastY = Y
        Y = minplus_loop(Y, Y)
        rounds += 1
    return Y, rounds
```

In [ ]:
```
minplus_fp(distances)
```

It turns out that after just the two rounds, the fixpoint is reached.

> **Digression:** The complexity of `minplus` as implemented is $O(n^3)$, and the fixpoint computation may need up to $\log n$ calls to `minplus` to converge, so the overall complexity is $O(n^3 \log n)$. More efficient algorithms are known, especially the Floyd-Warshall algorithm for the all-paths versions and Dijkstra's algorithm for the single-source version. But algorithmic efficiency is not our main aim here.

Let's try a bigger example, a graph with 10 nodes.

In [ ]:
```
def random_square_tensor(size):
    X = torch.rand(size, size)
    for i in range(size):
        X[i, i] = 0
    return X

X = random_square_tensor(10)
X
```

In [ ]:
```
minplus_fp(X)
```

Now let's talk about the "loop"-y implementation of the `minplus` function. If we're a little cleverer, we can use list comprehensions to hide the computation of the minimum, but we're still doing the whole computation sequentially.

In [ ]:
```
def minplus_loop2(A, B):
    (arows, acols), (brows, bcols) = A.size(), B.size()
    assert acols == brows

    R = torch.zeros(arows, bcols)
    assert acols == brows
    for i in range(arows):
        for j in range(bcols):
            R[i,j] = min([A[i,k] + B[k,j] for k in range(acols)])
    return R

minplus_loop2(test, test.transpose(0,1))
```

The `torch`-y way to perform this computation is to rely on vectorized computations. Doing so is a bit tricky however. We need to reshape the matrices a bit. We start by turning the two-dimensional A matrix from a $m \times n$ matrix (rows by columns, which may differ in the general case) into a three-dimensional $m \times 1 \times n$ matrix. Here's the result of that operation on the 4×4 `distances` matrix, using the torch `view` method. We'll refer to the reshaped matrix below as the A matrix henceforth. It will now be a tensor with 4 elements, each of which has 1 element, each of which has 4 elements (that are scalars).

In [ ]:
```
A = distances.view(4, 1, 4)
A
```

Now each row in the matrix contains a single element, a vector that corresponds to the column in the original. We'll do a similar operation on B of shape $n \times p$ (again, the `distances` matrix in our example), but this time reshaping the matrix to be $1 \times p \times n$. We'll refer to the reshaped matrix below as B. It will now be a tensor with just 1 element, and that element contains 4 elements, each of which contains 4 elements.

In [ ]:
```
B = distances.view(1, 4, 4)
B
```

Now if we add these two matrices componentwise, what do we get? Each of the four 1 × 4 elements in A corresponds to the same single 4 × 4 element in B, so they'll be added componentwise; that single element in B will be "broadcast" to each of the rows in A. Thus, the first element in A (`[[0., 1., 2., 6.]]` in the example) is to be added to the single element in B (`[[0., 1., 2., 6.], [1., 0., 2., inf], [2., 2., 0., 3.], [5., inf, 3., 0.]]` in the example). How does this work? Going one more level in, we now have that the single 4-element element `[0., 1., 2., 6.]` in

A corresponds to (and will be broadcast to) each of the four 4-element elements in the single element in B (the first of which is also `[0., 1., 2., 6.]`). Now, we've reached a set of elements that are all the same size as the element to be broadcast, so they are added elementwise, yielding `[0., 2., 4., 12.]` as the first summed 4-element element. The same thing happens for each of the other three elements in the reshaped B matrix element. Zooming out one step, we repeat this procedure for each of the other three 1 $\times$ 4 elements of A, broadcasting them onto the single 4 $\times$ 4 element of B. When all is said and done, we'll have an $m \times p \times n$ matrix. Each outermost element of the summed matrix corresponds to a broadcast of a 1 $\times$ 4 element of A onto the 4 $\times$ 4 element of B.

As another way to understand this process, we want to compute the elementwise sum of a matrix A of size 4 $\times$ 1 $\times$ 4 with another matrix B of size 1 $\times$ 4 $\times$ 4. If both matrices were of size 4 $\times$ 4 $\times$ 4, there would be no problem. However, the first dimension of A is of size 4 whereas the first dimension of B is of size 1. Therefore, we repeat B 4 times along the first dimension, expanding it into a matrix of size 4 $\times$ 4 $\times$ 4. Now the first dimension works out, but the second dimension of A is of size 1 whereas the second dimension of B is of size 4, so we also need to repeat A 4 times along the second dimension, expanding it into a matrix of 4 $\times$ 4 $\times$ 4. Now both A and B have been expanded into size 4 $\times$ 4 $\times$ 4, and we can directly compute their elementwise sum. This hidden expanding operation is exactly the broadcasting we've referred to above.

```
In [ ]: summed = distances.view(4, 1, 4) + distances.view(1, 4, 4)
        summed
```

How does this matrix translate to the $A_{ik} + B_{kj}$ additions we did before? To illustrate by example (indexing starting from 0), let's think about $(A \star B)_{1,2} = \min_k A_{1,k} + B_{k,2}$, where A and B now refer to the original 4 $\times$ 4 `distances` matrix that we started with. The $A_{1,k}$s would be found in the second element of `summed`, which is where we broadcasted the second (index 1) row of A. Since this matrix is symmetric, the $B_{k,2}$'s would be equivalent to the $B_{2,k}$'s, which would be found in the second row of each element of `summed`. Taking these two pieces of information together, we then look at the second row of the second element of `summed`. Indeed, $A_{1,0} + B_{0,2} = A_{1,0} + B_{2,0}$ would be the first element in the third row of the second element of `summed`, and this is true for each k. This tells us that to find our `minplus` result, we just need to find the minimum of each row of every element of `summed` (using the `min` method), and there are $4 \cdot 4$ such rows. Note that this solution only works for symmetric matrices - for non-symmetric matrices, you'll need to transpose `distances` before you reshape it into a $1 \times 4 \times 4$ tensor.

Calculate the result of the `minplus` by performing appropriate operations on `summed` to yield a tensor that should be identical to the `minplus_loop(distances, distances)` example above.

```
In [ ]: #TODO
        result = ...
        result
```

```
In [ ]: grader.check("minplus_v_example")
```

Now that you've seen an example of how the matrices can be reshaped and operated on to implement the `minplus` operation, write a function `minplus_v` (a "vectorized" version of `minplus_loop`) that computes the minplus of two rectangular matrices without using any looping constructs.

```
In [ ]: #TODO
        def minplus_v(A, B):
            ...
```

```
In [ ]: grader.check("minplus_v")
```

Finally, we'll use your vectorized `minplus_v` to implement a vectorized fixpoint calculation, and test the relative speeds.

```
In [ ]: def minplus_vfp(X):
            rounds = 0
            lastY = torch.zeros_like(X)
            Y = X
            while not(torch.equal(Y, lastY)):
                lastY = Y
                Y = minplus_v(Y, Y)
                rounds += 1
            return Y, rounds
```

```
In [ ]: minplus_vfp(distances)
```

```
In [ ]: example = random_square_tensor(20)

        timeit('minplus_fp(example)', number=10, globals=globals())
```

```
In [ ]: timeit('minplus_vfp(example)', number=10, globals=globals())
```

If you've implemented `minplus_v` correctly, the efficiency difference should be striking. This kind of engineered improvement is the difference between a computation taking a day and one taking a minute.

Submission Instructions

This lab should be submitted to Gradescope at http://go.cs187.info/lab0-1-submit, which will be made available some time before the due date.

Make sure that you have passed all public tests by running `grader.check_all()` below before submitting. Note that there are hidden tests on Gradescope, the results of which will be revealed after the submission deadline.

End of lab 0-1

To double-check your work, the cell below will rerun all of the autograder tests.

```
In [ ]: grader.check_all()
```