1 Bellman-Ford Algorithm

The Bellman-Ford algorithm is a way to find single source shortest paths in a graph with negative edge weights (but no negative cycles). The second for loop in this algorithm also detects negative cycles.

The first for loop relaxes each of the edges in the graph $n-1$ times. We claim that after $n-1$ iterations, the distances are guaranteed to be correct.

Overall, the algorithm takes $O(mn)$ time.

Algorithm 1: Bellman Ford Algorithm

```
$\forall v \in V, d[v] \leftarrow \infty$ // set initial distance estimates
//optional: set $\pi(v) \leftarrow$nil for all $v$, $\pi(v)$ represents the predecessor of $v$
$d[s] \leftarrow 0$ // set distance to start node trivially as 0
for $i$ from $1 \rightarrow n-1$ do
    for $(u,v) \in E$ do
        $d[v] \leftarrow \min\{d[v], d[u] + w(u,v)\}$ // update estimate of $v$
        // optional - if $d[v]$ changes, then $\pi(v) \leftarrow u$
// Negative Cycle Step
for $(u,v) \in E$ do
    if $d[v] > d[u] + w(u,v)$ then
        return "Negative Cycle"; // negative cycle detected
return $d[v]$ $\forall$ $v \in V$
```

1.1 Correctness

To prove correctness, we reformulate the algorithm in a dynamic-programming-like way and prove the distances are correct by induction. The pseudocode below is equivalent to the pseudocode above, except that the code above reuses the same distance array in all iterations of the loop, whereas the code below uses a different one every time. We wouldn't want to actually use the code below (it wastes a lot of space), but for correctness-proof purposes, it is convenient to have the output of each iteration labeled with the iteration number, instead of having to keep saying "the distance estimate on iteration k".

Algorithm 2: Equivalent formulation of the Bellman-Ford Algorithm

```
$\forall v \in V$ and $\forall k, d_k[v] \leftarrow \infty$ // set initial distance estimates
$d_k[s] \leftarrow 0$ // set distance to start node trivially as 0
for $k$ from $1 \to n-1$ do
    for $(u,v) \in E$ do
        $d_k[v] \leftarrow \min\{d_{k-1}[v], d_{k-1}[u] + w(u,v)\}$ // update estimate of $v$
// Negative Cycle Step
for $(u,v) \in E$ do
    if $d_{n-1}[v] > d_{n-1}[u] + w(u,v)$ then
        return "Negative Cycle"; // negative cycle detected
return $d_{n-1}[t]\ \forall\, t \in V$
```

We prove two things about this new algorithm:

1. Bellman-Ford detects negative cycles, i.e. if there is a negative cycle reachable from the source s, then for some edge (u, v), $d_{n-1}(v) > d_{n-1}(u) + w(u, v)$.
2. If the graph has no negative cycles, then the distance estimates on the last iteration are equal to the true shortest distances. That is, $d_{n-1}(v) = \delta(s, v)$ for all vertices v.

1.1.1 Bellman-Ford detects negative cycles

Suppose $v_0 \to v_1 \to \cdots \to v_k$ is a negative cycle reachable from s, where $v_0 = v_k$. Formally, this means we have

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0$$

We proceed by contradiction. Suppose that we have $d_{n-1}(v_i) \leq d_{n-1}(v_{i-1}) + w(v_{i-1}, v_i)$ for all $i = 1, \ldots, k$. Then summing up the inequality for each vertex on the cycle, we get

$$\sum_{i=1}^{k} d_{n-1}(v_i) \leq \sum_{i=1}^{k} d_{n-1}(v_{i-1}) + \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

Observe that the first two terms are the same. That's because $v_0 = v_k$, so

$$\sum_{i=1}^{k} d_{n-1}(v_{i-1}) = \sum_{i=0}^{k-1} d_{n-1}(v_i) = d_{n-1}(v_0) + \sum_{i=1}^{k-1} d_{n-1}(v_i) = \sum_{i=1}^{k} d_{n-1}(v_i)$$

Because those two terms are the same, we can cancel them out to get

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) \geq 0$$

contradicting the supposition that $v_0 \to v_1 \to \cdots \to v_k$ is a negative cycle.

1.1.2 Bellman-Ford correctly computes distances

We want to show that if the graph has no negative cycles, then $d_{n-1}(v) = \delta(s, v)$ for all vertices v. By induction on k, we will prove that $d_k(v)$ is the minimum weight of a path from s to v that uses $\leq k$ edges.

This will show that $d_{n-1}(v)$ is the minimum weight of a path from s to v that uses $\leq n-1$ edges. This is just the weight of the shortest path, because the fact that the graph has no negative cycles means there must always be a shortest path with no repeated vertices. (If the shortest path had a repeated vertex, we could splice out the cycle from the path and get a path that was equally short or shorter.)

Base case: If $k = 0$, then $d_k(v) = 0$ for $v = s$, and ∞ otherwise. So the claim is satisfied because there is a path of length 0 from s to itself, and no path of length 0 from s to any other vertex.

Inductive step: Suppose that for all vertices u, $d_{k-1}(u)$ is the minimum weight of a path from s to u that uses $\leq k-1$ edges.

If $v \neq s$, let P be a shortest simple path from s to v with $\leq k$ edges, and let u be the node just before v on P. Let Q be the path from s to u. Then path Q has $\leq k-1$ nodes and must be a shortest path from s to u on $k-1$ edges (or else we could replace Q with a shorter path, contradicting the fact that P is a shortest simple path on $\leq k$ edges). By the inductive hypothesis, $w(Q)$ (i.e. the weight of path Q) is $d_{k-1}(u)$.

In iteration k, we update $d_k(v) = \min(d_{k-1}(v), d_{k-1}(u) + w(u, v))$. We know that $d_{k-1}(u) + w(u, v) = w(Q) + w(u, v) = w(P)$, which shows that $d_k(v) \leq w(P)$. Furthermore, $d_{k-1}(v)$ is the length of a shortest simple path from s to v on at most $k-1$ edges, which must be at least as large as $w(P)$, since P has more edges to work with.

Therefore, $d_k(v) = w(P)$ is the minimum weight of a path from s to v that uses $\leq k$ edges.

2 Amortized analysis

Amortized analysis is a way of analyzing data structures so that we get tighter bounds on the runtime of data structure operations.

In amortized analysis, we average the time required to perform a sequence of data-structure operations over all the operations performed. With amortized analysis, we can show that if we average over a sequence of operations, the average cost of an operation is small, even though a single operation within the sequence might be expensive.

Specifically, we assign an **amortized cost** to each operation, that must satisfy the property that in any sequence of operations, the sum of the amortized costs must be at least as large as the sum of the true costs. The notion of amortized cost allows us to do things like say that some operations have zero amortized cost (and have those costs "covered" by other operations that have larger amortized costs).

Note that amortized analysis is not the same as average-case analysis! There is no probability here. Amortized analysis provides a guarantee on the average performance of each operation in the worst case scenario.

2.1 Aggregate analysis

In aggregate analysis, we determine an upper bound $T(n)$ on the total cost of a sequence of n operations. Then the average cost per operation is $T(n)/n$, and we say the amortized cost of any operation is this average cost.

2.1.1 Stack operations

We will first consider a contrived example. Consider a stack S. The stack has two fundamental operations

- `Push(S, x)` which pushes x onto S
- `Pop(S)` which pops the top of S and returns the popped object

Both of these operations take $O(1)$ time, so let's just say (for this example) that the cost of each operation is 1. This means the total cost of any sequence of n `Push` and `Pop` operations is n, and the actual running time for n operations is $\Theta(n)$.

Now we add a new operation, `Multipop(S, k)`, which removes the top $k > 0$ objects of stack S, and pops the entire stack if the stack contains fewer than k objects.

```
MULTIPOP(S, k)
while not STACK-EMPTY(S) and k > 0
    POP(S)
    k = k − 1
```

The total cost of `Multipop` is $\min(s, k)$, where s is the number of items on the stack.

Now suppose we have a sequence of n `Push`, `Pop`, and `Multipop` operations on an initially empty stack. Naively, the total cost of these operations is $O(n^2)$, because each `Multipop` operation costs $O(n)$, and there may be $O(n)$ `Multipop` operations. However, this is a gross overestimate, because the number and size of the `Multipop` operations are limited by the number of `Push` operations that were previously applied to the stack.

Each sequence of n operations on an initially empty stack can cost at most $O(n)$. Why? The total number of pops (including pops within `Multipop`) cannot be larger than the total number of pushes. The number of `Push` operations is at most n, so the total number of operations is at most $O(n)$.

Therefore the average cost of an operation in a sequence is $O(n)/n = O(1)$. We can assign the amortized cost of each operation to be the average cost.

2.1.2 Incrementing a binary counter

We want to implement a binary counter that counts upwards from 0, i.e.

`00000000 => 00000001 => 00000010 => 00000011 => 00000100 => ...`

We can implement this as an array $A[0..k-1]$ of bits that can either be 0 or 1. We interpret the elements of the array as the binary number $x = \sum_{i=0}^{k-1} A[i] \cdot 2^i$.

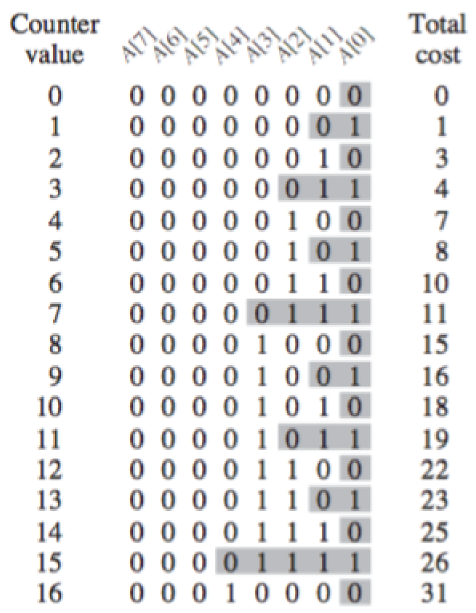
We define the `Increment` function on this counter, as follows:

```
INCREMENT(A)
1  $i = 0$
2  while $i < A.length$ and $A[i] == 1$
3      $A[i] = 0$
4      $i = i + 1$
5  if $i < A.length$
6      $A[i] = 1$
```

The cost of the `Increment` operation depends on the number of bits flipped. In the worst case scenario (applying `Increment` to `11111111`), it flips all the bits, which takes time $\Theta(k)$. Thus we can say that a sequence of n `Increment` operations on an initially zero counter takes time $O(nk)$.

However, this is once again a gross overestimate, because steps where all the bits are flipped are few and far between, and it is much more likely that only a couple bits are flipped.

You will notice that $A[0]$ is flipped every time `Increment` is called. $A[1]$ is flipped every other time `Increment` is called. $A[2]$ is flipped every fourth time, etc. So a sequence of n operations causes $A[0]$ to flip n times, $A[1]$ to flip $\lfloor n/2 \rfloor$ times, $A[2]$ to flip $\lfloor n/4 \rfloor$ times, etc.



Counter value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Total cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

Figure 17.2 An 8-bit binary counter as its value goes from 0 to 16 by a sequence of 16 INCREMENT operations. Bits that flip to achieve the next value are shaded. The running cost for flipping bits is shown at the right. Notice that the total cost is always less than twice the total number of INCREMENT operations.

In general, bit $A[i]$ flips $\lfloor n/2^i \rfloor$ times in a sequence of n `Increment` operations on an initially zero counter. So the total number of flips is

$$\sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n$$

Therefore, the worst case time for n `Increment` operations is $O(n)$, and the average cost of each operation (which we can set the amortized cost to) is $O(1)$.

2.2 The accounting method

In this method we may assign different amortized costs to different operations. The amortized costs do not necessarily have to correspond to the actual costs of the operations. We just need to make sure that for any sequence of operations, the sum of the amortized costs is at least as large as the sum of the actual costs.

2.2.1 Stack operations

Recall that the actual costs of the operations were

Operation	Cost
Push	1
Pop	1
Multipop	$\min(k, s)$

We may assign amortized costs as follows:

Operation	Cost	Amortized cost
Push	1	2
Pop	1	0
Multipop	$\min(k, s)$	0

Note that all of these amortized costs are constant, so all of the operations take amortized $O(1)$ time.

Suppose we use a dollar bill to represent each unit of cost. When we push an item onto the stack, we are charged 2 dollars. One of the dollars is used to pay for the actual cost of the push, and the other dollar is stored as "credit" on the item and can be used to pay other costs later.

Now when we pop an item off the stack, we charge the operation nothing, and we pay for its actual cost using the "credit" stored in the stack. Since each item has a dollar of credit stored on it, we can use that dollar to pay for the cost of popping the item.

`Multipop` operations can also be charged nothing. Every time we pop an item during the execution of `Multipop`, we use the dollar of credit stored on that item, and use it to pay the cost of a `Pop` operation. Each item in the stack has a dollar of credit stored on it, so all `Pop` and `Multipop` operations have already been paid for.

Since each item in the stack has 1 dollar of credit stored on it, and the stack always has a nonnegative number of items on it, the amount of credit is always nonnegative. So for any sequence of n operations, the total amortized cost is an upper bound on the total actual cost. Since the total amortized cost is $O(n)$, so is the total actual cost.

2.2.2 Incrementing a binary counter

Here let us say that setting a bit to 1 has an amortized cost of 2 dollars, and setting a bit to 0 has an amortized cost of 0 dollars. When we set the bit to 1, we use 1 dollar to pay for the cost of setting the bit, and the other dollar is stored as "credit" for when we flip the bit back to 0 later. At any point in time, every 1 in the counter has a dollar of credit on it, so we can always use the credit to pay for resetting the bit to 0.

In the `Increment` function, all of the `A[i] = 0` commands in the while loop are paid for with credit. (For ease of illustration, we are being kind of handwavy with regard to the costs of operations here, but if we want we can also tweak the relative amortized costs a bit so that the `i = i + 1` lines are paid for with credit as well.) So we only really have to worry about the `A[i] = 1` command. There is only one of these commands, and it costs 2 dollars. So the `Increment` function costs constant amortized time.

For n `Increment` operations, the total amortized cost is $O(n)$. Since the number of 1s in the counter never becomes negative, the amount of credit stays nonnegative at all times, so the total actual cost must also be at most $O(n)$.

2.3 The potential method

Instead of storing credits on specific objects in the data structure, we can represent "credit" as "potential energy" that is associated with the data structure as a whole.

Let

- D_0 be the initial data structure
- c_i be the cost of the ith operation for $i = 1, \ldots, n$
- D_i be the data structure that results after applying the ith operation to D_{i-1}.
- Φ be the **potential function** (which we invent ourselves), so that $\Phi(D_i)$ is the potential associated with data structure D_i. Note that $\Phi(D_i)$ is a real number.

We define the amortized cost $\hat{c}_i$ to be

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

i.e. the amortized cost is the actual cost plus the change in potential that resulted from the operation.

This means the total amortized cost of the n operations is

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$$

because the other potential terms cancel out.

As long as we define our potential function in such a way that $\Phi(D_n) \geq \Phi(D_0)$, the total amortized cost gives an upper bound on the total actual cost. One way to do this is by defining $\Phi(D_0)$ to be 0 and showing that $\Phi(D_i) \geq 0$ for all i.

Intuitively, if the potential difference $\Phi(D_i) - \Phi(D_{i-1})$ of the ith operation is positive, then we have "overcharged" the ith operation, which builds up extra "credit", or "potential", in the data structure. Whereas if the potential difference is negative, then we have "undercharged" the operation, and the decrease in the potential pays for the actual cost of the operation.

2.3.1 Stack operations

Define the potential function Φ on a stack to be the number of objects in the stack. (This is equivalent to our credit scheme, where we had one credit for each item in the stack.)

D_0 is the empty stack, and $\Phi(D_0) = 0$. Furthermore, since there are never negative items in the stack, we always have $\Phi(D_i) \geq 0 = \Phi(D_0)$. Therefore, the total amortized cost is always an upper bound on the total actual cost.

Suppose the ith operation is a `Push` operation (on a stack containing s objects). Then the amortized cost is $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + (s + 1) - s = 2$ (which is equal to the amortized cost we got previously).

The amortized cost of a `Pop` operation (on a stack containing s objects) is $1+(s-1)-s = 0$.

If the ith operation is `Multipop(S, k)`, where $k' = \min(k, s)$ objects are popped off the stack, the actual cost of the operation is k' and the potential difference is $-k'$. So the amortized cost is $k' - k' = 0$.

Thus, the amortized cost of each operation is $O(1)$, and the total amortized cost of n operations is $O(n)$. So the actual cost of n operations is $O(n)$.

2.3.2 Incrementing a binary counter

Define the potential of the counter $\Phi(D_i)$ to be the number of 1's in the counter **after** the ith operation.

Suppose the ith `Increment` operation resets t_i bits. The actual cost of the operation is $\leq t_i + 1$ (because in addition to resetting t_i bits, it sets at most one bit to 1).

If $\Phi(D_i) = 0$, then the ith operation resets all k bits, so $\Phi(D_{i-1}) = t_i = k$. If $\Phi(D_i) > 0$, then $\Phi(D_i) = \Phi(D_{i-1}) - t_i + 1$. Either way, $\Phi(D_i) \leq \Phi(D_{i-1}) - t_i + 1$, so the potential difference for the ith operation is $\leq 1 - t_i$.

So the amortized cost is $\hat{c}_i + \Phi(D_i) - \Phi(D_{i-1} \leq (t_i + 1) + (1 - t_i) = 2$.

Since $\Phi(D_0) = 0$ (when the counter starts at 0) and $\Phi(D_i) \geq 0$ for all i, the total amortized cost of n operations is an upper bound on the total actual cost, so the actual cost is $O(n)$.

Note that if the counter does not start at 0, $\Phi(D_0) \neq 0$. But we can still say that

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$$

i.e.

$$\begin{aligned} \sum_{i=1}^{n} c_i &= \sum_{i=1}^{n} \hat{c}_i - \Phi(D_n) + \Phi(D_0) \\ &\leq 2n - \Phi(D_n) + \Phi(D_0) \end{aligned}$$

Remember that $\Phi(D_0) \leq k$, since there are only k bits in the counter total. So as long as $k = O(n)$, the total actual cost is $O(n)$. Therefore, if we execute at least $n = \Omega(k)$ `Increment` operations, the total actual cost is $O(n)$, regardless of whether the counter starts from 0.