Today we will go over loop and recursion invariants.

0.1 Induction (useful for understanding loop invariants)

We can use induction when we want to show a statement is true for all positive integers n. (Note that this is not the only situation in which we can use induction, and that induction is not (usually) the only way to prove a statement for all positive integers.)

To use induction, we prove two things:

- **Base case:** The statement is true in the case where $n = 1$.
- **Inductive step:** If the statement is true for $n = k$, then the statement is also true for $n = k + 1$.

This actually produces an infinite chain of implications:

- The statement is true for $n = 1$
- If the statement is true for $n = 1$, then it is also true for $n = 2$
- If the statement is true for $n = 2$, then it is also true for $n = 3$
- If the statement is true for $n = 3$, then it is also true for $n = 4$
- $\ldots$

Together, these implications prove the statement for all positive integer values of n. (It does not prove the statement for non-integer values of n, or values of n less than 1.)

Example: Prove that $1 + 2 + \cdots + n = n(n + 1)/2$ for all integers $n \geq 1$.

Proof: We proceed by induction.

Base case: If $n = 1$, then the statement becomes $1 = 1(1 + 1)/2$, which is true.

Inductive step: Suppose the statement is true for $n = k$. This means $1 + 2 + \cdots + k = k(k+1)/2$. We want to show the statement is true for $n = k+1$, i.e. $1+2+\cdots+k+(k+1) = (k + 1)(k + 2)/2$.

By the induction hypothesis (i.e. because the statement is true for $n = k$), we have $1 + 2 + \cdots + k + (k + 1) = k(k + 1)/2 + (k + 1)$. This equals $(k + 1)(k/2 + 1)$, which is equal to $(k + 1)(k + 2)/2$. This proves the inductive step.

Therefore, the statement is true for all integers $n \geq 1$.

0.1.1 Strong induction

Strong induction is a useful variant of induction. Here, the inductive step is changed to

- **Base case:** The statement is true when $n = 1$.

- **Inductive step:** If the statement is true for all values of $1 \leq n < k$, then the statement is also true for $n = k$.

This also produces an infinite chain of implications:

- The statement is true for $n = 1$
- If the statement is true for $n = 1$, then it is true for $n = 2$
- If the statement is true for both $n = 1$ and $n = 2$, then it is true for $n = 3$
- If the statement is true for $n = 1$, $n = 2$, and $n = 3$, then it is true for $n = 4$
- ...

Strong induction works on the same principle as weak induction, but is generally easier to prove theorems with.

Example: Prove that every integer n greater than or equal to 2 can be factored into prime numbers.

Proof: We proceed by (strong) induction.

Base case: If $n = 2$, then n is a prime number, and its factorization is itself.

Inductive step: Suppose k is some integer larger than 2, and assume the statement is true for all numbers $n < k$. Then there are two cases:

Case 1: k is prime. Then its prime factorization is just k.

Case 2: k is composite. This means it can be decomposed into a product xy, where x and y are both greater than 1 and less than k. Since x and y are both less than k, both x and y can be factored into prime numbers (by the inductive hypothesis). That is, $x = p_1 \dots p_s$ and $y = q_1 \dots q_t$ where $p_1, \dots, p_s$ and $q_1, \dots, q_t$ are prime numbers.

Thus, k can be written as $(p_1 \dots p_s) \cdot (q_1 \dots q_t)$, which is a factorization into prime numbers.

This proves the statement.

1 Loop invariants (useful for Problem 3)

1.1 Merge

As an example of a loop invariant, we will prove the correctness of the merge step in Merge-Sort, which combines two sorted arrays into a single sorted array.

How does Merge work? Assume you are given two sorted arrays, L and R, and you want to produce a sorted array that has all the elements in both L and R.

Then you keep a pointer to the first element of each array. You know that the first element of the final array will either be the first element of L or the first element of R. So you choose the lowest one, and copy it to the result array. Then you increment the corresponding pointer so it points to the second element in that array instead of the first.

Now say we picked from array L in the first step. Then the second-lowest element in the result array either has to be the second element of L, or the first element of R. So once again, we choose the lowest one, and copy it to the result array, and increment the corresponding pointer. We keep doing this until we run out of elements in both the arrays.

Here is the pseudocode for Merge:[1]

```
Merge(L, R):
    m = length(L) + length(R)
    S = empty array of size m
    i = 1; j = 1
    for k = 1 to m:
        if L[i] <= R[j]:
            S[k] = L[i]
            i = i + 1
        else: (L[i] > R[j])
            S[k] = R[j]
            j = j + 1
    return S
```

Example: Suppose we are merging $L = [2, 4, 5, 7]$ with $R = [1, 2, 3, 6, 9, 11]$. We start with $i = 1$ and $j = 1$. Observing that $R[1] < L[1]$, we let $S[1] = 1$ be the first element of the result array, and set $j = 2$. Now $L[1] \leq R[2]$, so we set $S[2] = 2$, and $i = 2$. Now $R[2] < L[2]$, so $S[3] = 2$, and $j = 3$, etc.

1.2 Proof of correctness

To prove Merge, we will use **loop invariants**. A loop invariant is a statement that we want to prove is satisfied at the beginning of every iteration of a loop. In order to prove this, we need to prove three conditions:

Initialization: The loop invariant is satisfied at the beginning of the for loop.

Maintenance: If the loop invariant is true before the ith iteration, then the loop invariant will be true before the $i + 1$st iteration.

Termination: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

[1]The astute observer may ask what happens if one of the arrays gets depleted first. This code, as written, would result in a runtime exception, as we try to access an element that is outside the bounds of the array. In order to fix this, we add the "sentinel element" ∞ to the end of both arrays. ∞ will always be larger than any element that is actually in the array, so this will make us always pick from the other array after the first array has been depleted.

Note that the logic here is an application of induction, where you are inducting on the number of iterations of the loop.

The loop invariant we will use for this problem is

Loop invariant: At the start of each iteration k of the for loop, the nonempty part of S contains the $k-1$ smallest elements of L and R, in sorted order. Moreover, $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied to S.

Specifically, here is how to prove the invariant:

Initialization: The loop invariant holds prior to the first iteration of the loop. Here, $i = j = 1$, and S is completely empty. $L[1]$ is the smallest element of L, while $R[1]$ is the smallest element of R, so the initialization step holds.

Maintenance: To see that each iteration maintains the loop invariant, suppose without loss of generality that $L[i] \leq R[j]$. Then $L[i]$ is the smallest element not yet copied to S. The current nonempty part of S consists of the $k-1$ smallest elements, so after the loop is over and $L[i]$ is copied to S, the nonempty part of S will consist of the k smallest elements. Incrementing k (in the for loop update) and i reestablishes the loop invariant for the next iteration.

Termination: At termination, $k = m+1$. By the loop invariant, S contains the m smallest elements of L and R, in sorted order. This is the result that we wanted (i.e. the merging of the two sorted arrays to produce a new sorted array).

2 Recursion invariants (useful for Problem 5)

For recursive algorithms, we may define a recursion invariant. Recursion invariants are another application of induction.

2.1 Exponentiation via repeated squaring

Suppose we want to find 3^n for some nonnegative integer n. The naive way to do it is using a for loop:

```
answer = 1
for i = 1 to n:
    answer = answer * 3
return answer
```

This runs in $O(n)$ time, since the body of the for loop takes $O(1)$ time to execute, and the for loop test is executed $n+1$ times. (This is $n+1$ instead of n because the for loop test also gets executed on the iteration where we break out of the loop.)

However, we can get a faster algorithm if we write this recursively. By that I mean we can write a function that calls itself.

```
Exponentiator(n):
    if n = 0:
        return 1
    else if n mod 2 = 0: (i.e. n is even)
        x = Exponentiator(n / 2)
        return x * x
    else: (if n is odd)
        x = Exponentiator((n - 1) / 2)
        return 3 * x * x
```

Note that it is important to separate this calculation into even and odd cases so that the input to Exponentiator is always an integer. If this algorithm were extended to fractional inputs, it might never terminate.

2.2 Recursion invariant

To prove the correctness of this algorithm, we use a recursion invariant.

Recursion invariant: At each recursive call, Exponentiator(k) returns 3^k.

Base case (initialization): When $k = 0$, Exponentiator(k) returns $1 = 3^0$.

Maintenance: We can divide this into two cases: k is even, and k is odd.

Suppose k is even. Then the algorithm sets x to Exponentiator($k/2$), which by the recursion invariant is $3^{k/2}$. The algorithm then returns $x * x$, which is $3^{k/2} \cdot 3^{k/2} = 3^k$.

Suppose k is odd. Then the algorithm sets x to Exponentiator($(k-1)/2$), which by the recursion invariant is $3^{(k-1)/2}$. The algorithm then returns $3 * x * x$, which is $3 \cdot 3^{(k-1)/2} \cdot 3^{(k-1)/2} = 3^k$. Thus, the maintenance step holds.

"Termination": At the top level of the recursive call, Exponentiator(n) gives 3^n, which is the solution to the problem.