CS173
Induction and Dynamic Programming

Tandy Warnow

September 1, 2018
This week

We will cover

- Basic inductive proof
- Weak induction
- Strong induction - and when it’s needed
- Proving statements about two variables using induction
- Starting to think about dynamic programming
Induction Proofs

The *idea* behind induction is simple. If I have a very contagious cold, and I sneeze then the person to my right will catch the cold; then she will sneeze, and the person to her right will catch it; and then he will sneeze, etc. Eventually everyone (to my right) will catch the cold.
Necessary parts of induction proofs

- Base case
- Inductive Hypothesis, that is expressed in terms of a property holding for some arbitrary value $K$
- Use the inductive hypothesis to prove the property holds for the next value (typically $K + 1$).
- Point out that $K$ was arbitrary so the result holds for all $K$.
- Optional: say “Q.E.D.”
Weak Induction vs. Strong Induction

- Weak Induction asserts a property $P(n)$ for one value of $n$ (however arbitrary)
- Strong Induction asserts a property $P(n)$ is true for all values of $n$ starting with a base case and up to some final value $K$

Sometimes Strong Induction is needed.
Recurrence relations

Recurrence relations are generally functions defined recursively:

1. \( g(1) = 3 \) and \( g(n) = 3 + g(n - 1) \) for \( n \geq 2 \)
2. \( f(1) = f(2) = 1 \) and \( f(n) = f(n - 1) + f(n - 2) \) for \( n \geq 3 \).

Note that \( f(n) \) depends on \( f(n - 1) \) and \( f(n - 2) \).

Hence you must use strong induction for anything you want to prove about \( f(n) \), but you could have used weak induction for \( g(n) \).

Strong induction is always valid, so practice using it.
Proving properties about Fibonacci numbers

Definition of function $f : \mathbb{Z}^+ \rightarrow \mathbb{Z}$:

\[ f(1) = f(2) = 1 \text{ and } f(n) = f(n-1) + f(n-2) \text{ for } n \geq 3 \]

We wish to prove $f(n) \geq 2n$ for $n \geq 8$.

Let’s check some values...

$f(3) = f(2) + f(1) = 2$

$f(4) = f(3) + f(2) = 2 + 1 = 3$

$f(5) = f(4) + f(3) = 3 + 2 = 5$

$f(6) = f(5) + f(4) = 8$

$f(7) = f(6) + f(5) = 13$

$f(8) = f(7) + f(6) = 21$

$f(9) = f(8) + f(7) = 34$

$f(10) = f(9) + f(8) = 55$

So the statement holds for $n = 8, 9, 10$. 
Proving $f(n) \geq 2n$ for $n \geq 8$

Definition of function $f: \mathbb{Z}^+ \rightarrow \mathbb{Z}$:

- $f(1) = f(2) = 1$ and $f(n) = f(n-1) + f(n-2)$ for $n \geq 3$

We wish to prove $f(n) \geq 2n$ for $n \geq 8$.

The Inductive Hypothesis is

- $P(N): f(n) \geq 2n$ for all integers $n$ between 8 and $N$.

Base cases: we already have shown $P(N)$ is true for $N = 8, 9, 10$.

The Inductive Step is to show that $P(N) \rightarrow P(N + 1)$.

$P(N + 1)$ asserts:

- $f(n) \geq 2n$ for all integers $n$ between 8 and $N + 1$.

$P(N)$ asserts that $f(n) \geq 2n$ for all $n$ between 8 and $N$.

So we only need to show that $P(N)$ implies $f(N + 1) \geq 2(N + 1)$.
Proving \( f(n) \geq 2n \) for \( n \geq 8 \)

Definition of function \( f : \mathbb{Z}^+ \rightarrow \mathbb{Z} \):

\[
\begin{align*}
    f(1) &= f(2) = 1 \\
    f(n) &= f(n-1) + f(n-2) \quad \text{for } n \geq 3
\end{align*}
\]

I.H.: \( P(N) : f(n) \geq 2n \) for all integers \( n \) between 8 and \( N \).

We need to show that \( P(N) \rightarrow P(N+1) \).

Enough to show \( P(N) \rightarrow f(N+1) \geq 2(N = 1) \).

We let \( N \geq 10 \), since our base cases cover \( N = 8, 9, 10 \).

Consider \( f(N+1) \).

By definition, \( f(N+1) = f(N) + f(N-1) \)

By the I.H., \( f(N) \geq 2N \) and \( f(N-1) \geq 2(N-1) \)

Hence \( f(N+1) \geq 2N + 2(N-1) = 4N - 2 \)

Since \( N \geq 8 \), \( 4N - 2 \geq 2(N+1) \)

Hence \( f(N+1) \geq 2(N+1) \)

Since \( N \) was arbitrary, \( P(N) \) is true for all \( N \geq 8 \), and we are done.
Using induction to prove theorems about functions of two variables

Let $f : \mathbb{Z}^+ \times \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$ be defined by

- $f(n, m) = n + m$ if $n = 1$ or $m = 1$,
- $f(n, m) = f(n - 1, m) + f(n, m - 1)$, otherwise

We would like to prove that $f(n, m) \geq n + m$ for all $n \geq 1$ and $m \geq 1$.

Base cases: $n = 1$ or $m = 1$ follows immediately. So we prove the rest by induction.

What is our inductive hypothesis?
Inductive hypothesis

Recall that

- $f(n, m) = n + m$ if $n = 1$ or $m = 1$,
- $f(n, m) = f(n - 1, m) + f(n, m - 1)$, otherwise

What happens if we try to do induction on $n$?
Recall that

- \( f(n, m) = n + m \) if \( n = 1 \) or \( m = 1 \),
- \( f(n, m) = f(n - 1, m) + f(n, m - 1) \), otherwise

We can't do induction on \( n \) because \( f(n, m) \) depends on \( f(n, m - 1) \).

We also can't do induction on \( m \) because \( f(n, m) \) depends on \( f(n - 1, m) \).
Recall that

- \( f(n, m) = n + m \) if \( n = 1 \) or \( m = 1 \),
- \( f(n, m) = f(n-1, m) + f(n, m-1) \), otherwise

We need a value that goes down... so that \( f(n, m) \) depends on values to which the inductive hypothesis can be applied.

What value goes down?
Recall that

- $f(n, m) = n + m$ if $n = 1$ or $m = 1$,
- $f(n, m) = f(n - 1, m) + f(n, m - 1)$, otherwise

The sum of the parameters goes down! So, our inductive hypothesis will be:

$$f(n, m) \geq n + m$$ for all positive integers $n, m$ with $n + m \leq K$
The base case

Recall $f : \mathbb{Z}^+ \times \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$, and the inductive hypothesis is:

$$P(K) : f(n, m) \geq n + m \text{ for all positive integers } n, m \text{ with } n + m \leq K$$

The smallest value for $n + m$ is 2; hence, the base case is $K = 2$. When $K = 2$, $n = m = 1$ and the statement holds.
Finishing the Induction Proof

Recall $f : \mathbb{Z}^+ \times \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$, and the inductive hypothesis is:

$$P(K) : f(n, m) \geq n + m \text{ for all positive integers } n, m \text{ with } n + m \leq K$$

To finish the induction proof, we need to show $P(K) \rightarrow P(K + 1)$, which is equivalent to showing

$$P(K) \rightarrow \forall n, m \text{ such that } n + m \leq K + 1, \ f(n, m) \geq n + m$$

However, since the I.H. assumes $f(n, m) \geq n + m$ whenever $n + m \leq K$, we only need to show that

- $P(K) \rightarrow f(n, m) \geq n + m \text{ when } n + m = K + 1$. 
Another induction proof, continued

Let \( n, m \) be given so that \( n + m = K + 1 \).

If \( n = 1 \) or \( m = 1 \), then by definition \( f(n, m) = n + m \), and the statement holds.
So assume \( n \geq 2 \) and \( m \geq 2 \), so that

\[
f(n, m) = f(n - 1, m) + f(n, m - 1)
\]

Note that \( n + m = K + 1 \) and so \( n + m - 1 = K \).
Hence we can apply the Inductive Hypothesis to \( f(n - 1, m) \) and \( f(n, m - 1) \).
Therefore,

\[
f(n - 1, m) \geq n + m - 1
\]

and

\[
f(n, m - 1) \geq n + m - 1
\]

Hence

\[
f(n, m) = f(n - 1, m) + f(n, m - 1) \geq 2(n + m - 1)
\]
Another induction proof, continued

So far we have shown that when $n, m$ are both at least 2 and $n + m = K + 1$, then

$$f(n, m) = f(n - 1, m) + f(n, m - 1) \geq 2(n + m - 1)$$

However, a little more arithmetic finishes this!

$$f(n, m) \geq 2(n + m - 1) = n + m + (n + m - 2) \geq n + m$$

since $n + m - 2 \geq 0$.

Since $K$ was arbitrary, the statement holds for all $K \geq 2$, and hence for all pairs of positive integers $n, m$ that sum to $K$. This is what we wanted to prove. Q.E.D.
Summarizing what we did

Recall that we had a recursively defined function $f : \mathbb{Z}^+ \times \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$, defined by

- $f(n, m) = n + m$ if $n = 1$ or $m = 1$,
- $f(n, m) = f(n-1, m) + f(n, m-1)$, otherwise

We wanted to prove that $f(m, n) \geq m + n$ for all positive integers $m, n$.

It is easy to verify this inequality for the case where $m = 1$ or $n = 1$. To prove it true for all $m, n$, we used induction.

But induction must be done for some single parameter.

We used $K = m + n$ as our single parameter.

Our inductive hypothesis was

$$f(n, m) \geq n + m$$

for all positive integers $n, m$ with $n + m \leq K$
Let sets $S_{i,j}$ be defined recursively by

- $S_{1,i} = S_{i,1} = \{i\}$ for all $i \in \mathbb{Z}^+$
- $S_{i,j} = S_{i-1,j} \cup S_{j,i-1} \cup \{ij\}$ for integers $i, j > 1$.

We’d like to understand $S_{i,j}$. But we’ll start by computing a few values.

1. What is $S_{1,2}$?
2. What is $S_{2,1}$?
3. What is $S_{2,2}$?
4. What is $S_{3,2}$?
5. What is $S_{5,9}$?

We aren’t that close to having a closed form solution to $S_{i,j}$. But we at least have the idea that $|S_{i,j}| \geq \min\{i, j\}$. 
To prove $|S_{i,j}| \geq \min\{i, j\}$, we will use induction. What will we induct on?

Answer: We will induct on $K = i + j$. Our inductive hypothesis is:

$$P(K) : \forall i, j \in \mathbb{Z}^+ \text{ such that } i + j \leq K, \ |S_{i,j}| \geq \min\{i, j\}.$$
What we learned

We learned:

▶ The base case is sometimes more than one value.
▶ We learned about the difference between strong and weak induction, and that strong induction is always at least as powerful as weak induction - so we will only use strong induction henceforth.
▶ The inductive hypothesis is sometimes not on an obvious parameter, but on something defined using obvious parameters (like the sum)
▶ Induction can be used to prove properties about recursively defined functions and sets.
▶ Induction proofs are not that difficult!
Running time analyses

Often algorithms are described recursively or using divide-and-conquer. We will show how to analyze the running times by:

- writing the running time as a recurrence relation
- solving the running time
- proving the running time correct using induction
Analyzing running times

To analyze the running time of an algorithm, we have to know what we are counting, and what we mean. First of all, we usually want to analyze the **worst case** running time. This means an upper bound on the total running time for the operation, usually expressed as a function of its input **size**. (For example, the size of a graph could be $n + m$, where $n$ is the number of vertices and $m$ is the number of edges.)
Running time analysis

We are going to count **operations**, with each operation having the same cost:

- I/O (reads and writes)
- Numeric operations (additions and multiplications)
- Comparisons between two values
- Logical operations
Now we begin with algorithms, and analyzing their running times. Let $A$ be an algorithm which has inputs of size $n$, for $n \geq n_0$. Let $t(n)$ describe the worst case largest running time of $A$ on input size $n$. 
Running times of recursively defined algorithms

Algorithms that are described recursively typically have the following structure:

▶ Solve the problem if $n = n_0$; else
  ▶ Preprocessing
  ▶ Recursion to one or more smaller problems
  ▶ Postprocessing

As a result, their running times can be described by recurrence relations of the form

\[
t(n_0) = C \text{ (some positive constant)}
\]
\[
t(n) = f(n) + \sum_{i \in I} t(i) + g(n) \text{ if } n > n_0
\]

For the second bullet,

▶ $f(n)$ is the preprocessing time
▶ $I$ is a set of dataset sizes that we run the algorithm on recursively
▶ $g(n)$ is the time for postprocessing
Example: Bubblesort

Bubblesort is an algorithm that sorts an array of integers by moving from left-to-right, swapping pairs of elements that are out of order. From https://en.wikipedia.org/wiki/Bubble_sort:

```plaintext
procedure bubbleSort( A : list of sortable items )
    n = length(A)
    repeat
        swapped = false
        for i = 1 to n-1 inclusive do
            if A[i-1] > A[i] then
                swap(A[i-1], A[i])
                swapped = true
            end if
        end for
        n = n - 1
    until not swapped
end procedure
```
Recursive bubblesort

Step 0: If $n = 1$, then Return $A$
Step 1: Scan array from left-to-right, swapping adjacent entries that are out of order
Step 2: Recurse on $A[0...n - 2]$
Running time analysis

\( t(n) \) is the running time for recursive BubbleSort on inputs of size \( n \).

1. If \( n = 1 \), the running time is \( C_1 \) for some constant \( C_1 \).
2. The “preprocessing” takes place in Step 0 (checking to see if \( n = 1 \)) and Step 1 (the left-to-right scan, swapping adjacent elements that are out of order), and uses no more than \( C_2 n \) operations.
3. There is only one subproblem and it has \( n - 1 \) elements; hence the recursion takes \( t(n - 1) \) operations.
4. There is no postprocessing stage for this algorithm.

Hence
- \( t(1) = C_1 \)
- for \( n > 1 \), then \( t(n) \leq C_2 n + t(n - 1) \)
Bubblesort running time

We have

- \( t(1) = C_1 \)
- \( t(n) = C_2 n + t(n - 1) \)

Let \( C = \max\{C_1, C_2\} \).

We will prove (using induction) that that \( \forall n \in \mathbb{Z}^+, t(n) \leq Cn^2 \).
Bubblesort running time, continued

We rewrite the recurrence with $\leq$ and obtain the following recurrence for $t(n)$:

- $t(1) \leq C_1$, and
- $t(n) \leq C_2 n + t(n-1)$ if $n > 1$

Let $C = \max\{C_1, C_2\}$.

Our inductive hypothesis is $P(K): \forall k \in \{1, 2, \ldots, K\}, t(k) \leq Cn^2$.

We note that $t(1) \leq C_1 \leq C = C \times 1^2$, so the base case ($K = 1$) holds.
Bounding Bubblesort running time

We now wish to show that $t(K + 1) \leq C(K + 1)^2$.
Since $K + 1 > 1$, we can use the recurrence relation and obtain
$$t(K + 1) \leq C_2(K + 1) + t(K) \leq C(K + 1) + t(K).$$
Then we apply the I.H. to $t(K)$ and obtain $t(K) \leq CK^2$.
Hence
$$t(K + 1) \leq C(K + 1) + t(K) \leq C(K + 1) + CK^2 \leq CK^2 + 2CK + C = C(K + 1)^2$$
Hence we have proved what we set out to prove. Since $K$ was arbitrary, the theorem is true.
Another running time analysis

Consider MergeSort, another nice algorithm for sorting.
For the sake of this algorithm, assume $n = 2^k$.
Input: $A[1...n]$ of integers
Output: $A$ in sorted order (smallest to largest)

- If $n = 1$, return the array
- Divide $A$ into two arrays $A[1...m]$ and $A[m + 1...n]$, where $m = \frac{n}{2}$.
- Recurse on $A[1...m]$ and on $A[m + 1...n]$, so that each is sorted in increasing order
- Merge the two sorted arrays, by taking the smallest off the “top” of each array, and placing into a third array, until all elements moved into third array.
Running time analysis of Merge Sort

- If $n = 1$ the number of operations is $C_1$
- For $n > 1$,
  - the preprocessing is $C_2$
  - there are two subproblems, and each has $\frac{n}{2}$ elements, hence the recursive part uses $2t(\frac{n}{2})$
  - the postprocessing is $C_3n$
- Let $C = \max\{C_1, C_2, C_3\}$

Hence, $t(n)$ (for $n > 1$) satisfies:

$$t(n) = 2t(\frac{n}{2}) + Cn,$$

for some positive constant $C$

It is not hard to see that for some constant $C' > 0$,

$$t(n) \leq C'n \log n$$

for all $n \in \mathbb{Z}^+$. 
Another running time analysis

Consider how to calculate Fibonacci numbers, $F(n)$, defined by

- $F(1) = F(2) = 1$
- $F(n) = F(n-1) + F(n-2)$ if $n > 2$

Let’s do this calculation recursively.

Input: $n \in \mathbb{Z}^+$

Algorithm:

- If $[n = 1 \text{ or } n = 2]$ then Return (1)
- Else
  - Recursively compute $F(n-1)$ and store in $X$
  - Recursively compute $F(n-2)$ and store in $Y$
  - Return $X + Y$
The running time $t_1(n)$ of this algorithm satisfies:

- $t_1(1) = C$
- $t_1(2) = C$
- $t_1(n) = t_1(n - 1) + t_1(n - 2) + C'$

for some positive integers $C, C'$.

It’s immediately obvious that $t_1(n) \geq F(n)$ for all $n \in \mathbb{Z}^+$ (compare the recurrence relations).

This is a problem, because $F(n)$ grows exponentially (look at http://mathworld.wolfram.com/FibonacciNumber.html), and so $t_1(n)$ grows at least exponentially!
Recursive computation of the Fibonacci numbers

When we compute the Fibonacci numbers recursively, we compute $F(n)$ by independently computing $F(n - 1)$ and $F(n - 2)$.

Note that $F(n - 1)$ also requires that we compute $F(n - 2)$, and so $F(n - 2)$ is computed twice, rather than once and then re-used.

It would be much better if we had stored the computations for $F(i)$ for smaller values of $i$ (in \{1, 2, \ldots, n - 1\}) so that they could be re-used.
The recursion tree for the Fibonacci numbers

A better way of computing $F(n)$

The simple recursive way of computing $F(n)$ is exponential, but there is a very simple **dynamic programming** approach that runs in linear time!
Input: $n \in \mathbb{Z}^{+}$

- If $n \leq 2$ return 1. Else:
  - $F[1] := 1$
  - $F[2] := 1$
  - For $i = 3$ upto $n$, DO
    - $F[i] := F[i - 1] + F[i - 2]$
  - Return $F[n]$

The running time $t_2(n)$ for this algorithm satisfies

- $t_2(1) \leq C_0$
- $t_2(2) \leq C_1$
- $t_2(n) = t_2(n - 1) + C_2$ if $n > 2$

for some positive constants $C_0, C_1, C_2$.

Note the difference in the recursive definition for $t_1(n)$ and $t_2(n)$. 
Bounding this recurrence relation

The running time $t_2(n)$ for this algorithm satisfies

- $t_2(1) \leq C_0$
- $t_2(2) \leq C_1$
- $t_2(n) \leq t_2(n - 1) + C_2$

for some positive constants $C_0$, $C_1$, and $C_2$. Let $C' = \max\{C_0, C_1, C_2\}$. It is easy to see that $C' > 0$.

We will prove that $t_2(n) \leq 2C'n$ for all $n \geq 1$, by induction on $n$. 
Running time analysis of the DP algorithm for $F(n)$

Note that

- $t_2(n) \leq C'$ if $n \leq 2$ and
- $t_2(n) \leq t_2(n - 1) + C'$ for $n > 2$

We prove by induction that $t_2(n) \leq C'n$ for all $n \in \mathbb{Z}^+$. 
Running time analysis for DP algorithm for $F(n)$

**Proof:** The base case is $n = 1, 2$, and is verified. The inductive hypothesis is

- $P(K) : t_2(n) \leq C'n$ for all $n \in \{1, 2, \ldots, K\}$.

We want to show that $P(K) \rightarrow P(K + 1)$.

Note, it suffices to show that $P(K) \rightarrow t_2(K + 1) \leq C'(K + 1)$. 
Running time analysis of the DP algorithm for $F(n)$

Recall the I.H.:

- $P(K) : t_2(n) \leq C' n$ for all $n \in \{1, 2, \ldots, K\}$.

We want to show that $t_2(K + 1) \leq C'(K + 1)$.

Since $K \geq 2$, $K + 1 > 2$. Hence, $t_2(K + 1) = t_2(K) + C'$.

By the I.H., $t_2(K) \leq C'K$. Hence,

$$t_2(K + 1) \leq t_2(K) + C' \leq C'K + C' = C'(K + 1)$$

which is what we wanted to prove.

Since $K$ was arbitrary, it follows that $t_2(n) \leq C'n$ for all $n \in \mathbb{Z}^+$. 
We

- Discussed strong vs. weak induction
- Showed how to do an induction proof to establish a property about a recursively defined function $f$ of two variables. The key is defining a parameter that goes down (so that $f(n, m)$ depends on some $f(i, j)$ where $f(i, j)$ can be analyzed using the inductive hypothesis). We showed that doing induction on the sum of $n$ and $m$ worked for our example.
- Talked about running time analyses, and proving upper bounds on them using induction.
- Talked about dynamic programming and recursive algorithms.