Induction

Cosmetic changes made by Steven Archer.

Induction

Induction is a proof technique that allows you to show that a statement holds for all natural numbers. Let's start with an example:

Example

We want to prove the statement

$$\forall n \in \mathbb{N} \quad \sum_{i=0}^{n} i = \frac{n(n+1)}{2}.$$

Alternatively,

$$\forall n \quad 0 + 1 + 2 + \cdots + n = \frac{n(n+1)}{2}.$$

Formally, we want to show that "$\forall n \in \mathbb{N}\ P(n)$" is true, for the predicate $P(n)$ corresponding to $\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$.

Induction

One strategy is to check $P(n)$ for $n = 0, 1, 2, \ldots$ and so on. But this is painful because there are infinitely many cases to check.

- Suppose we know that $P(k)$ is true for some number k
- In other words, suppose that $\sum_{i=0}^{k} i = \frac{k(k+1)}{2}$
- This is called the inductive hypothesis
- Let's see what happens for $k + 1$:

$$\underbrace{\sum_{i=0}^{k} i}_{lhs} + (k+1) = \underbrace{\frac{k(k+1)}{2}}_{rhs} + (k+1)$$

$$= \frac{(k+1)(k+2)}{2}$$

where *lhs*=*rhs* on the first line follows by the assumption that $P(n)$ is true, and the second line follows by simple arithmetic.

Induction

- We have just shown that $P(k+1)$ is true. By the same logic, if $P(n+1)$ is true then $P(n+2)$ is true, and $P(n+3)$ is true, and so on for all values $\geq n$.
- All that remains to show that the statement is true for all n, is to check it for $n = 0$, which is called the *base case*. But this is easy:

$$\text{for } n = 0: \qquad 0 = \frac{0(0+1)}{2}.$$

The Principle of Mathematical Induction

- To prove the statement

$$\forall n \in \mathbb{N} \quad P(n)$$

- it suffices to perform the following three steps:
  1. *Base case:* Prove that $P(0)$ is true.
  2. *Inductive hypothesis:* For arbitrary $k \geq 0$, assume that $P(k)$ is true.
  3. *Inductive step:* Given the inductive hypothesis, prove that $P(k+1)$ is true.

Intuition

- Picture the statement $P(n)$ as an infinite series of dominos.
  - The base case ($n = 0$) knocks over the first domino.
  - The inductive hypothesis and inductive step check that, for all k, domino k is close enough to domino $k + 1$ to knock it over.
- It follows that all the dominos will fall.

Example: Triangle Numbers

Theorem

$\forall n \in \mathbb{N} \quad \sum_{i=0}^{n} i = \frac{n(n+1)}{2}.$

Example: Triangle Numbers

Proof.

By induction on n.

- *Base case ($n = 0$):* We have that $\sum_{i=0}^{0} i = 0 = \frac{0(0+1)}{2}$, and so the base case is correct.
- *Inductive hypothesis:* For arbitrary $k \geq 0$, assume that $\sum_{i=0}^{k} i = \frac{k(k+1)}{2}$. In other words, "let's assume we proven the statement for an arbitrary value of k".
- *Inductive step:* Prove the statement for $k + 1$, i.e. show that $\sum_{i=0}^{k+1} i = \frac{(k+1)(k+2)}{2}$:

□

Example: Triangle Numbers

Proof.

- $$\begin{aligned}\sum_{i=0}^{k+1} i &= \sum_{i=0}^{k} i + (k+1) \\ &= \frac{k(k+1)}{2} + (k+1) \\ &= \frac{k(k+1) + 2(k+1)}{2} \\ &= \frac{(k+1)(k+2)}{2}\end{aligned}$$

  where the 1^{st} line follows by definition, the 2^{nd} by the inductive hypothesis, and the 3^{rd} and 4^{th} by arithmetic.
- The claim follows by the Principle of Mathematical Induction.

□

False Example

Theorem

All cars are the same colour.

Proof.

- Rephrase the statement as $\forall n \in \mathbb{N}_+$, every set of n cars is monochromatic.
- *Base case:* Prove $P(1)$. It is obvious that every set of 1 car is monochromatic.
- *Inductive hypothesis:* Every set of k cars is monochromatic.
- *Inductive step:* Show that $P(k) \to P(k+1)$.
  1. For each set $S = \{i_1, \ldots, i_{k+1}\}$ of $k+1$ cars, consider the k-element subsets $S_1 = \{i_1, \ldots, i_k\}$ and $S_2 = \{i_2, \ldots, i_{k+1}\}$
  2. By the inductive hypothesis, S_1 is monochromatic. Hence cars $i_1, \ldots, i_n$ have the same colour.
  3. Similarly, S_2 is monochromatic and $i_2, \ldots, i_{k+1}$ have the same colour.
  4. Hence all cars in S have the same colour.
- By induction, all cars have the same colour.

Example: Stamps

Theorem

Any integer amount of postage from 8c upwards can be composed from 3c and 5c stamps.

Example: Stamps

Proof.

- Let $P(n)$ be the statement that n can be composed from $3c$ and $5c$ stamps.
- *Base case:* $P(8)$. Obvious.
- *Inductive hypothesis:* Assume $P(8), \ldots, P(k)$ are all true.
- *Inductive step:* Prove $P(k+1)$.
  1. If $k = 9$ then use three $3c$ stamps; if $k = 10$ then use two $5c$ stamps.
  2. If $k + 1 > 10$ then $P(k-2)$ is true by the inductive hypothesis. If $k-2$ can be constructed from $3c$ and $5c$ stamps, then so can $k+1$ by adding one more $3c$ stamp.
- The theorem follows by the principle of induction.

□

Why did we need to assume $P(j)$ is true for all $8 \leq j \leq k$ instead of just assuming that $P(k)$ is true? Is doing so a problem?

Induction and Recursion

A recursive function is constructed to call itself with a "smaller" argument. There is a tight relationship between induction and recursion. In short, induction is the "go to" method for proving the correctness of recursive functions.

Induction and Recursion

Example

Define $n!$, pronounced "n factorial", as

$$n! := \begin{cases} 1 & \text{if } n = 0 \\ \prod_{k=1}^{n} k & \text{else.} \end{cases}$$

$$3! = 1 \times 2 \times 3 = 6$$
$$4! = 1 \times 2 \times 3 \times 4 = 24$$
$$5! = 1 \times 2 \times 3 \times 4 \times 5 = 120$$

Now, recursively define the function f via:

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot f(n-1) & \text{else.} \end{cases}$$

Note that f is defined by calling itself on a "smaller version" of its input.

Example: Factorials

Theorem

For all $n \in \mathbb{N}$, $f(n) = n!$

Example: Factorials

Proof.

- Let $P(n)$ be the statement "f(n)=n!"
- By induction on n:
- *Base case:* $P(0)$ is the proposition that $f(0) = 0! = 1$, which follows immediately from the definition.
- *Inductive hypothesis:* Assume that $P(k)$ is true.
- *Inductive step:* By the definition above

$$\begin{aligned} f(k+1) &= (k+1) \cdot f(k) \\ &= (k+1) \cdot k! \\ &= (k+1) \cdot k \cdot (k-1) \cdots 1 = (k+1)! \end{aligned}$$

  where the first line follows by definition of f, the second by the inductive hypothesis, and the third by definition of $n!$.
- The theorem follows by the principle of induction.

More practice with induction

Theorem

$2^n < n!$ *for all* $n \geq 4$.

Proof.

- By induction.
- *Base case:* $2^4 = 16 < 24 = 4!$, so the result holds when $n = 4$.
- *Inductive hypothesis:* Let $k \geq 4$, and assume that $2^k < k!$.
- *Inductive step:* Observe that

$$\begin{aligned} 2^{k+1} &= 2 \cdot 2^k \\ &< 2 \cdot k! && \text{(by inductive hypothesis)} \\ &< (k+1) \cdot k! && \text{(since } 2 < k+1\text{)} \\ &= (k+1)! \end{aligned}$$

as required.

□

Pseudocode for Binary Search

1. // **PRECONDITION:** `W` **is a word and** `D` **is a subset of the dictionary with at least** `1` **page.**
2. // **POSTCONDITION: Either the definition of** `W` **is returned or '**`W` **not found' is returned.**
3. `function FindWord(W, D)`
4. `    // base case`
5. `    if (D has precisely one page) then`
6. `        look for W in D by brute force.`
7. `        if (found) then return Definition(W, D)`
8. `        else return 'W not found'`
9. `    // recursive case`
10. `    let W' := 1st word on middle page of D`
11. `    if (W comes before W') then`
12. `        return FindWord(1st half of D)`
13. `    else`
14. `        return FindWord(2nd half of D)`

Pseudocode for Binary Search

Theorem

`FindWord()` is correct. That is, it returns the definition of `W`, if `W` is in the dictionary.

Pseudocode for Binary Search

Proof.

- We proceed by induction on n, the number of pages in `D`.
- *Base case (n=1):* If `D` is one page, then line 6 searches for `W` by brute force. If `W` is present it is found and returned; else, '`W` not found' is returned, as desired.
- *Inductive hypothesis:* Assume that `FindWord()` is correct for all $1 \leq j \leq k$.
- *Inductive step:* After line 9, we know `W'` and can therefore determine whether `W` lies in the 1^{st} or the 2^{nd} half of the dictionary. In the 1^{st} case, we recurse on the 1^{st} half of D; in the 2^{nd}, we recurse on the 2^{nd} half.
- By hypothesis, the recursive call correctly finds `W` in the 1^{st} or 2^{nd} half, or returns '`W` not found'. Since `FindWord()` returns the call's answer, it follows that `FindWord()` correctly finds `W` in `D` of size $n + 1$.
- By the principle of induction, `FindWord()` is correct.

Tours on graphs

Recall from a few weeks ago:
Let G be an undirected graph.

- a walk in G is an alternating sequences of vertices and edges

$$v_1 e_1 v_2 e_2 \cdots e_{n-1} v_n$$

  such that each edge e_i is incident with both vertices v_i and v_{i+1}.
- a path is a walk where no vertex appears more than once
- a cycle is a walk where only the *first* and *last* vertices are equal.

Euler and graphs

- An Eulerian walk is a walk in G that uses every edge exactly once.
- An Eulerian tour is an Eulerian walk that starts and finishes at the same vertex.

The seven bridges of Königsberg

Given an undirected graph G,

- does it have an Eulerian tour?
- can you *tell* which graphs have an Eulerian tour just by looking at them?
- can you design an algorithm that finds an Eulerian tour – if one exists?
- can you prove that the algorithm is guaranteed to work?

Eulerian Tours

Theorem

If undirected graph $\mathtt{G} = (\mathtt{V}, \mathtt{E})$ *has an Eulerian tour then (i)* $\mathtt{G}$ *is connected (except for isolated nodes) and (ii)* $\mathtt{G}$ *has even degree (i.e. every vertex has an even number of edges).*

Eulerian Tours

Proof.

- Suppose that G has an Eulerian tour.
- *(i)* G *is connected:*
  Every vertex that has an edge incident to it (i.e. every non-isolated edge) must lie on the tour, and is therefore connected with all other vertices on the tour. Thus, G is connected excepted for isolated vertices.
- *(ii)* G *has even degree:*
  The strategy is to show that we can pair up all the edges in the graph. Notice that (except for the starting vertex s), every time a tour enters a vertex along an edge, it also leaves the vertex along a different edge. Let's pair those two edges up. It follows that every vertex (except maybe s) has an even number of edges.
  The start vertex s is different because of the first edge, which was used to leave it and start the tour. Notice, however, that the tour ends at vertex s, so we can pair the first edge (which leaves s) with the last edge in the tour (which enters s).

Subroutine that finds tours (not necessarily Eulerian)

```
1  // PRECONDITION: G is an undirect, connected graph of even degree; with vertex s.
2  // POSTCONDITION: A tour is returned.
3  function FindTour(G, s)
4      curr_node := s
5      tour := [ ]
6      while HasUntraversedEdge(curr_node) do
7          edge := Pick(curr_node,untransversed edge)
8          curr_node := OtherVertex(edge,curr_node)
9          tour := Append(tour, edge)
10     return tour
```

Subroutine that finds tours (not necessarily Eulerian)

Lemma

When `findTour(G, s)` *enters any vertex* $\mathtt{v} \neq \mathtt{s}$ *it must have traversed an odd no. of edges incident to* $\mathtt{v}$.
When `findTour(G, s)` *enters* $\mathtt{s}$ *it must have traversed an even no. of edges incident to* $\mathtt{s}$.

Subroutine that finds tours (not necessarily Eulerian)

Proof.

- By induction on the length of the walk.
- We only prove the case when $\mathtt{v} \neq \mathtt{s}$
- *Base case:* If the length of the walk is zero, then the walk has traversed an even number of edges.
- *Inductive hypothesis:* Assume that, upon entering any vertex $\mathtt{w} \neq \mathtt{s}$, any walk of length $0, 1, \ldots, k$ must have traversed an odd number of edges incident to $\mathtt{w}$.
- *Inductive step:* Suppose a walk of length $k+1$ enters vertex $\mathtt{v} \neq s$. There are two cases:
  1. The walk never entered vertex $\mathtt{v}$ before. The walk has now traversed one edge incident to $\mathtt{v}$, which is an odd number.
  2. The walk has entered vertex $\mathtt{v}$ before. The last time the walk entered vertex $\mathtt{v}$, it had traversed an odd number of edges (*by hypothesis*). It must then have *left* vertex $\mathtt{v}$, so the number was even – and now it is re-entering, so the number is odd.
- The theorem follows by induction.

Subroutine that finds tours (not necessarily Eulerian)

Lemma

`findTour(G, s)` returns a walk that terminates at `s`. That is, `findTour()` does in fact return a tour.

Proof.

- Since every vertex of `G` has even degree, whenever a walk starting at `s` enters a vertex `v` $\neq$ `s`, it has an escape route – *because it has only used up an **odd** number of edges so far.*
- The `while` loop in `FindTour()` must therefore end at vertex `s`. It follows that the list of edges returned by `FindTour()` is a tour.

□

Subroutine that splices together disjoint tours

1. // **PRECONDITION: $T, T_1, \ldots, T_n$ are a collection of disjoint tours (that is, they have no edges in common) such that T shares a vertex with each of $T_1, \ldots, T_n$.**
2. // **POSTCONDITION: outputs a single tour T' that goes through all the edges in $T, T_1, \ldots, T_n$.**
3. `function` Splice$(T, T_1, \ldots, T_n)$
4. Construct T' by traversing all edges of T and,
5. when a vertex s_i of tour T_i is reached,
   take a detour through tour T_i, and then
6. continue as before.
7. `return` T'

Routine that finds Eulerian tours

1. **// PRECONDITION: `G` is an undirect, connected graph of even degree; with vertex `s`.**
2. **// POSTCONDITION: An Eulerian tour is returned.**
3. `function Euler(G, s)`
4. `tour := FindTour(G, s)`
5. Let $G_1, \ldots, G_n :=$ connected components of G when edges in `tour` are removed
6. Let s_i be first vertex in `tour` that intersects G_i
7. `return Splice(tour, Euler(`G_1, s_1`),...,Euler(`G_n, s_n`))`

Putting it all together

Theorem

The undirected graph $\mathtt{G} = (\mathtt{V}, \mathtt{E})$ *has an Eulerian tour if (i)* $\mathtt{G}$ *is connected (except for isolated edges) and (ii) every vertex of* $\mathtt{G}$ *has an even number of edges.*

Putting it all together

Proof.

- By induction on number m, the number of edges in G.
- *Base case:* If $m = 0$ there are no edges so there is nothing to check.
- *Inductive hypothesis:* `Euler(G, s)` outputs an Eulerian tour in `G` for any even degree, connected graph with at most k edges.
- *Inductive step:* Suppose that `G` has $k + 1$ edges. We know that `T` = `FindTour(G, s)` is a tour, and so has even degree at every vertex. After removing the edges in `T` from `G`, we are left with an even graph with $< k$ edges, *but it might be disconnected.*
- Let $\texttt{G}_1, \ldots, \texttt{G}_t$ be the connected components that tour `T` touches at vertices $\texttt{s}_1, \ldots \texttt{s}_t$. Each has even degree and is connected (up to isolated edges). By the inductive hypothesis $\texttt{Euler}(\texttt{G}_i, \texttt{s}_i)$ returns an Eulerian tour of $\texttt{G}_i$. Finally, `Splice()` combines all the Eulerian tours on the components together into one giant Eulerian tour on the whole graph, and we are done.

Sorting lists

A basic task performed by computers is *sorting* items according to a predefined order (e.g. alphabetically).

- *Selection sort*
  Find the largest item, remove it from the original list and put it at the end of the output list. Repeat until there are no more items.
  Check: The worst case runtime for selection sort is

$$(n-1)+(n-2)+\cdots+1=\sum_{i=1}^{n-1} i=\frac{n(n-1)}{2}.$$

- *Insertion sort*
  Remove the first item from the original list and put it in the output list. Repeat, with the second item and so forth; each time inserting the new item in the correct position in the output list, until no more items are left in the input list.
  Check: The worst case runtime for insertion sort is

$$\sum_{i=1}^{n-1} i=\frac{n(n-1)}{2}.$$

Mergesort

The idea behind `MergeSort()` is to recursively chop the list into two shorter lists, sort both, and then merge them back.

Theorem

The maximum number of comparisons $C(n)$ used by `MergeSort()` satisfies

$$C(n) \leq n \log_2 n - n + 1.$$

Why does this matter?

- Consider sorting a list of 25 million items on a 1GHz CPU. Assume, optimistically, that each comparison takes 10 CPU cycles (including loading, storing and list operations).
- The CPU can perform 100 million comparisons per second.
  - Selection sort takes over a month.
  - Merge sort takes < 6 seconds.

Merge sort

1. // PRECONDITION: `S`, a list of `n` items from a totally ordered set.
2. // POSTCONDITION: A sorted list.
3. `function MergeSort(S)`
4. `    if |S| = 1 then return S`
5. `    else`
6. `        divide S into sublists T (first n/2 items) and U (remaining n/2 items)`
7. `        T' =MergeSort(T)`
8. `        U' =MergeSort(U)`
9. `        S' =Merge(T', U')`
10. `        return S'`

Merge subroutine

1. // **PRECONDITION: two sorted lists `T` and `U`.**
2. // **POSTCONDITION: A single sorted list `S` that combines `T` and `U`.**
3. `function Merge(T,U)`
4. `    pos_T = 1; pos_U = 1`
5. `    out_list = [ ]`
6. `    while pos_T ≤length(T) do`
7. `        while pos_U ≤length(U) do`
8. `            if T[pos_T] ≤ U[pos_U] then`
9. `                Append(out_list,T[pos_T])`
10. `                pos_T = pos_T + 1`
11. `            else`
12. `                Append(out_list,U[pos_U])`
13. `                pos_U = pos_U + 1`
14. `    Append(out_list, “remaining elements”)`
15. `    return out_list`

The stable matching algorithm

Suppose you run a recruitment agency, and your task is to match up n job-seekers and n companies. Each person has an ordered *preference list* over the n companies; similarly, each company has a list describing their preferences over the potential employees.

Person	Company		
	1	2	3
Alice	Dupont	ECNZ	Fonterra
Bob	ECNZ	Dupont	Fonterra
Chris	Dupont	ECNZ	Fonterra

Companies	People		
	1	2	3
Dupont	Bob	Alice	Chris
ECNZ	Alice	Bob	Chris
Fonterra	Alice	Bob	Chris

Question: Can you pair everybody up so they're all "happy"? That is, so that nobody can expect to benefit by changing?

A pairing is unstable if there is a person and a company who prefer each

The stable matching algorithm

Alvin Roth and Lloyd Shapley won the Nobel Prize in Economics in 2012 for their work extending the Stable Matching algorithm.
`input`: two lists, one of n companies and n job-seekers, along with their preferences for one another.

1. `repeat`
2. Every company proposes to the highest ranked person on their list who has not already rejected the offer.
3. Each person collects all the proposals they have received on the current round. To the company they like best they say "maybe, come back tomorrow" (the job-seeker has the company *on a string*); to the others they say "never".
4. Each rejected company crosses the person who rejected them off their list.
5. `until` every person has a company on a string. On this day, each person accepts the offer from the company on their string.

"Stable matching" in real life

While this algorithm presumably won't work for many real world situations, it is used to pair up medical residents with teaching hospitals in the US. It's still used today – with modifications to take into account, for example, that married residents should be placed at the same hospital.

Key observation: Every company starts the algorithm with his first choice as a possibility. As the algorithm proceeds, their best available option *can only get worse*.

On the other hand, every job-seekers options *can only get better* as time passes.

Intuition: As the algorithm progresses, each companies options gets steadily worse and each job-seekers gets steadily better. At some point, the companies and job-seekers must "meet in the middle", and the resulting pairing should be stable.

"Stable matching" in real life

Lemma

"Things can only get better". If company M proposes to person W on the n^{th} day, then on every subsequent day W has a company on their string whom they like at least as much as M.

Proof.

- Proof by induction on the day j, such that $j \geq n$.
- *Base case* ($j = n$)*:* Two things could happen on the n^{th} day:
  1. W has no-one on a string, in which case they say "maybe" to M; or
  2. they have another company M' on a string, in which case they only says maybe to M if they prefers M to M'.

  In both cases, on day n, W has a company on their string they like at least as much as M.
- *Inductive hypothesis:* Suppose the claim is true for day $j \geq n$.
- *Inductive step:* We prove the claim for day $j + 1$. By the inductive hypothesis on day j, W had a company M' on their string who they like at least as much as M.

The stable matching algorithm

Lemma

Everyone is paired up when the stable matching algorithm terminates.

Proof.

- Proof by contradiction.
- Suppose there is a company M who has nobody to offer a job to when the algorithm terminates. It must have proposed to all of the job-seekers on the list. (Why?).
- By the "things can only get better" lemma, every person has had someone on their string after M proposed to them. So, when the algorithm terminates, the n people have n companies on their string, not including M. So there must be at least $n + 1$ companies.
- But this is a contradiction.

□

The stable matching algorithm

Theorem

The pairing produce by the stable matching algorithm is always stable.

Proof.

- Direct proof that no company M can be involved in mutual defecting.
- Consider any pair (M, W) in the matching and suppose that M prefers another job-seeker W^* to their current hire W. We will argue that W^* prefers their current company to M, so M has no chance of attracting the persons attention.
- Observe that, since W^* occurs above W in M's list, it must have proposed to them before it proposed to W. Therefore, W^* must have rejected M for someone they prefer. By the "things can only get better" lemma, W^* likes their final partner at least as much as the company they rejected M for.
- Therefore, W^* prefers their current partner to M, and M cannot be part of a mutual defection.