Insertion Sort

Joseph C Osborn

April 1, 2025

Outline

Sorting Algorithms

Insertion Sort

Proving Sortedness

Sorted Lists

- Next semester, you'll take CS62
- You'll talk a lot about algorithms and data structures
  - Including sorting lists
- Today we'll get a preview in the functional setting
- So what's a sorted list?

Sorting a List

- Sort the list `[1,5,2,3,1]` in ascending order

Sorting a List

- Sort the list `[1,5,2,3,1]` in ascending order
  - `[1,1,2,3,5]`

Sorting a List

- Sort the list `[1,5,2,3,1]` in ascending order
  - `[1,1,2,3,5]`
- We can sort lists of anything `Ord`

Sorting a List

- Sort the list `[1,5,2,3,1]` in ascending order
  - `[1,1,2,3,5]`
- We can sort lists of anything `Ord`
- Two important things about the output list:

Sorting a List

- Sort the list [1,5,2,3,1] in ascending order
  - [1,1,2,3,5]
- We can sort lists of anything Ord
- Two important things about the output list:
  1. It is in ascending order

Sorting a List

- Sort the list `[1,5,2,3,1]` in ascending order
  - `[1,1,2,3,5]`
- We can sort lists of anything `Ord`
- Two important things about the output list:
  1. It is in ascending order
  2. It is a *permutation* of the input list

Ascending Order

- We can write this different ways for a list l:
  - forall x, if x appears at n in l, and n is not the end of the list, then `nth l (n+1)` is at least as big as x.
  - forall n, if n $<$ `length l` then `nth l n` $\leq$ `nth l (n+1)`
  - "An empty list and a one element list are sorted; prepending an element x to a list is sorted if x $\leq$ the first element of the list, if any" (an inductive definition!)
  - `sorted(l)` = `True` where:

```
sorted (x:y:l) = x <= y && sorted (y:l)
sorted _ = True
```

Permutations

- We need this property too!
  - Otherwise `[]` is a perfect output for any "sorting function"
- The new list needs the same elements as the old list, but possibly in a different order

Key Idea

- Here's a simple sorting function

Key Idea

- Here's a simple sorting function
- We'll sort a list by making a new list from it, one element at a time

Key Idea

- Here's a simple sorting function
- We'll sort a list by making a new list from it, one element at a time
- The new list will *always* be sorted, by construction

Key Idea

- Here's a simple sorting function
- We'll sort a list by making a new list from it, one element at a time
- The new list will *always* be sorted, by construction
- Inserting the new element into the new list does one of two things:

Key Idea

- Here's a simple sorting function
- We'll sort a list by making a new list from it, one element at a time
- The new list will *always* be sorted, by construction
- Inserting the new element into the new list does one of two things:
  - If this element is smaller than the front of the new list (or if the new list is empty), just cons it on

Key Idea

- Here's a simple sorting function
- We'll sort a list by making a new list from it, one element at a time
- The new list will *always* be sorted, by construction
- Inserting the new element into the new list does one of two things:
  - If this element is smaller than the front of the new list (or if the new list is empty), just cons it on
  - If this element is bigger than the front of the new list, recurse with the tail of the new list

Key Idea

- Here's a simple sorting function
- We'll sort a list by making a new list from it, one element at a time
- The new list will *always* be sorted, by construction
- Inserting the new element into the new list does one of two things:
  - If this element is smaller than the front of the new list (or if the new list is empty), just cons it on
  - If this element is bigger than the front of the new list, recurse with the tail of the new list
- Once we've inserted every element, we're done!

Insert

```
insertion_sort [] = []
insertion_sort (x:l) = insert x (insertion_sort l)

insert _x [] = [x]
insert x (y:l)
  | x <= y = x:y:l
  | otherwise = y:(insert x l)
```

- Try it on these lists: `[2, 1, 3]`, `[1, 2, 3]`, `[3, 2, 1]`.

Preservation Properties

- We often want to prove that applying some function doesn't lose us some property
  - We call these "preservation properties"
  - E.g., "map preserves length" is the property that mapping over a list doesn't change its length
  - E.g., "filter preserves order" states that the order of elements in a list won't change through filter
- Preservation properties are an easy way to build up proofs about a whole procedure
  - If each step of the procedure preserves the thing we care about, then the whole procedure will too

Insert Preserves Sortedness

- Claim: forall l, `sorted l` = `True` implies forall x, `sorted (insert x l)` = `True`

Insert Preserves Sortedness

- Claim: forall l, `sorted l = True` implies forall x, `sorted (insert x l) = True`
- By induction on l.

Insert Preserves Sortedness

- Claim: forall l, `sorted l = True` implies forall x, `sorted (insert x l) = True`
- By induction on l.
- (l=[]). Let x be given; we know `insert x [] = [x]` and we know single-element lists are trivially sorted.

Insert Preserves Sortedness

- Claim: forall l, `sorted l` = `True` implies forall x, `sorted (insert x l)` = `True`
- By induction on l.
- (l=[]). Let x be given; we know `insert x []` = `[x]` and we know single-element lists are trivially sorted.
- (l=(y:l')). IH: `sorted l'` = `True` implies forall x, `sorted (insert x l')` = `True`.

Insert Preserves Sortedness

- Claim: forall l, `sorted l` = `True` implies forall x, `sorted (insert x l)` = `True`
- By induction on l.
- (l=[]). Let x be given; we know `insert x []` = `[x]` and we know single-element lists are trivially sorted.
- (l=(y:l')). IH: `sorted l'` = `True` implies forall x, `sorted (insert x l')` = `True`.
  - WTP `sorted (y:l')` = `True` implies forall z, `sorted (insert z (y:l'))` = `True`.

Insert Preserves Sortedness

- Claim: forall l, `sorted l` = `True` implies forall x, `sorted (insert x l)` = `True`
- By induction on l.
- (l=[]). Let x be given; we know `insert x []` = `[x]` and we know single-element lists are trivially sorted.
- (l=(y:l')). IH: `sorted l'` = `True` implies forall x, `sorted (insert x l')` = `True`.
  - WTP `sorted (y:l')` = `True` implies forall z, `sorted (insert z (y:l'))` = `True`.
  - Assume `sorted (y:l')` = `True`. Let z be given.

Insert Preserves Sortedness

- (IH: `sorted l'` = `True` implies forall x, `sorted (insert x l')` = `True`.)

Insert Preserves Sortedness

- (IH: `sorted l'` = `True` implies forall x, `sorted (insert x l')` = `True`.)
- We have to show `sorted (insert z (y:l'))` = `True`. By cases on whether z $\leq$ y.

Insert Preserves Sortedness

- (IH: `sorted l'` = `True` implies forall x, `sorted (insert x l')` = `True`.)
- We have to show `sorted (insert z (y:l'))` = `True`. By cases on whether z $\leq$ y.
- (z $\leq$ y): We have to show `sorted (z:y:l')` = `True`.

Insert Preserves Sortedness

- (IH: `sorted l'` = `True` implies forall x, `sorted (insert x l')` = `True`.)
- We have to show `sorted (insert z (y:l'))` = `True`. By cases on whether $z \leq y$.
- ($z \leq y$): We have to show `sorted (z:y:l')` = `True`.
  - We know $z \leq y$, and we already assumed `(y:l')` is sorted, so we are done.

Insert Preserves Sortedness

- (IH: `sorted l'` = `True` implies forall x, `sorted (insert x l')` = `True`.)
- We have to show `sorted (insert z (y:l'))` = `True`. By cases on whether z $\leq$ y.
- (z $\leq$ y): We have to show `sorted (z:y:l')` = `True`.
  - We know z $\leq$ y, and we already assumed `(y:l')` is sorted, so we are done.
- (y $<$ z): We have to show `sorted (y:(insert z l'))` = `True`.

Insert Preserves Sortedness

- (IH: `sorted l'` = `True` implies forall x, `sorted (insert x l')` = `True`.)
- We have to show `sorted (insert z (y:l'))` = `True`. By cases on whether $z \leq y$.
- ($z \leq y$): We have to show `sorted (z:y:l')` = `True`.
  - We know $z \leq y$, and we already assumed `(y:l')` is sorted, so we are done.
- ($y < z$): We have to show `sorted (y:(insert z l'))` = `True`.
  - Considering `sorted (y:(insert z l'))`, we know y is no bigger than the first element of `insert z l'`, which will either be z or some element of l'.

Insert Preserves Sortedness

- (IH: `sorted l'` = `True` implies forall x, `sorted (insert x l')` = `True`.)
- We have to show `sorted (insert z (y:l'))` = `True`. By cases on whether $z \leq y$.
- ($z \leq y$): We have to show `sorted (z:y:l')` = `True`.
  - We know $z \leq y$, and we already assumed `(y:l')` is sorted, so we are done.
- ($y < z$): We have to show `sorted (y:(insert z l'))` = `True`.
  - Considering `sorted (y:(insert z l'))`, we know y is no bigger than the first element of `insert z l'`, which will either be z or some element of l'.
    - Because we know $y < z$ in this case and we assumed `sorted (y:l')`.

Insert Preserves Sortedness

- (IH: `sorted l'` = `True` implies forall x, `sorted (insert x l')` = `True`.)
- We have to show `sorted (insert z (y:l'))` = `True`. By cases on whether $z \leq y$.
- ($z \leq y$): We have to show `sorted (z:y:l')` = `True`.
  - We know $z \leq y$, and we already assumed `(y:l')` is sorted, so we are done.
- ($y < z$): We have to show `sorted (y:(insert z l'))` = `True`.
  - Considering `sorted (y:(insert z l'))`, we know y is no bigger than the first element of `insert z l'`, which will either be z or some element of l'.
    - Because we know $y < z$ in this case and we assumed `sorted (y:l')`.
  - So `sorted (y:(insert z l'))` is true exactly when `sorted (insert z l')` is true.

Insert Preserves Sortedness

- (IH: `sorted l'` = `True` implies forall x, `sorted (insert x l')` = `True`.)
- We have to show `sorted (insert z (y:l'))` = `True`. By cases on whether z $\leq$ y.
- (z $\leq$ y): We have to show `sorted (z:y:l')` = `True`.
  - We know z $\leq$ y, and we already assumed `(y:l')` is sorted, so we are done.
- (y $<$ z): We have to show `sorted (y:(insert z l'))` = `True`.
  - Considering `sorted (y:(insert z l'))`, we know y is no bigger than the first element of `insert z l'`, which will either be z or some element of l'.
    - Because we know y $<$ z in this case and we assumed `sorted (y:l')`.
  - So `sorted (y:(insert z l'))` is true exactly when `sorted (insert z l')` is true.
  - We could have `sorted (insert z l')` by our IH, if we could prove `sorted (y:l')`; and we already know `sorted (y:l')` from our assumption.

insertion$_{sort}$ Produces Sorted Lists

- Claim: forall l, `sorted (insertion_sort l)` = `True`

$\text{insertion}_{\text{sort}}$ Produces Sorted Lists

- Claim: forall l, `sorted (insertion_sort l)` = `True`
- Proof. By induction on l.

$\text{insertion}_{\text{sort}}$ Produces Sorted Lists

- Claim: forall l, `sorted (insertion_sort l) = True`
- Proof. By induction on l.
  - (l=[]). `insertion_sort [] = []`. Empty lists are trivially sorted.

$\text{insertion}_{\text{sort}}$ Produces Sorted Lists

- Claim: forall l, `sorted (insertion_sort l) = True`
- Proof. By induction on l.
  - (l=[]). `insertion_sort [] = []`. Empty lists are trivially sorted.
  - (l=(x:l')). IH: `sorted (insertion_sort l') = True`

insertion$_{sort}$ Produces Sorted Lists

- Claim: forall l, `sorted (insertion_sort l)` = `True`
- Proof. By induction on l.
  - (l=[]). `insertion_sort []` = `[]`. Empty lists are trivially sorted.
  - (l=(x:l')). IH: `sorted (insertion_sort l')` = `True`
  - WTP: `sorted (insertion_sort (y:l'))` = `True`.

$\text{insertion}_{\text{sort}}$ Produces Sorted Lists

- Claim: forall l, `sorted (insertion_sort l) = True`
- Proof. By induction on l.
  - (l=[]). `insertion_sort [] = []`. Empty lists are trivially sorted.
  - (l=(x:l')). IH: `sorted (insertion_sort l') = True`
  - WTP: `sorted (insertion_sort (y:l')) = True`.
  - `insertion_sort (y:l')` is just `insert y (insertion_sort l')`

$\text{insertion}_{\text{sort}}$ Produces Sorted Lists

- Claim: forall l, `sorted (insertion_sort l)` = `True`
- Proof. By induction on l.
  - (l=[]). `insertion_sort [] = []`. Empty lists are trivially sorted.
  - (l=(x:l')). IH: `sorted (insertion_sort l')` = `True`
  - WTP: `sorted (insertion_sort (y:l'))` = `True`.
  - `insertion_sort (y:l')` is just `insert y (insertion_sort l')`
  - Since our IH states that `insertion_sort l'` is sorted, and we know that insert preserves sortedness, we know `insert y (insertion_sort l')` must also be sorted.

Using Sortedness

- Consider: forall l f, `filter f (insertion_sort l) = insertion_sort (filter f l)`
  1. Why is this an interesting property?
  2. Prove it!