Sorting

Rob Hackman

Fall 2025

University of Alberta

Table of Contents

One problem, many solutions

The problem of sorting a collection of objects, as with any problem, has many[1] potential solutions to it.

[1] Technically *infinite* solutions though this is true because of the triviality of creating nearly identical "different" solutions.

One problem, many solutions

The problem of sorting a collection of objects, as with any problem, has many[1] potential solutions to it.

As such, we will investigate a number of different solutions to this problem.

[1] Technically *infinite* solutions though this is true because of the triviality of creating nearly identical "different" solutions.

One problem, many solutions

The problem of sorting a collection of objects, as with any problem, has many[1] potential solutions to it.

As such, we will investigate a number of different solutions to this problem.

we will start with an intuitive solution to sorting a collection of entities — *insertion sort*.

[1] Technically *infinite* solutions though this is true because of the triviality of creating nearly identical "different" solutions.

Playing Cards

Since we have stated that insertion sort is an intuitive solution, we will start with an example in which one may instinctively perform it.

Playing Cards

Since we have stated that insertion sort is an intuitive solution, we will start with an example in which one may instinctively perform it.

Imagine you are playing a card game with friends with a standard 52-card deck of French-suited playing cards. The game you are playing is a trick-taking game in which it makes sense to sort your cards by their rank from lowest to highest.

Playing Cards

Since we have stated that insertion sort is an intuitive solution, we will start with an example in which one may instinctively perform it.

Imagine you are playing a card game with friends with a standard 52-card deck of French-suited playing cards. The game you are playing is a trick-taking game in which it makes sense to sort your cards by their rank from lowest to highest.

You are allowed to pick up each cards as your friend deals them out, so each time your friend deals you a card you add it to your hand. Let's examine what one might do as they are dealt their cards!

The procedure of insertion sort

The procedure we followed for sorting our hand of cards may be so intuitive it seems there is not much to say about what was done. How might we define the procedure we followed?

The procedure of insertion sort

The procedure we followed for sorting our hand of cards may be so intuitive it seems there is not much to say about what was done. How might we define the procedure we followed?

We may simply say “Whenever we were dealt a card we immediately placed in its correct location”.

The procedure of insertion sort

The procedure we followed for sorting our hand of cards may be so intuitive it seems there is not much to say about what was done. How might we define the procedure we followed?

We may simply say "Whenever we were dealt a card we immediately placed in its correct location".

But that definition ignores some important details of the procedure! For one, it assumes that a "correct location" always exists — is that always true?

Fixing our procedure

The issue in the counterexample shown was that the instructions of our procedure had a flaw.

Fixing our procedure

The issue in the counterexample shown was that the instructions of our procedure had a flaw.

"Whenever we were dealt a card we immediately placed in its correct location" fails when no such correct location exists, as in our example! How can we guarantee a correct location exists?

Fixing our procedure

The issue in the counterexample shown was that the instructions of our procedure had a flaw.

"Whenever we were dealt a card we immediately placed in its correct location" fails when no such correct location exists, as in our example! How can we guarantee a correct location exists?

A correct location exists when we are placing a card in a hand that is already sorted. So, instead we will adjust our procedure with that constraint in mind.

Updated insertion sort procedure

Our new updated procedure for insertion sort is as follows

Updated insertion sort procedure

Our new updated procedure for insertion sort is as follows

"Given we have a sorted hand of cards, whenever we are dealt a card immediately insert it into the location in our hand where it is in sorted order"

Updated insertion sort procedure

Our new updated procedure for insertion sort is as follows

"Given we have a sorted hand of cards, whenever we are dealt a card immediately insert it into the location in our hand where it is in sorted order"

However, this new assumption may seem to defeat the purpose of our exercise. If we already have a sorted hand of cards we wouldn't need to sort it! How does this help us build a sorted list from an arbitrary one?

The `insert` function

The process we defined on the previous slide is *not* the definition of insertion sort, rather it is the *step* that must be followed for each item of the collection to sort.

The `insert` function

The process we defined on the previous slide is *not* the definition of insertion sort, rather it is the *step* that must be followed for each item of the collection to sort.

The process we've defined is that of *inserting* a single element into a sorted list to build a new sorted list one element larger. Let us first define this function `insert` and then see how this can be used to build a solution to sorting an arbitrary list.

Writing `insert`

We now define `insert`

```
def insert(elem, sl):
    '''
    insert produces a new list sorted in non-decreasing
           order, the result of inserting elem into sl

    elem     - X
    sl       - LList of X, sorted in
               non-decreasing order
    returns - LList of X

    Example:
      insert(5, LL(-1, 4, 10, 20)) -> LL(-1, 4, 5, 10, 20)
    '''
```

Writing `insert`

We must find the correct location to insert `elem`. As such we will need to look at multiple values in our `LList`. Recursion over our `LList` will be necessary.

```
def insert(elem, sl):
```

Writing `insert`

So, we start with a base case, when is it trivial to know where to insert a card in sorted order? Thinking back on our card example there was no decision to be made when we were dealt our first card and our hand was empty!

```
def insert(elem, sl):
```

Writing `insert`

Therefore when our `LList` is empty then we of course know where to insert `elem`, as there is only one choice.

```
def insert(elem, sl):
  if isEmpty(sl):
    return cons(elem, empty())
```

Writing `insert`

Now, we must consider our recursive case. As always we consider our "current item" and what must be done with it. So what is relevant about the first of our `LList`?

```
def insert(elem, sl):
  if isEmpty(sl):
    return cons(elem, empty())
```

Writing `insert`

Since our `LList` is sorted in non-decreasing order that means everything after `first(sl)` is guaranteed to be greater than or equal to it. So if `elem` is *smaller* than `first(sl)` then `elem` should become the first of our new `LList`!

```
def insert(elem, sl):
  if isEmpty(sl):
    return cons(elem, empty())
  if elem < first(sl):
    return cons(elem, sl)
```

Writing `insert`

On the other hand, if `elem` is not smaller than `first(sl)` then inserting it *after* `first(sl)` in our new `LList` cannot be incorrect. So we should insert `elem` into the rest of `sl`, making sure not to forget to include `first(sl)` in our newly created `LList`.

```
def insert(elem, sl):
  if isEmpty(sl):
    return cons(elem, empty())
  if elem < first(sl):
    return cons(elem, sl)
  return cons(first(sl), insert(elem, sl))
```

Writing `insert`

Our `insert` function is now complete, and can now be used to solve the problem of sorting an arbitrary `LList`. `LList`.

```
def insert(elem, sl):
  if isEmpty(sl):
    return cons(elem, empty())
  if elem < first(sl):
    return cons(elem, sl)
  return cons(first(sl), insert(elem, sl))
```

Writing `insertionSort`

Now we will write `insertionSort`.

```
def insertionSort(l):
    '''
    insertionSort sorts the given LList in non-decreasing
                  order.
    l        - LList of X, X is comparable.
    returns - LList of X

    Examples:
      insertionSort(LL(5, 9, 8, 2)) -> LL(2, 5, 8, 9)
      insertionSort(LL(13, 3, 9, 4, 2)) -> LL(2, 3, 4, 9, 13)
    '''
```

Writing `insertionSort`

Not sure where to begin, we think back to how we sorted our hand. To begin with we had an empty hand, and each time we were given an item we inserted it in sorted order. That *is* our algorithm for insertion sort, so we start the same — with an empty `LList` as our base case.

```
def insertionSort(l):
```

Writing `insertionSort`

Since an empty collection is trivially sorted, our function correctly sorts an empty `LList`. What to do if `l` is *not* empty?

```
def insertionSort(l):
  if isEmpty(l):
    return empty()
```

Writing `insertionSort`

When `l` is not empty then we aim to build our answer up from the recursive result combined with our current element. So we call `insertionSort` recursively on the rest of `l`.

```
def insertionSort(l):
  if isEmpty(l):
    return empty()
  ror = insertionSort(rest(l))
```

Writing `insertionSort`

`insertionSort` is a function that promises us to return the sorted version of a given `LList`. That means that our result of recursion is the sorted version of the rest of our `LList`!

Given a sorted version of the rest of `l` the only work that remains to be completed is that of inserting the first of `l` into that sorted `LList`!

```
def insertionSort(l):
  if isEmpty(l):
    return empty()
  ror = insertionSort(rest(l))
```

Writing `insertionSort`

Since our recursive result is the sorted `LList` with everything *except* the first of `l` then simply inserting the first of `l` into that result is the sorted `LList` with *all* the elements of `l`!

```
def insertionSort(l):
  if isEmpty(l):
    return empty()
  ror = insertionSort(rest(l))
  return insert(first(l), ror)
```

Runtime of insertion sort

What is the runtime of `insertionSort`?

Runtime of insertion sort

What is the runtime of `insertionSort`?

- Each call to `insertionSort` performs only basic operations plus a call to `insert`.

Runtime of insertion sort

What is the runtime of `insertionSort`?

- Each call to `insertionSort` performs only basic operations plus a call to `insert`.
- There is one recursive call for each non-base case application of `insertionSort`, each applied to the `rest` of the `LList`, so there will be n calls to `insertionSort` for a `LList` of length n.

Runtime of insertion sort

What is the runtime of `insertionSort`?

- Each call to `insertionSort` performs only basic operations plus a call to `insert`.
- There is one recursive call for each non-base case application of `insertionSort`, each applied to the `rest` of the `LList`, so there will be n calls to `insertionSort` for a `LList` of length n.
- Then for each of the n calls the work performed is a constant amount of work by `insertionSort` plus the work performed by the call to `insert`

Runtime of insertion sort

What is the runtime of `insertionSort`?

- Each call to `insertionSort` performs only basic operations plus a call to `insert`.
- There is one recursive call for each non-base case application of `insertionSort`, each applied to the `rest` of the `LList`, so there will be n calls to `insertionSort` for a `LList` of length n.
- Then for each of the n calls the work performed is a constant amount of work by `insertionSort` plus the work performed by the call to `insert`
- So, the runtime of `insertionSort` will be nx where x is the runtime of `insert`, so we must determine the runtime of `insert`

Runtime of `insert`

What is the runtime of `insert`?

Runtime of `insert`

What is the runtime of `insert`?

- `insert` performs only basic operations, except for its recursive call.

Runtime of `insert`

What is the runtime of `insert`?

- `insert` performs only basic operations, except for its recursive call.
- How many times does `insert` recursively call itself?

Runtime of `insert`

What is the runtime of `insert`?

- `insert` performs only basic operations, except for its recursive call.
- How many times does `insert` recursively call itself?
  - It depends! `insert` is finished its work when it finds the correct location to insert the given element.

Runtime of `insert`

What is the runtime of `insert`?

- `insert` performs only basic operations, except for its recursive call.
- How many times does `insert` recursively call itself?
  - It depends! `insert` is finished its work when it finds the correct location to insert the given element.
  - If the element should be inserted as the first element, then `insert` is never recursively called and it does a constant amount of work!

Runtime of `insert`

What is the runtime of `insert`?

- `insert` performs only basic operations, except for its recursive call.
- How many times does `insert` recursively call itself?
  - It depends! `insert` is finished its work when it finds the correct location to insert the given element.
  - If the element should be inserted as the first element, then `insert` is never recursively called and it does a constant amount of work!
  - If the element should be inserted as the last element, then `insert` must walk the entire `LList` recursively, so it performs n recursive calls (each doing constant work).

Runtime of `insert`

What is the runtime of `insert`?

- `insert` performs only basic operations, except for its recursive call.
- How many times does `insert` recursively call itself?
  - It depends! `insert` is finished its work when it finds the correct location to insert the given element.
  - If the element should be inserted as the first element, then `insert` is never recursively called and it does a constant amount of work!
  - If the element should be inserted as the last element, then `insert` must walk the entire `LList` recursively, so it performs n recursive calls (each doing constant work).
- So best-case `insert` is $\Theta(1)$ and worst-case it is $\Theta(n)$. We will worry only about worst-case runtime.

Completed runtime of insertion sort

Now that we know the work done by `insert` is linear relative to the length of its `LList` parameter we can calculate the work performed by `insertionSort`

Completed runtime of insertion sort

Now that we know the work done by `insert` is linear relative to the length of its `LList` parameter we can calculate the work performed by `insertionSort`

The amount of work performed by `insertionSort` is then $n \times \Theta(n) \rightarrow \Theta(n^2)$.

Completed runtime of insertion sort

Now that we know the work done by `insert` is linear relative to the length of its `LList` parameter we can calculate the work performed by `insertionSort`

The amount of work performed by `insertionSort` is then $n \times \Theta(n) \rightarrow \Theta(n^2)$.

So considered worst case `insertionSort` takes quadratic time — as such, it cannot be used to write `findPairs` faster than $\Theta(n^2)$.

Table of Contents

Another sorting algorithm

Sometimes when playing a card game a player is not allowed to pick up their cards until the entire hand is dealt. As such, they pick up their entire hand that is dealt and then may want to sort it[2].

[2] Technically, they could choose to pick up their hand one card at a time and perform insertion sort. However, we will consider that they pick it up all at once and then want to sort it.

Another sorting algorithm

Sometimes when playing a card game a player is not allowed to pick up their cards until the entire hand is dealt. As such, they pick up their entire hand that is dealt and then may want to sort it[2].

If you have ever played such a game, and attempted to sort your hand after picking up all the cards, you may have in the past fumbled around swapping the cards around in your hand until you reach a sorted order. The algorithm we are about to describe is a formalized order of that fumbling sort.

[2] Technically, they could choose to pick up their hand one card at a time and perform insertion sort. However, we will consider that they pick it up all at once and then want to sort it.

Bubble sort algorithm

Bubble sort's name comes from the fact procedure of repeated comparisons of adjacent pairs, which "bubbles" the largest value to the end of our sequence. So this process, the process of comparing the first n adjacent pairs to each other, we will call one single "bubbling". Then the algorithm for Bubble sort becomes quite simple.

Bubble sort algorithm

Bubble sort's name comes from the fact procedure of repeated comparisons of adjacent pairs, which "bubbles" the largest value to the end of our sequence. So this process, the process of comparing the first n adjacent pairs to each other, we will call one single "bubbling". Then the algorithm for Bubble sort becomes quite simple.

- Perform a "bubbling" of the sequence

Bubble sort algorithm

Bubble sort's name comes from the fact procedure of repeated comparisons of adjacent pairs, which "bubbles" the largest value to the end of our sequence. So this process, the process of comparing the first n adjacent pairs to each other, we will call one single "bubbling". Then the algorithm for Bubble sort becomes quite simple.

- Perform a "bubbling" of the sequence
- If you have completed n bubblings, where n is the length of the sequence, then return the result of that bubbling

Bubble sort algorithm

Bubble sort's name comes from the fact procedure of repeated comparisons of adjacent pairs, which "bubbles" the largest value to the end of our sequence. So this process, the process of comparing the first n adjacent pairs to each other, we will call one single "bubbling". Then the algorithm for Bubble sort becomes quite simple.

- Perform a "bubbling" of the sequence
- If you have completed n bubblings, where n is the length of the sequence, then return the result of that bubbling
- If you have not completed n bubblings, then begin at step one again but operating on the sequence that was produced by the previous bubbling

Implementing `bubble`

We will now implement `bubble` which takes a single `LList` parameter and performs a single "bubbling" procedure on that `LList`.

```
def bubble(l):
```

Implementing `bubble`

As with any of our functions we will determine our base case. When is the result of "bubbling" a `LList` trivial to know?

If the `LList` contains zero or one value then the result is simply the `LList` itself

```
def bubble(l):
  if len(l) <= 1:
    return l
```

Implementing `bubble`

Now what needs to be done if the `LList` is not empty or one element large? The bubbling process works by comparing each set of adjacent pairs and swapping them if necessary.

The adjacent pair we can easily access is the first pair, so that is the pair on which we operate.

```
def bubble(l):
  if len(l) <= 1:
    return l
  if first(l) > second(l):
```

Implementing `bubble`

If the first item in our `LList` in our pair is greater than the second then their positions should swap in our resultant `LList`!

So our result should be the `LList` that has the first and second item swapped.

```
def bubble(l):
  if len(l) <= 1:
    return l
  if first(l) > second(l):
    newFirst = second(l)
    newRest = cons(first(l), rest(rest(l)))
```

Implementing `bubble`

However this has only compared the *first* adjacent pair. We must compare each adjacent pair in our whole `LList` to complete one bubbling, so we recursively call `bubble` on the (new) rest of our `LList` and build our final answer from its result!

```
def bubble(l):
  if len(l) <= 1:
    return l
  if first(l) > second(l):
    newFirst = second(l)
    newRest = cons(first(l), rest(rest(l)))
    return cons(newFirst, bubble(newRest))
```

Implementing `bubble`

This only handles the recursive case where we *should* swap the current adjacent pair we're looking at. If we should *not* perform such a swap then the result should simply be the recursive result combined with our current first element.

```
def bubble(l):
  if len(l) <= 1:
    return l
  if first(l) > second(l):
    newFirst = second(l)
    newRest = cons(first(l), rest(rest(l)))
    return cons(newFirst, bubble(newRest))
  return cons(first(l), bubble(rest(l)))
```

Implementing `bubble`

Now we have the function `bubble` that returns the result of applying the "bubbling" procedure to a `LList` once. Now to implement `bubbleSort` we simply need to call this function multiple times.

```
def bubble(l):
  if len(l) <= 1:
    return l
  if first(l) > second(l):
    newFirst = second(l)
    newRest = cons(first(l), rest(rest(l)))
    return cons(newFirst, bubble(newRest))
  return cons(first(l), bubble(rest(l)))
```

Implementing `bubbleSort`

We will now implement `bubbleSort` which takes a single `LList` parameter and returns the result of sorting that `LList` in non-decreasing order using the bubble sort algorithm.

```
def bubbleSort(l):
```

Implementing `bubbleSort`

We note that we want to call `bubble` a number of times equal to the length of the given `LList`. In order to repeat our process that many times we need to write a recursion on that number. So we write a helper function that takes a `LList` and an integer that represents the number of times to perform `bubble`.

```
def bubbleSort(l):
  def repeatBubble(bl, n):
```

Implementing `bubbleSort`

If n is zero, then `bubble` does not need to be applied to the parameter `bl` at all. As such, the answer is just `bl`.

```
def bubbleSort(l):
  def repeatBubble(bl, n):
    if n == 0:
      return bl
```

Implementing `bubbleSort`

If n is non-zero then we want to apply `bubble` that many times. We observe that three result of applying `bubble` to `bl` n times is the same thing as applying `bubble` to `bubble(b1)` $n-1$ times. As such, our recursion becomes obvious.

```
def bubbleSort(l):
  def repeatBubble(bl, n):
    if n == 0:
      return bl
    return repeatBubble(bubble(bl), n-1)
```

Implementing `bubbleSort`

The last thing left to do is to have `bubbleSort` return the result of n bubblings.

```
def bubbleSort(l):
  def repeatBubble(bl, n):
    if n == 0:
      return bl
    return repeatBubble(bubble(bl), n-1)
  return repeatBubble(l, len(l))
```

Runtime of bubble sort

Runtime of bubble sort

- `bubbleSort` simply returns `repeatBubble` of the `LList` and its length, so its runtime will be that of `repeatBubble`.

Runtime of bubble sort

- `bubbleSort` simply returns `repeatBubble` of the `LList` and its length, so its runtime will be that of `repeatBubble`.
- Each call to `repeatBubble` performs only basic operations plus a call to `bubble`.

Runtime of bubble sort

- `bubbleSort` simply returns `repeatBubble` of the `LList` and its length, so its runtime will be that of `repeatBubble`.
- Each call to `repeatBubble` performs only basic operations plus a call to `bubble`.
- There is one recursive call for each non-base case application of `repeatBubble`, each applied to the $n - 1$, so there will be n recursive calls to `repeatBubble` for a `LList` of length n.

Runtime of bubble sort

- `bubbleSort` simply returns `repeatBubble` of the `LList` and its length, so its runtime will be that of `repeatBubble`.
- Each call to `repeatBubble` performs only basic operations plus a call to `bubble`.
- There is one recursive call for each non-base case application of `repeatBubble`, each applied to the $n - 1$, so there will be n recursive calls to `repeatBubble` for a `LList` of length n.
- Then for each of the n calls the work performed is a constant amount of work by `repeatBubble` plus the work performed by the call to `bubble`

Runtime of bubble sort

- `bubbleSort` simply returns `repeatBubble` of the `LList` and its length, so its runtime will be that of `repeatBubble`.
- Each call to `repeatBubble` performs only basic operations plus a call to `bubble`.
- There is one recursive call for each non-base case application of `repeatBubble`, each applied to the $n - 1$, so there will be n recursive calls to `repeatBubble` for a `LList` of length n.
- Then for each of the n calls the work performed is a constant amount of work by `repeatBubble` plus the work performed by the call to `bubble`
- So, the runtime of `repeatBubble` will be nx where x is the runtime of `bubble`, so we must determine the runtime of `bubble`

Runtime of `bubble`

What is the runtime of `bubble`?

Runtime of `bubble`

What is the runtime of `bubble`?

- `bubble` performs only basic operations, except for its recursive call.

Runtime of `bubble`

What is the runtime of `bubble`?

- `bubble` performs only basic operations, except for its recursive call.
- `bubble` recursively calls itself until the `LList` is empty or length one, and each recursive application of `bubble` is to a `LList` with one less element in it so there will be $n - 1$ recursive calls.

Runtime of `bubble`

What is the runtime of `bubble`?

- `bubble` performs only basic operations, except for its recursive call.
- `bubble` recursively calls itself until the `LList` is empty or length one, and each recursive application of `bubble` is to a `LList` with one less element in it so there will be $n - 1$ recursive calls.
- So `bubble` performs only basic operations each call, and recursively calls itself $n - 1$ times, as such it does $\Theta(n)$ work.

Completed runtime of bubble sort

Now that we know that the work done by `bubble` is linear relative to the length of the `LList` parameter we can calculate the work performed by `repeatBubble` and thus the work performed by `bubbleSort`.

Completed runtime of bubble sort

Now that we know that the work done by `bubble` is linear relative to the length of the `LList` parameter we can calculate the work performed by `repeatBubble` and thus the work performed by `bubbleSort`.

The amount of work performed by `repeatBubble` is then $n \times \Theta(n) \rightarrow \Theta(n^2)$

Completed runtime of bubble sort

Now that we know that the work done by `bubble` is linear relative to the length of the `LList` parameter we can calculate the work performed by `repeatBubble` and thus the work performed by `bubbleSort`.

The amount of work performed by `repeatBubble` is then $n \times \Theta(n) \rightarrow \Theta(n^2)$

So `bubbleSort` also has a quadratic runtime and also cannot be used to write `findPairs` faster than $\Theta(n^2)$.

Some improvements on our `bubbleSort` implementation

There are some notes to be made on our `bubbleSort` implementation, which can be observed from our card example.

- We do not need to perform n bubblings but rather $n - 1$ bubblings
- Each bubbling does not need to bubble *every* adjacent pair, because after k bubblings the last k items are already guaranteed to be in their sorted position

Neither of these changes will change the asymptotic runtime of `bubbleSort` but they will improve our implementation!

Practice Question: Update our implementation of `bubbleSort` with the improvements noted above.

Table of Contents

An observation from a *different* problem

We now will take a pause from implementing sorting algorithms directly and instead consider a different problem that will help to give rise to a different solution to the problem of sorting.

An observation from a *different* problem

We now will take a pause from implementing sorting algorithms directly and instead consider a different problem that will help to give rise to a different solution to the problem of sorting.

In some card games there is sometimes an extra hand dealt out to a non-existent player often called a dummy hand. Further in some games an individual player may get to pick up and use the cards from this dummy hand at some point in the game.

An observation from a *different* problem

We now will take a pause from implementing sorting algorithms directly and instead consider a different problem that will help to give rise to a different solution to the problem of sorting.

In some card games there is sometimes an extra hand dealt out to a non-existent player often called a dummy hand. Further in some games an individual player may get to pick up and use the cards from this dummy hand at some point in the game.

You are the player who gets the dummy hand at a certain point, after you have already sorted your own hand. By whatever mechanism the dummy hand is also already sorted. How can you effectively combine these hands to keep your hand sorted?

An observation from a *different* problem

We now will take a pause from implementing sorting algorithms directly and instead consider a different problem that will help to give rise to a different solution to the problem of sorting.

An observation from a *different* problem

We now will take a pause from implementing sorting algorithms directly and instead consider a different problem that will help to give rise to a different solution to the problem of sorting.

In some card games there is sometimes an extra hand dealt out to a non-existent player often called a dummy hand. Further in some games an individual player may get to pick up and use the cards from this dummy hand at some point in the game.

An observation from a *different* problem

We now will take a pause from implementing sorting algorithms directly and instead consider a different problem that will help to give rise to a different solution to the problem of sorting.

In some card games there is sometimes an extra hand dealt out to a non-existent player often called a dummy hand. Further in some games an individual player may get to pick up and use the cards from this dummy hand at some point in the game.

You are the player who gets the dummy hand at a certain point, after you have already sorted your own hand. By whatever mechanism the dummy hand is also already sorted. How can you effectively combine these hands to keep your hand sorted?

The merge algorithm

The algorithm we employed on our two sorted hands of cards can more generally be applied to any two sorted sequences to produce a combined sorted sequence. We'll call this function the `merge` function and write it in Python for two `LLists`

The `merge` function

Our merge function operates the same as our example did on our two hands of cards, comparing the first of each `LList` and determining which should go in the resultant `LList` first.

```
def merge(sl1, sl2):
  if first(sl1) < first(sl2):
    return cons(first(sl1), ...)
  else:
    return cons(first(sl2), ...)
```

The `merge` function

But what `LList` should the selected item be the first of?

```
def merge(sl1, sl2):
  if first(sl1) < first(sl2):
    return cons(first(sl1), ...)
  else:
    return cons(first(sl2), ...)
```

The `merge` function

But what `LList` should the selected item be the first of?

We want it to be the first in our newly sorted `LList`, so the `LList` we should `cons` it onto is the result of merging the *rest* of our values.

```
def merge(sl1, sl2):
  if first(sl1) < first(sl2):
    return cons(first(sl1), ...)
  else:
    return cons(first(sl2), ...)
```

The `merge` function

In this case the "rest of our values" refers to the values we have not yet included in our final answer, which is all of them except for the one selected.

```
def merge(sl1, sl2):
  if first(sl1) < first(sl2):
    return cons(first(sl1), merge(rest(sl1), sl2))
  else:
    return cons(first(sl2), merge(sl1, rest(sl2)))
```

The `merge` function

In this case the "rest of our values" refers to the values we have not yet included in our final answer, which is all of them except for the one selected.

For this reason in each case we do not change the `LList` we did not take an item from when passing it as a recursive argument, just as only the hand which we took a card from changed in each step.

```
def merge(sl1, sl2):
  if first(sl1) < first(sl2):
    return cons(first(sl1), merge(rest(sl1), sl2))
  else:
    return cons(first(sl2), merge(sl1, rest(sl2)))
```

The `merge` function

We have written our recursive cases, but now must write our base case! Since in each recursion we are making either `sl1` or `sl2` smaller we must have a base case for each `LList`.

```
def merge(sl1, sl2):
  if first(sl1) < first(sl2):
    return cons(first(sl1), merge(rest(sl1), sl2))
  else:
    return cons(first(sl2), merge(sl1, rest(sl2)))
```

The `merge` function

We have written our recursive cases, but now must write our base case! Since in each recursion we are making either `sl1` or `sl2` smaller we must have a base case for each `LList`.

If one of our `LLists` is empty then the result of merging a non-empty `LList` with an empty `LList` is simply the non-empty `LList`!

```
def merge(sl1, sl2):
  if first(sl1) < first(sl2):
    return cons(first(sl1), merge(rest(sl1), sl2))
  else:
    return cons(first(sl2), merge(sl1, rest(sl2)))
```

The merge function

With our base cases complete our merge function is now finished.

```
def merge(sl1, sl2):
  if isEmpty(sl1):
    return sl2
  if isEmpty(sl2):
    return sl1
  if first(sl1) < first(sl2):
    return cons(first(sl1), merge(rest(sl1), sl2))
  else:
    return cons(first(sl2), merge(sl1, rest(sl2)))
```

Considering the runtime of `merge`

How many basic operations does `merge` perform? Since `merge` has two inputs which both affect its execution we must specify in terms of both.

Considering the runtime of `merge`

How many basic operations does `merge` perform? Since `merge` has two inputs which both affect its execution we must specify in terms of both.

If we consider the length of `sl1` to be h and the length of `sl2` to be d then how many basic operations are performed by `merge`?

Considering the runtime of `merge`

How many basic operations does `merge` perform? Since `merge` has two inputs which both affect its execution we must specify in terms of both.

If we consider the length of `sl1` to be h and the length of `sl2` to be d then how many basic operations are performed by `merge`?

`merge` only performs basic operations other than its recursive call, and only executes one recursive call each time it executes, so we simply need to determine how many times `merge` recursively calls itself.

Considering the runtime of `merge`

`merge` reaches a base case when *one* of the `LLists` is empty. How many items must be looked at is determined by the values within the `LLists`.

Considering the runtime of `merge`

`merge` reaches a base case when *one* of the `LLists` is empty. How many items must be looked at is determined by the values within the `LLists`.

Imagine that `sl1` is values $v_0, v_1, ..., v_h$ and `sl2` is values $w_0, w_1, ..., w_d$, and that v_0 is larger than w_d. How many times does `merge` run?

Considering the runtime of `merge`

`merge` reaches a base case when *one* of the `LLists` is empty. How many items must be looked at is determined by the values within the `LLists`.

Imagine that `sl1` is values $v_0, v_1, ..., v_h$ and `sl2` is values $w_0, w_1, ..., w_d$, and that v_0 is larger than w_d. How many times does `merge` run?

In that case `merge` is called d times, because every item in `sl2` is smaller than even the smallest item in `sl1`. After `merge` is called d times then `sl2` is empty and a base case is reached.

Considering the runtime of `merge`

So, in the best case `merge` runs as many times as the length of *one* of it's input `LLists`, and for truly best case that length is the smaller one. So in the best case `merge` is $\Theta(min(h, d))$

Considering the runtime of `merge`

So, in the best case `merge` runs as many times as the length of *one* of it's input `LLists`, and for truly best case that length is the smaller one. So in the best case `merge` is $\Theta(min(h, d))$

However, we know we are interested in the worst case. So what is it the worst case?

Considering the runtime of `merge`

So, in the best case `merge` runs as many times as the length of *one* of it's input `LLists`, and for truly best case that length is the smaller one. So in the best case `merge` is $\Theta(min(h, d))$

However, we know we are interested in the worst case. So what is it the worst case?

Imagine that $sl1$ and $sl2$ are the first twenty even and the first twenty odd non-zero naturals. So `sl1` is the `LList` $2, 4, ..., 40$ and `sl2` is $1, 3, ..., 39$

Considering the runtime of `merge`

So, in the best case `merge` runs as many times as the length of *one* of it's input `LLists`, and for truly best case that length is the smaller one. So in the best case `merge` is $\Theta(min(h, d))$

However, we know we are interested in the worst case. So what is it the worst case?

Imagine that $sl1$ and $sl2$ are the first twenty even and the first twenty odd non-zero naturals. So `sl1` is the `LList` $2, 4, ..., 40$ and `sl2` is $1, 3, ..., 39$

In this case we have to repeatedly switch which `LList` we make smaller each recursion, and so we will actually look at *every* value in each `LList`.

Considering the runtime of `merge`

Our `merge` function can't do worse than calling itself once for each value, since it removes that value from the recursion each time it chooses it. So since we've found a case where it is called for every value in our two input `LLists`, and it can't do worse than that, we have found our worst case.

So the worst-case asymptotic runtime for `merge` is $\Theta(h + d)$. We also note that $h + d$ is the length of the resultant `LList`, so this is linear in the length of the result.

Waht use is `merge`?

So the `merge` function allows us to build a sorted `LList` from two already sorted `LLists`.

Waht use is `merge`?

So the `merge` function allows us to build a sorted `LList` from two already sorted `LLists`.

But how does this help us to sort an arbitrary `LList`?

Waht use is `merge`?

So the `merge` function allows us to build a sorted `LList` from two already sorted `LLists`.

But how does this help us to sort an arbitrary `LList`?

Well with the recursive way of thinking we've developed throughout this course it actually helps us quite a lot!

Writing `mergeSort`

We will now write a sorting algorithm that uses `merge` and the power of recursion to sort a `LList`.

```
def mergeSort(l):
```

Writing `mergeSort`

As with most of our recursions we will start with our base case. Once again we note that an empty `LList` or a `LList` of one element is trivially sorted.

```
def mergeSort(l):
  if len(l) <= 1:
    return l
```

Writing `mergeSort`

Next we must decide how to recursively call ourselves. If our goal is to use `merge` to help us solve the problem then we must end up with *two* sorted `LLists`.

```
def mergeSort(l):
  if len(l) <= 1:
    return l
```

Writing `mergeSort`

Next we must decide how to recursively call ourselves. If our goal is to use `merge` to help us solve the problem then we must end up with *two* sorted `LLists`.

Since `mergeSort` is a function that promises it will return the sorted version of a `LList`, and we need two sorted `LLists` to provided to `merge` we can recursively call `mergeSort` to produce the arguments for `merge`!

```
def mergeSort(l):
  if len(l) <= 1:
    return l
```

Writing `mergeSort`

Which `LLists` should we recursively call `mergeSort` on? The elements we provide to each call must be distinct from those provided to the other, so that we do not duplicate elements in our final result.

```
def mergeSort(l):
  if len(l) <= 1:
    return l
  return merge(mergeSort(???), mergeSort(???))
```

Writing `mergeSort`

Which `LLists` should we recursively call `mergeSort` on? The elements we provide to each call must be distinct from those provided to the other, so that we do not duplicate elements in our final result.

So we need to somehow split up `l` into two `LLists` to sort. In order to help us do so we'll write two helper functions `skipN` and `takeN`

```
def mergeSort(l):
  if len(l) <= 1:
    return l
  return merge(mergeSort(???), mergeSort(???))
```

Writing `mergeSort`

`takeN` and `skipN` are shown below, simple recursive functions.

```
def takeN(l, n):
  # Returns a LList of the first n items of l
  if n == 0:
    return empty()
  return cons(first(l), takeN(rest(l), n-1))

def skipN(l, n):
  # Returns a LList of all except the first n items of l
  if n == 0:
    return l
  return skipN(rest(l), n-1)
```

Writing `mergeSort`

With `skipN` and `takeN` we can easily produce two portions of our parameter `l` to call `mergeSort` on. The question now is how many elements to take for each recursive call?

```
def mergeSort(l):
  if len(l) <= 1:
    return l
  return merge(mergeSort(???), mergeSort(???))
```

Writing `mergeSort`

With `skipN` and `takeN` we can easily produce two portions of our parameter `l` to call `mergeSort` on. The question now is how many elements to take for each recursive call?

A natural choice is to split the `LList` in *half* each recursion.

```
def mergeSort(l):
  if len(l) <= 1:
    return l
  return merge(mergeSort(???), mergeSort(???))
```

Writing `mergeSort`

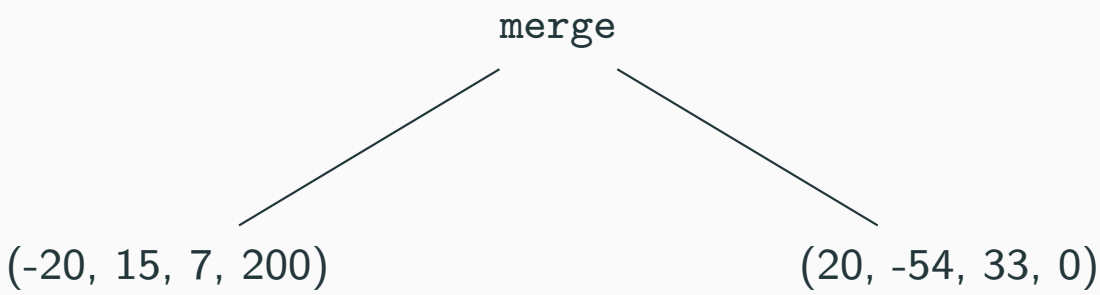
Below is shown our completed `mergeSort`.

```
def mergeSort(l):
  if len(l) <= 1:
    return l
  leftHalf = takeN(l, len(l)//2)
  rightHalf = skipN(l, len(l)//2)
  return merge(mergeSort(leftHalf), mergeSort(rightHalf))
```

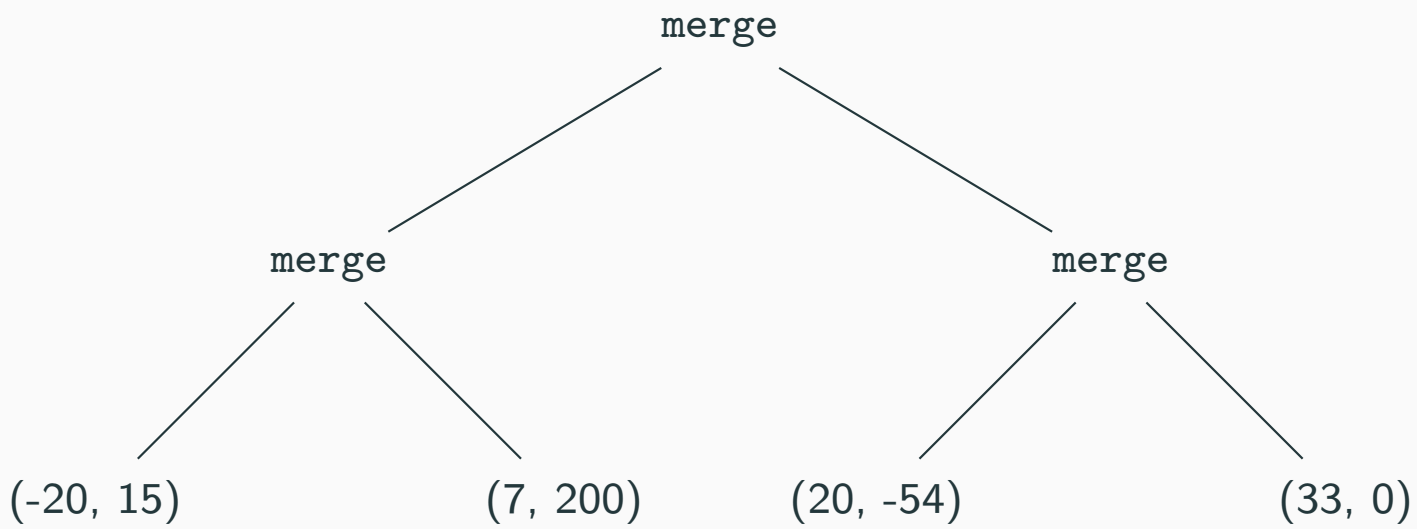
Example application of `mergeSort`

(-20, 15, 7, 200, 20, -54, 33, 0)

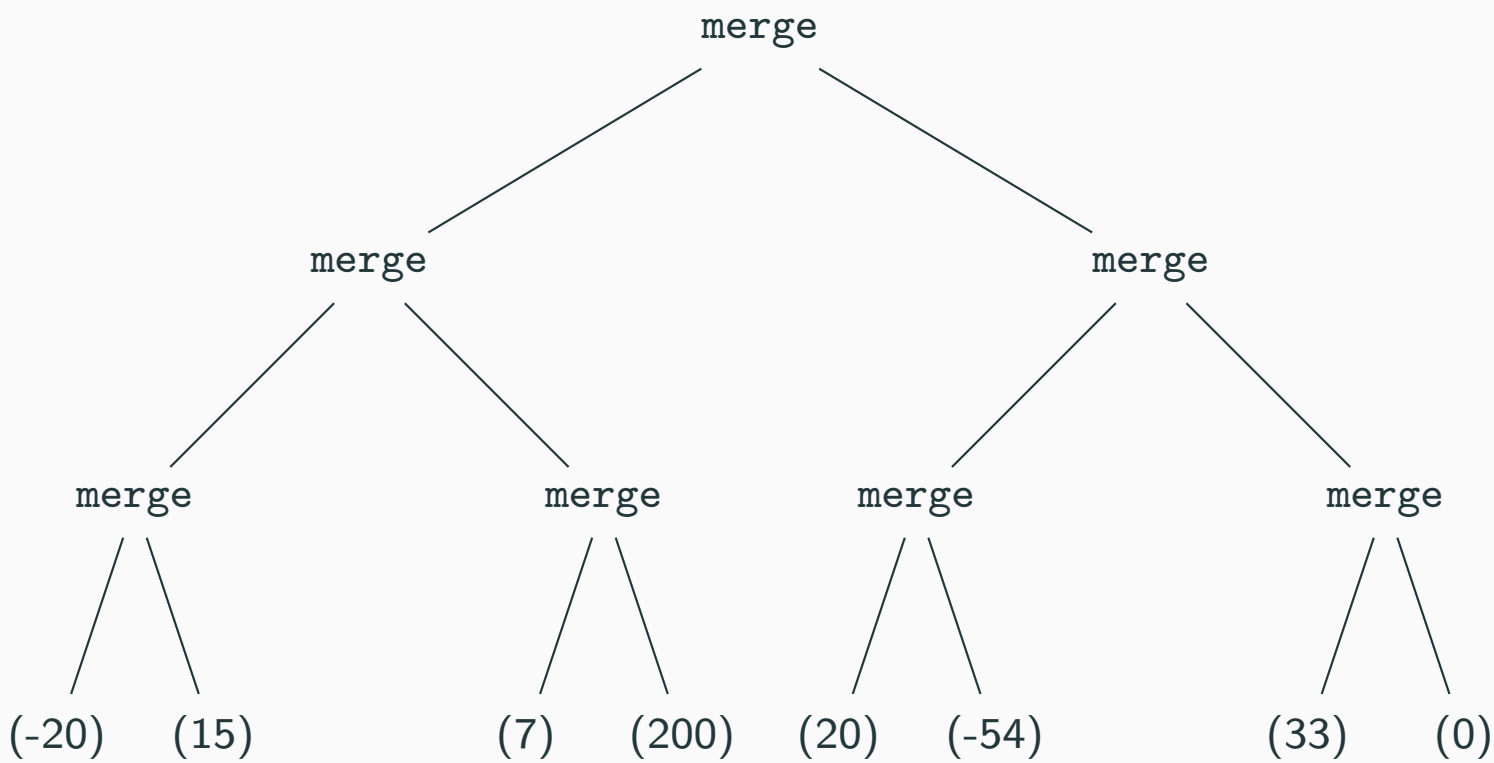
Example application of `mergeSort`



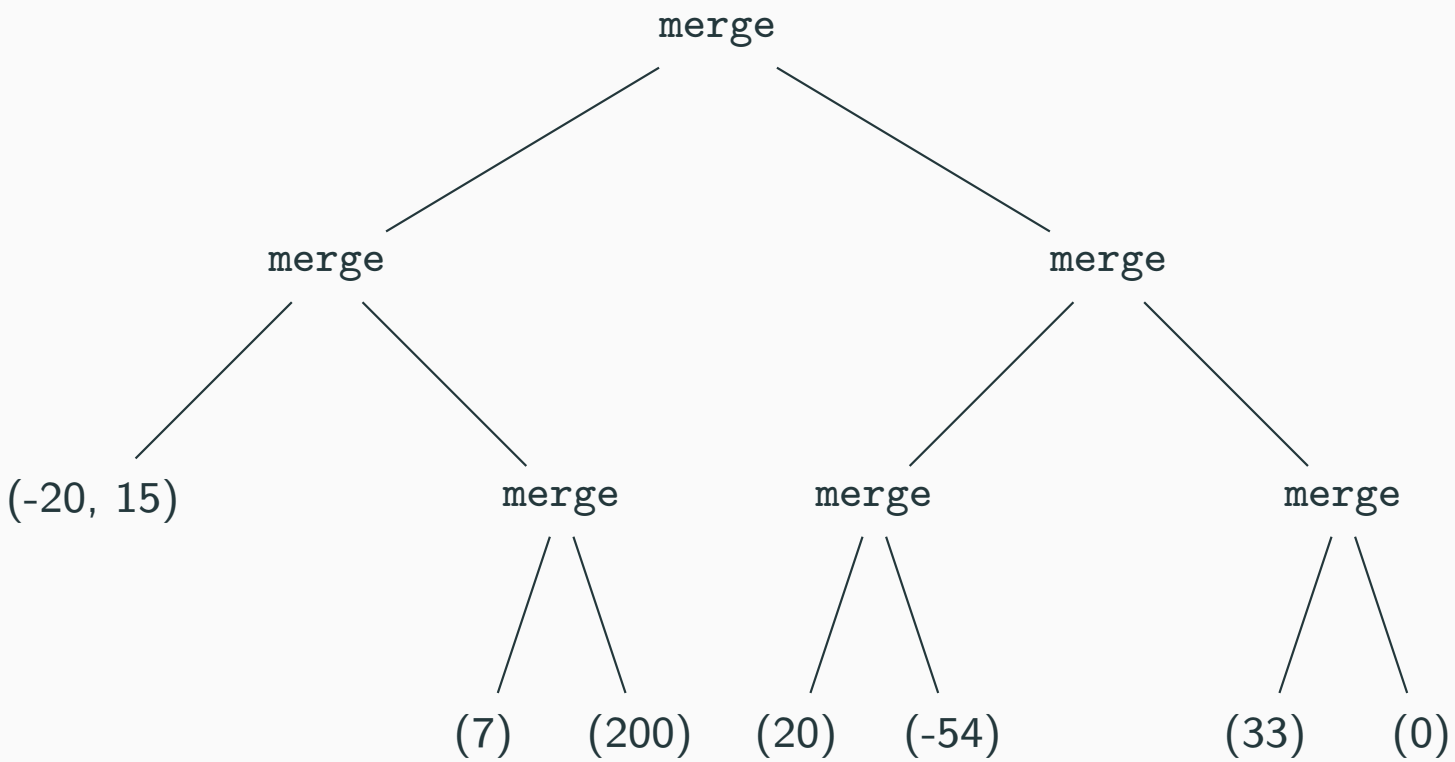
Example application of `mergeSort`



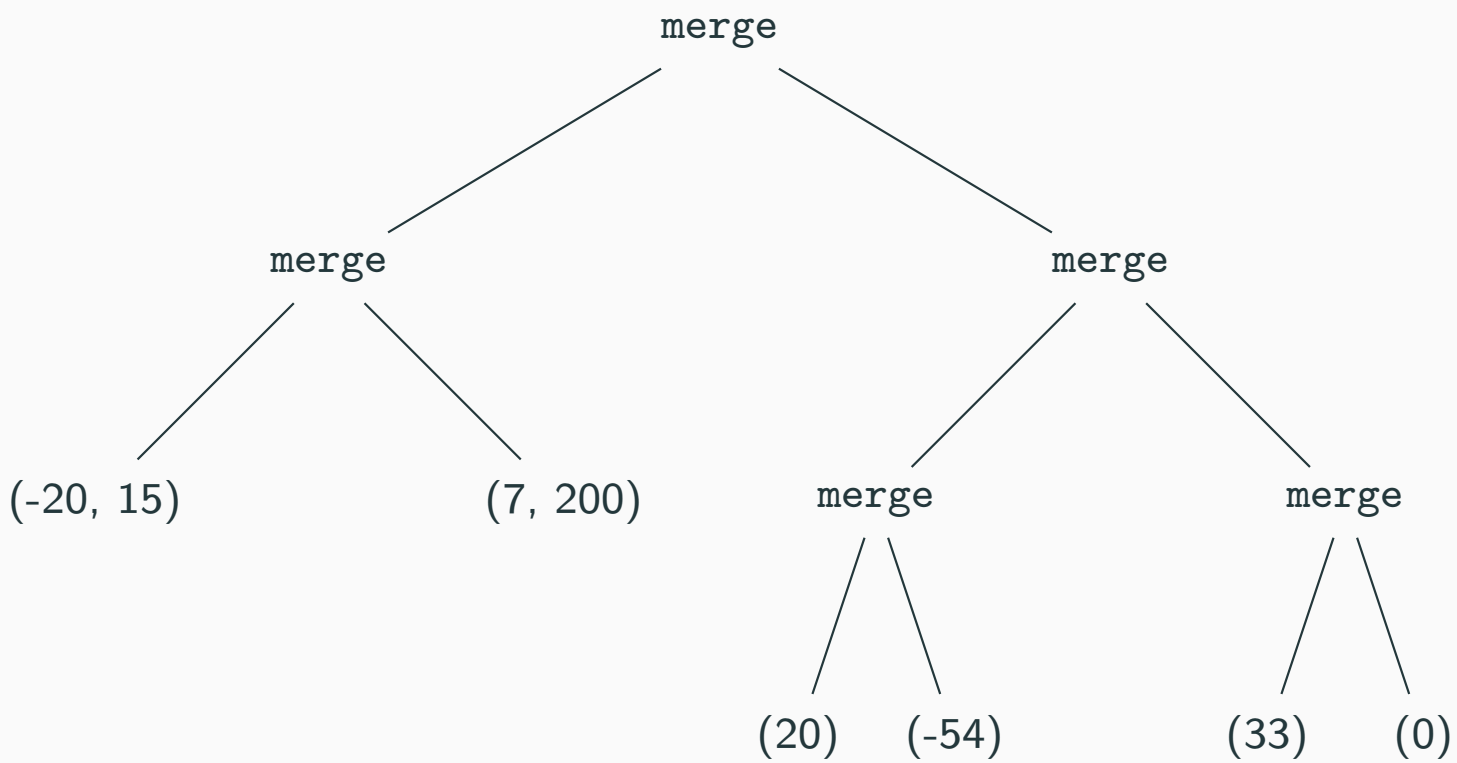
Example application of `mergeSort`



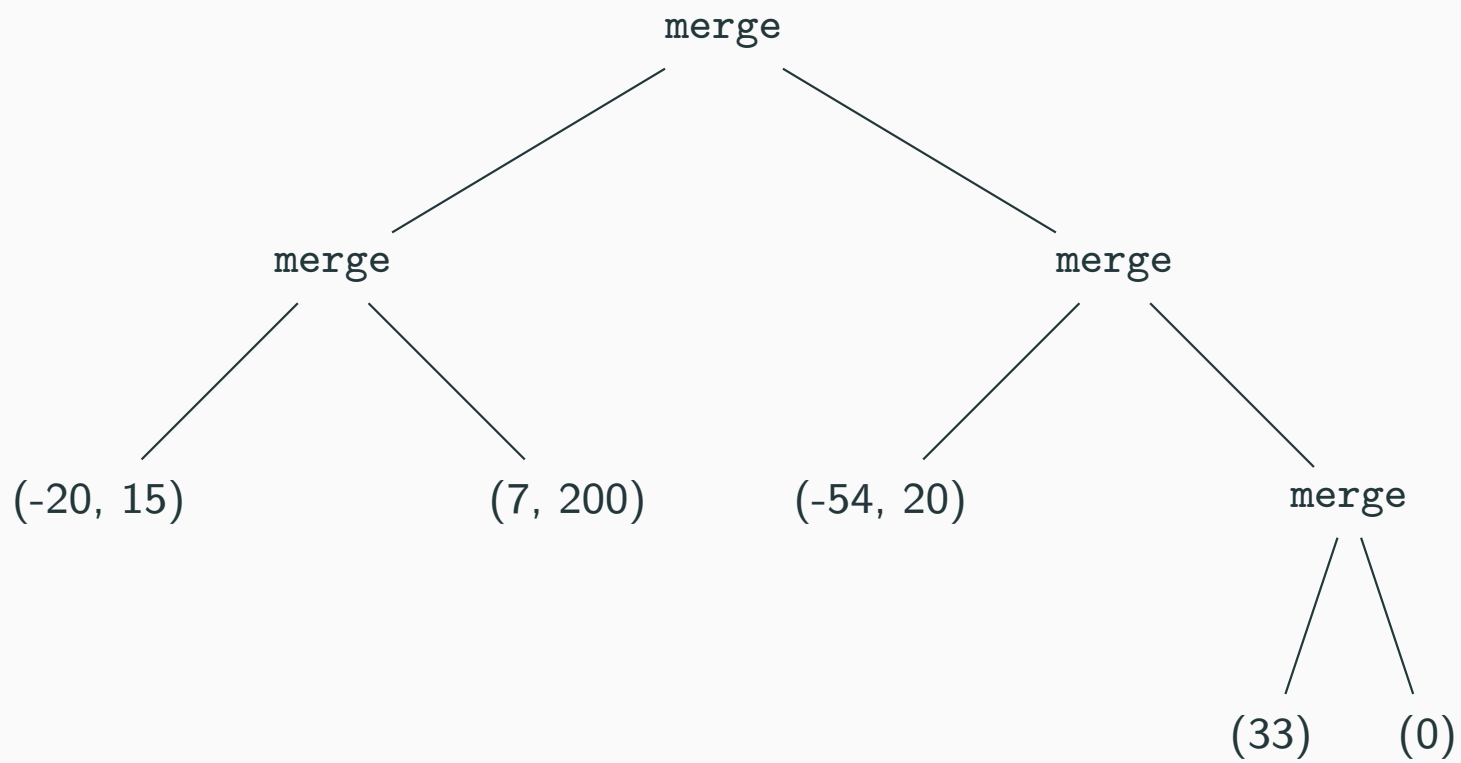
Example application of `mergeSort`



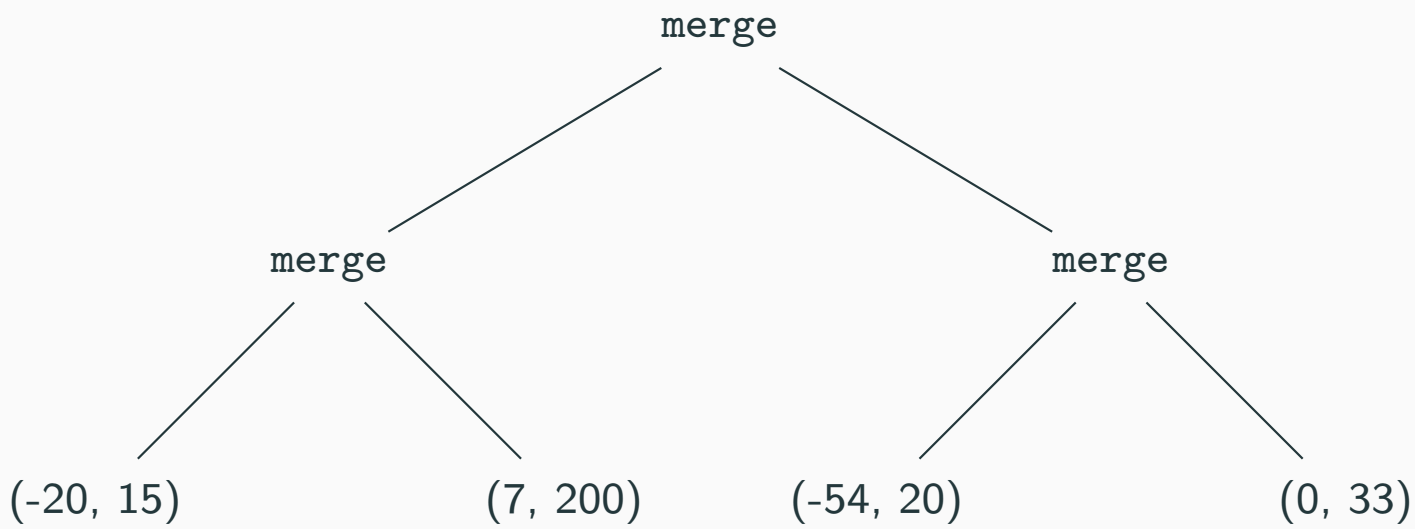
Example application of `mergeSort`



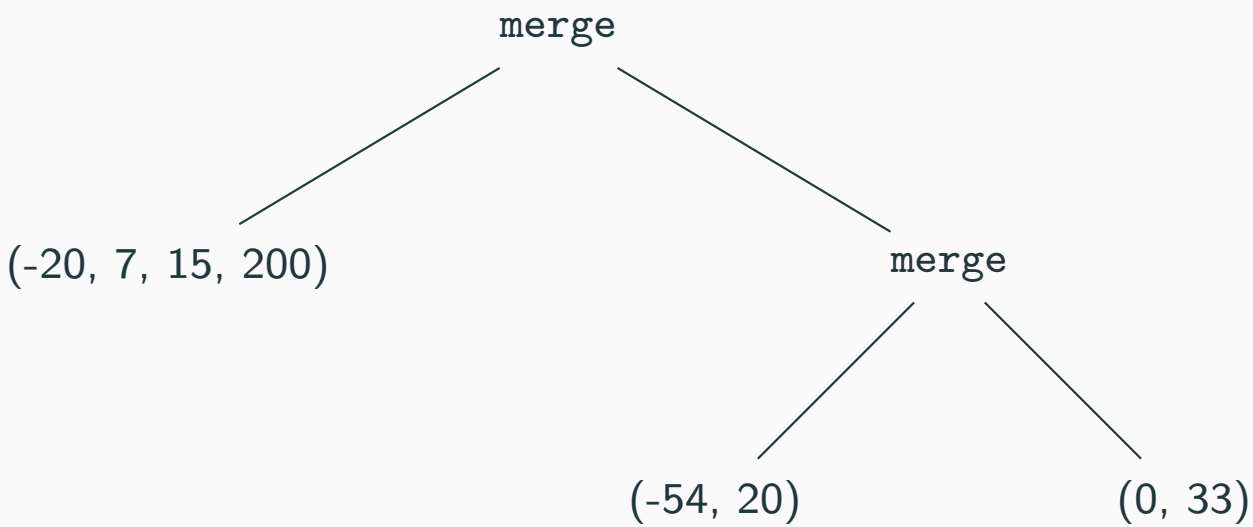
Example application of `mergeSort`



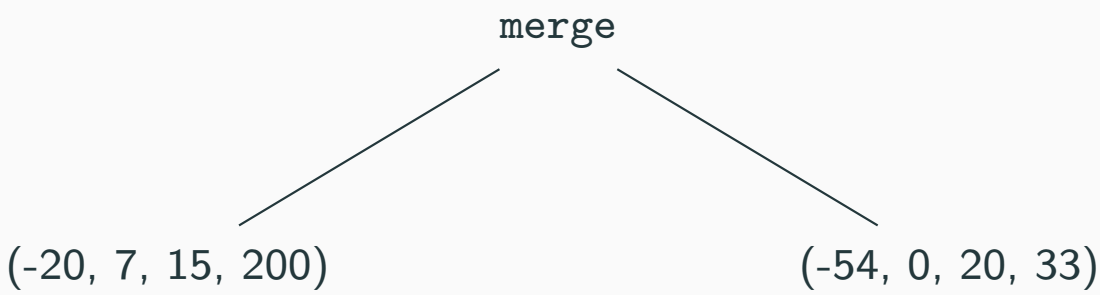
Example application of `mergeSort`



Example application of `mergeSort`



Example application of `mergeSort`



Example application of `mergeSort`

(-54, -20, 0, 7, 15, 20, 33, 200)

Runtime of `mergeSort`

What then is the asymptotic runtime of `mergeSort`?

Runtime of `mergeSort`

What then is the asymptotic runtime of `mergeSort`?

First we must think about how much work each application of `mergeSort` does in terms of its input length.

Runtime of `mergeSort`

What then is the asymptotic runtime of `mergeSort`?

First we must think about how much work each application of `mergeSort` does in terms of its input length.

Each call to `mergeSort` calls `skipN`, `takeN`, and `merge` once. If the length of the `LList` given to `mergeSort` is n then:

- The calls to `skipN` and `takeN` both do $\frac{n}{2}$ work
- The call to `merge` does n work
- Therefore each call to `mergeSort` does $2n$ work

The call to `merge` does So each call to `mergeSort` is doing a linear amount of work relative to the input size.

Runtime of `mergeSort`

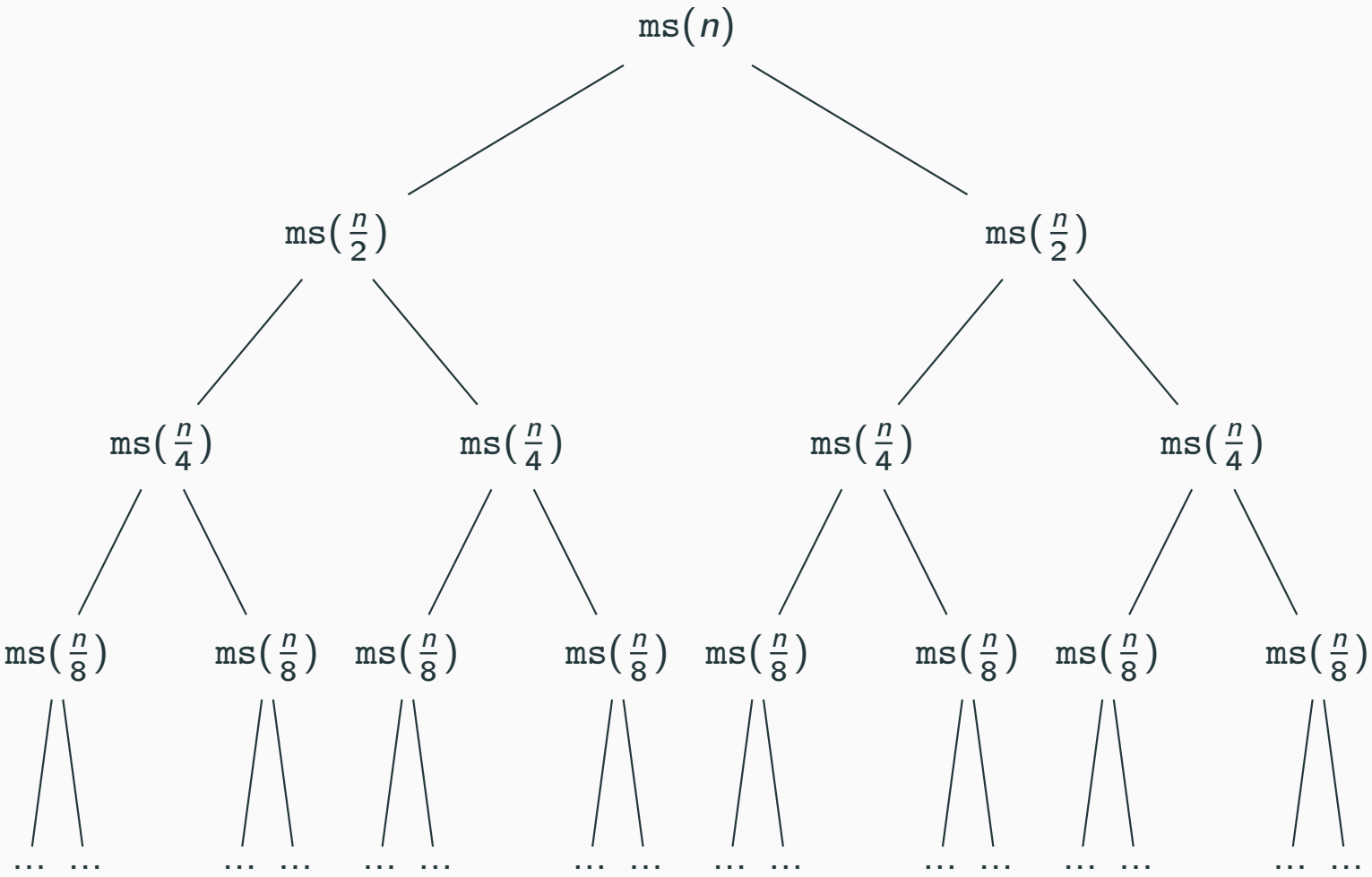
However, unlike our previous functions we cannot simply calculate how many times we call `mergeSort` and multiply by that by n because each call to `mergeSort` isn't doing an amount of work relative to our original input n but their own very different argument!

Runtime of `mergeSort`

However, unlike our previous functions we cannot simply calculate how many times we call `mergeSort` and multiply by that by n because each call to `mergeSort` isn't doing an amount of work relative to our original input n but their own very different argument!

To understand how much work `mergeSort` does we will draw a diagram and interpret it. Remember that each call to `mergeSort` is doing an amount of work linearly relative to its input size.

Analyzing `mergeSort`



Look at each "row" of our diagram. How much work is actually being done in each row?

Analyzing `mergeSort`

If we number our rows starting at zero and increasing by one each row we go down then the number of calls to `mergeSort` performed in row i is 2^i. That's a lot of calls very quickly!

Analyzing `mergeSort`

If we number our rows starting at zero and increasing by one each row we go down then the number of calls to `mergeSort` performed in row i is 2^i. That's a lot of calls very quickly!

However, how much work is actually being performed in each of these rows? We know each `mergeSort` does work linear to its input size other than its recursive calls which are depicted in the lower rows, so the recursive work is calculated separately.

Analyzing `mergeSort`

If we number our rows starting at zero and increasing by one each row we go down then the number of calls to `mergeSort` performed in row i is 2^i. That's a lot of calls very quickly!

However, how much work is actually being performed in each of these rows? We know each `mergeSort` does work linear to its input size other than its recursive calls which are depicted in the lower rows, so the recursive work is calculated separately.

In each row we have 2^i calls, but each call in that row is only doing $\frac{n}{2^i}$ work! So each row is doing an amount of work equal to $2^i \cdot \frac{n}{2^i} \rightarrow \frac{2^i n}{2^i} \rightarrow n$

Analyzing `mergeSort`

Since we now know each row is really only performing n work, where n is the size of our original input we need only to find how many rows there will be for a `LList` of length n and multiply!

Analyzing `mergeSort`

Since we now know each row is really only performing n work, where n is the size of our original input we need only to find how many rows there will be for a `LList` of length n and multiply!

Each time we recursively call ourselves we divide n by two, and we stop when our `LList` is empty or length one. So since the input to each row has half the length of the previous row we simply need to answer the question "How any times can we divide n by two before it becomes one?"

Analyzing `mergeSort`

Since we now know each row is really only performing n work, where n is the size of our original input we need only to find how many rows there will be for a `LList` of length n and multiply!

Each time we recursively call ourselves we divide n by two, and we stop when our `LList` is empty or length one. So since the input to each row has half the length of the previous row we simply need to answer the question "How any times can we divide n by two before it becomes one?"

The logarithm function is exactly the function that answers that question! As such, the number of rows we will have when `mergeSort` is called on a `LList` of length n is then $\log_2 n$

Analyzing `mergeSort`

This means the runtime of `mergeSort` is then $\Theta(n log_2 n)$ — which is much better than the $\Theta(n^2)$ of bubble sort and insertion sort!

Analyzing `mergeSort`

This means the runtime of `mergeSort` is then $\Theta(n log_2 n)$ — which is much better than the $\Theta(n^2)$ of bubble sort and insertion sort!

Remember our `findPairs` function which had a runtime of n^2 for the naive solution on an arbitrary `LList`, but a runtime of n for a sorted `LList`.

Analyzing `mergeSort`

This means the runtime of `mergeSort` is then $\Theta(n log_2 n)$ — which is much better than the $\Theta(n^2)$ of bubble sort and insertion sort!

Remember our `findPairs` function which had a runtime of n^2 for the naive solution on an arbitrary `LList`, but a runtime of n for a sorted `LList`.

Now with `mergeSort` we could write `findPairs` that takes only $n log_2 n$ time on an arbitrary `LList`!

Improving `findPairs`

The algorithm to write `findPairs` on an arbitrary `LList` in $nlog_2n$ time becomes quite simple, and is as follows.

Improving `findPairs`

The algorithm to write `findPairs` on an arbitrary `LList` in $nlog_2n$ time becomes quite simple, and is as follows.

1. Call `mergeSort` to sort the `LList`, this takes $nlog_2n$ time

Improving `findPairs`

The algorithm to write `findPairs` on an arbitrary `LList` in $nlog_2n$ time becomes quite simple, and is as follows.

1. Call `mergeSort` to sort the `LList`, this takes $nlog_2n$ time
2. Call `findPairsSorted` on the result of `mergeSort`, this takes n time

Improving `findPairs`

The algorithm to write `findPairs` on an arbitrary `LList` in $nlog_2n$ time becomes quite simple, and is as follows.

1. Call `mergeSort` to sort the `LList`, this takes $nlog_2n$ time
2. Call `findPairsSorted` on the result of `mergeSort`, this takes n time
3. You are finished, having taken $nlog_2n + n$ time, the smaller term is irrelevant to asymptotic growth

Improving `findPairs`

The algorithm to write `findPairs` on an arbitrary `LList` in $n\log_2 n$ time becomes quite simple, and is as follows.

1. Call `mergeSort` to sort the `LList`, this takes $n\log_2 n$ time
2. Call `findPairsSorted` on the result of `mergeSort`, this takes n time
3. You are finished, having taken $n\log_2 n + n$ time, the smaller term is irrelevant to asymptotic growth
4. So, this algorithm is $\Theta(n\log_2 n)$

Table of Contents

Changing sort comparator

So far all the sorting algorithms we've written have sorted in non-decreasing order. What if we wanted to sort in a different order — for example non-increasing?

Changing sort comparator

So far all the sorting algorithms we've written have sorted in non-decreasing order. What if we wanted to sort in a different order — for example non-increasing?

As it stands now, we'd have to write a whole new function to sort differently. Consider the two implementations of merge sort on the following slides.

Merge sort in non-decreasing order

```
def mergeSort(l):
  def merge(sl1, sl2):
    if isEmpty(sl1):
      return sl2
    if isEmpty(sl2):
      return sl1
    if first(sl1) < first(sl2):
      return cons(first(sl1), merge(rest(sl1), sl2))
    return cons(first(sl2), merge(sl1, rest(sl2)))
  if len(l) <= 1:
    return l
  leftHalf = mergeSort(takeN(l, len(l)//2))
  rightHalf = mergeSort(skipN(l, len(l)//2))
  return merge(leftHalf, rightHalf)
```

Merge sort in non-increasing order

```
def mergeSort(l):
  def merge(sl1, sl2):
    if isEmpty(sl1):
      return sl2
    if isEmpty(sl2):
      return sl1
    if first(sl1) > first(sl2):
      return cons(first(sl1), merge(rest(sl1), sl2))
    return cons(first(sl2), merge(sl1, rest(sl2)))
  if len(l) <= 1:
    return l
  leftHalf = mergeSort(takeN(l, len(l)//2))
  rightHalf = mergeSort(skipN(l, len(l)//2))
  return merge(leftHalf, rightHalf)
```

The difference in sorts

How did the two implementations of `mergeSort` differ?

The difference in sorts

How did the two implementations of `mergeSort` differ?

They differed in only *one* way! They differed in how they compared two elements of the `LList`.

The difference in sorts

How did the two implementations of `mergeSort` differ?

They differed in only *one* way! They differed in how they compared two elements of the `LList`.

One version compared `first(sl1) > first(sl2)` while the other compared `first(sl1) < first(sl2)`

The difference in sorts

How did the two implementations of `mergeSort` differ?

They differed in only *one* way! They differed in how they compared two elements of the `LList`.

One version compared `first(sl1) > first(sl2)` while the other compared `first(sl1) < first(sl2)`

As we did when writing other higher order functions we can abstract out the differences of these two implementations by simply adding a parameter to represent this process.

Parameterized `mergeSort`

We simply add a second parameter `cmp` to `mergeSort`. The parameter `cmp` should be the function that takes two parameters *a* and *b* and returns `True` if *a* should come before *b* is a sorted sequence and `False` otherwise.

Such a function is called a *comparator* function as it *compares* elements of the sequence.

Parameterized `mergeSort`

```
def mergeSort(l, cmp):
  def merge(sl1, sl2):
    if isEmpty(sl1):
      return sl2
    if isEmpty(sl2):
      return sl1
    if cmp(first(sl1), first(sl2)):
      return cons(first(sl1), merge(rest(sl1), sl2))
    return cons(first(sl2), merge(sl1, rest(sl2)))
  if len(l) <= 1:
    return l
  leftHalf = mergeSort(takeN(l, len(l)//2), cmp)
  rightHalf = mergeSort(skipN(l, len(l)//2), cmp)
  return merge(leftHalf, rightHalf)
```

Improved sorting

The benefit of adding a comparator function parameter to our sorting functions is that we can specify how we would like our elements sorted each time we call the function without having to write a whole new sorting function.

Improved sorting

The benefit of adding a comparator function parameter to our sorting functions is that we can specify how we would like our elements sorted each time we call the function without having to write a whole new sorting function.

Being able to specify the comparator function doesn't only help us to change the order in which we sort. Previously when we only used the < operator we could only sort `LLists` of data for which the < operator was defined.

Improved sorting

The benefit of adding a comparator function parameter to our sorting functions is that we can specify how we would like our elements sorted each time we call the function without having to write a whole new sorting function.

Being able to specify the comparator function doesn't only help us to change the order in which we sort. Previously when we only used the < operator we could only sort `LLists` of data for which the < operator was defined.

As we now provided the comparator function we can define it to work on whatever type we have stored within our `LList`. For example, we might want to store cards.

Representing playing cards

How can we represent playing cards in our Python program?

Representing playing cards

How can we represent playing cards in our Python program?

We start by identifying that a playing card is nothing more than two pieces of data — a rank and a suit.

Representing playing cards

How can we represent playing cards in our Python program?

We start by identifying that a playing card is nothing more than two pieces of data — a rank and a suit.

Our possible ranks, in order, are the numbers 2 – 10 as well as the Jack, Queen, King, and Ace.

Representing playing cards

How can we represent playing cards in our Python program?

We start by identifying that a playing card is nothing more than two pieces of data — a rank and a suit.

Our possible ranks, in order, are the numbers 2 – 10 as well as the Jack, Queen, King, and Ace.

Our suits, in order, are Diamonds, Clubs, Hearts, and Spades.

Storing cards

So, we define a data type to represent cards as shown below.

```
'''
A Card is a
- LL(Rank, Suit)

A Rank is
 - An integer in the range [2, 14]

A suit is
  - An integer in the range [0,3]
'''
```

Storing cards

Why are our ranks the numbers $2 - 14$ when they should be $2 - 10$ as well as Jack, Queen, King, and Ace?

Even stranger why are our suits $0 - 3$ when they should be Diamonds, Clubs, Hearts, and Spades?

Storing cards

Why are our ranks the numbers $2 - 14$ when they should be $2 - 10$ as well as Jack, Queen, King, and Ace?

Even stranger why are our suits $0 - 3$ when they should be Diamonds, Clubs, Hearts, and Spades?

We need to be able to compare our ranks to each other as well as our suits to each other. The strings that represent our suits or ranks are not easily comparable.

Storing cards

Why are our ranks the numbers $2 - 14$ when they should be $2 - 10$ as well as Jack, Queen, King, and Ace?

Even stranger why are our suits $0 - 3$ when they should be Diamonds, Clubs, Hearts, and Spades?

We need to be able to compare our ranks to each other as well as our suits to each other. The strings that represent our suits or ranks are not easily comparable.

So, we use our functional way of thinking to propose a mapping between integers and ranks as well as integers and suits.

Storing cards

The mapping we propose for our ranks is as follows:

$$2 \longleftrightarrow 2$$

$$3 \longleftrightarrow 3$$

$$4 \longleftrightarrow 4$$

$$5 \longleftrightarrow 5$$

$$6 \longleftrightarrow 6$$

$$7 \longleftrightarrow 7$$

$$8 \longleftrightarrow 8$$

$$9 \longleftrightarrow 9$$

$$10 \longleftrightarrow 10$$

$$11 \longleftrightarrow \textit{Jack}$$

$$12 \longleftrightarrow \textit{Queen}$$

$$13 \longleftrightarrow \textit{King}$$

$$14 \longleftrightarrow \textit{Ace}$$

Storing cards

The mapping we propose for our suits is as follows:

$$0 \longleftrightarrow \textit{Diamonds}$$
$$1 \longleftrightarrow \textit{Clubs}$$
$$2 \longleftrightarrow \textit{Hearts}$$
$$3 \longleftrightarrow \textit{Spades}$$

Both of these mappings are chosen such that the integer representing any rank A is higher than another rank $B's$ integer if A should appear to the right of B in a sorted hand. The same is true for our mapping of ranks.

Suited ordering

Quite often in card games we sort not only based on card rank, but also on card suit. In this sorting the card suit first determines order, and only if two cards have the same suit is rank used to determine which should come first.

Suited ordering

Quite often in card games we sort not only based on card rank, but also on card suit. In this sorting the card suit first determines order, and only if two cards have the same suit is rank used to determine which should come first.

That is, imagine you are given a hand of cards and you want to sort it so that all diamonds come first, then clubs, then hearts, then spades. Within each collection of suits you want the card sorted by rank in non-decreasing order.

Suited ordering

Quite often in card games we sort not only based on card rank, but also on card suit. In this sorting the card suit first determines order, and only if two cards have the same suit is rank used to determine which should come first.

That is, imagine you are given a hand of cards and you want to sort it so that all diamonds come first, then clubs, then hearts, then spades. Within each collection of suits you want the card sorted by rank in non-decreasing order.

Given that a `Card` is a `LL(Rank, Suit)` can you write a comparator function that compares two cards $c1$ and $c2$ and returns `True` if $c1$ should appear before $c2$ in the sorted order described above?

Suited ordering comparator function

The comparator function for two cards becomes quite easy to write, as we have defined our data in such a way that it is natural to work with.

```
def suitOrder(c1, c2):
  if second(c1) < second(c2):
    return True
  if second(c1) > second(c2):
    return False
  return first(c1) < first(c2)
```

Sorting a `LList` of `Cards`

Then sorting a `LList` of `Cards` in suited ordering is as simple as calling `mergeSort` on the `LList` and passing the correct comparator function.

```
hand = LL(LL(13, 2), LL(2, 3), LL(7, 1),
          LL(14, 0), LL(4, 3), LL(10, 0))

mergeSort(hand, suitOrder)
 -> LL(LL(10, 0), LL(14, 0), LL(7, 1),
          LL(13, 2), LL(2, 3), LL(4, 3))
```

The sort is correct, but now our data doesn't really resemble *cards* because of how we've mapped between ranks, suits and integers.

Internal representation versus presentation

Using numbers to represent our suits and all of our ranks is the *internal representation* or the *implementation* we've chosen for our data. The internal representation of data should be chosen in such a way that it aids in the work we need to perform, which is why we chose the mapping we did.

However, when it comes to actually displaying data we often want to show something other than the internal representation. We may choose to call this the *presentation* of the data. This can be achieved simply by constructing a function that takes a particular type of data and returns whatever we want to display it as (e.g. a nice string).

Displaying `Cards` nicely

To improve the presentation of our cards we'll write a function `cardToStr` which can be called whenever we want to actually display a hand of cards. This function should take a `Card` as we've defined them and produce a string. The string should be the rank of the card followed by the suit character which is one of ♦, ♣, ♥, or ♠. For the ranks Jack, Queen, King, and Ace the character J, Q, K, and A should be used.

The `cardToStr` function

```
def cardToStr(c):
  suits = ('♦', '♣', '♥', '♠')
  ranks = ("2", "3", "4", "5", "6", "7", "8", "9",
           "10", "J", "Q", "K", "A")
  return ranks[first(c)-2] + suits[second(c)]
```

Now we can simply map this function onto a `LList` of cards in order to produce a `LList` of the string representation of those cards!

New kinds of data

Considering `Cards` as a data type we've now investigated how we can easily use functions to map between internal representations and presentations we'd like to display.

However, our hand of cards actually demonstrates another concept we have not discussed in detail. If we think of our hand as simply being a `LList` of `Cards` then it is easy to reason about. However, in reality our hand of cards is a `LList` of `LLists`.

When we have collection data types that themselves can contain collections as their elements we call it *multi-dimensional* data. Next we will investigate some of the many use cases of this data.