Revisiting Efficiency and Searching

Rob Hackman

Fall 2025

University of Alberta

Table of Contents

Random Access Data

So far our `LList` has been our only collection of arbitrary items. Our `LList` time is a *sequential access* data type, which means to access the data of our `LList` we must walk through each element of it sequentially. More concretely accessing the n^{th} item of a sequential access data structure would take $\Theta(n)$ time.

Random Access Data

So far our `LList` has been our only collection of arbitrary items. Our `LList` time is a *sequential access* data type, which means to access the data of our `LList` we must walk through each element of it sequentially. More concretely accessing the n^{th} item of a sequential access data structure would take $\Theta(n)$ time.

In contrast, a *random access* data structure provides *constant* access time to any arbitrary item of that collection. Our string type is random accessing, since indexing is constant time. Our strings can only hold characters though — we don't currently have a type to store arbitrary data.

The `tuple` type

We now introduce a built-in Python type we have not used yet the `tuple`. A `tuple` in Python is a sequence, so all sequence operations that were introduced with strings also apply to `tuples`.

The `tuple` type

We now introduce a built-in Python type we have not used yet the `tuple`. A `tuple` in Python is a sequence, so all sequence operations that were introduced with strings also apply to `tuples`.

A `tuple` can be created using the function `tuple`, or in some cases by representing a `tuple` literal using comma separated expressions contained with parentheses[1].

[1] In cases where ambiguity exists, a `tuple` literal can be forced by including a comma after the last value.

The `tuple` type

We now introduce a built-in Python type we have not used yet the `tuple`. A `tuple` in Python is a sequence, so all sequence operations that were introduced with strings also apply to `tuples`.

A `tuple` can be created using the function `tuple`, or in some cases by representing a `tuple` literal using comma separated expressions contained with parentheses[1].

```
tupA = tuple(1, 2, 3)
tupB = ("hello", 12, LL(1, 2, 3))
oneElemTup = (7,)
```

[1] In cases where ambiguity exists, a `tuple` literal can be forced by including a comma after the last value.

Recall sequence operations

All of our sequence operations are also available on `tuples`, and so we now recall them and note their algorithmic complexity.

Syntax	Operation	Complexity
`s[i]`	Indexing	$O(1)$
`s[start:end:skip]`	Slicing	$O(k)$
`s1 + s2`	Concatenation	$O(k)$
`s*c`	Sequence Repetition	$O(k)$
`s1==s2`	Equality	$O(min(m, n))$[2]
`len(s)`	Length	$O(1)$

The above k all represent the length of the resultant sequence.

[2]Here m and n represent the lengths of both sequences respectively.

Recursion over `tuples`

As `tuples` represent a sequence of arbitrary data, we could operate on them similarly to how we operate on our `LLists`. Consider writing the function `sumTuple` and how we might write it.

Recursion over `tuples`

As `tuples` represent a sequence of arbitrary data, we could operate on them similarly to how we operate on our `LLists`. Consider writing the function `sumTuple` and how we might write it.

```
def sumTuple(t):
```

Recursion over `tuples`

As `tuples` represent a sequence of arbitrary data, we could operate on them similarly to how we operate on our `LLists`. Consider writing the function `sumTuple` and how we might write it.

```
def sumTuple(t):
```

Again, our natural base case is when the `tuple` is empty, how do we check that?

Recursion over `tuples`

As `tuples` represent a sequence of arbitrary data, we could operate on them similarly to how we operate on our `LLists`. Consider writing the function `sumTuple` and how we might write it.

```
def sumTuple(t):
  if t == tuple():
    return 0
```

We can construct an empty `tuple` simply by calling the `tuple` function with no arguments. To check if our parameter is empty we can compare equality with the empty `tuple`.

Recursion over `tuples`

As `tuples` represent a sequence of arbitrary data, we could operate on them similarly to how we operate on our `LLists`. Consider writing the function `sumTuple` and how we might write it.

```
def sumTuple(t):
  if t == tuple():
    return 0
```

How can we calculate the "rest" of a `tuple` for our recursive call?

Recursion over `tuples`

As `tuples` represent a sequence of arbitrary data, we could operate on them similarly to how we operate on our `LLists`. Consider writing the function `sumTuple` and how we might write it.

```
def sumTuple(t):
  if t == tuple():
    return 0
  return t[0] + sumTuple(t[1:])
```

Slicing can be used to construct a `tuple` that represents the "rest" of our parameter. However, this is a $O(n)$ operation! This means our `sumTuple` function is $O(n^2)$.

More efficient recursion over `tuples`

Since slicing is an expensive operation to perform on `tuple` just to ignore one element we would rather not perform it to act as our "`rest`" operation. What can we do instead?

More efficient recursion over `tuples`

Since slicing is an expensive operation to perform on `tuple` just to ignore one element we would rather not perform it to act as our "`rest`" operation. What can we do instead?

Use indexing!

More efficient recursion over `tuples`

Since slicing is an expensive operation to perform on `tuple` just to ignore one element we would rather not perform it to act as our "`rest`" operation. What can we do instead?

Use indexing! The entire point of random access data is that *any* element can be accessed in constant time, not just the first! As such, our recursion doesn't need the first item to be the one we're looking at it just needs to know *which* element we're looking at!

Solution — write a helper function that takes a second parameter, which is the index of our current element. Then the problem simply becomes recursion on a natural number, with the `tuple` simply being a parameter that is along for the ride.

Better `sumTuple`

```
def sumTuple(t):
  def sumTupleHelper(i):
    if i >= len(t):
      return 0
    return t[i] + sumTupleHelper(i+1)
  return sumTupleHelper(0)
```

Now, our `sumTuple` function is linear as it should be, as opposed to quadratic!

Better `sumTuple`

```
def sumTuple(t):
  def sumTupleHelper(i):
    if i >= len(t):
      return 0
    return t[i] + sumTupleHelper(i+1)
  return sumTupleHelper(0)
```

Now, our `sumTuple` function is linear as it should be, as opposed to quadratic!

This same issue has existed with most of the functions we've written that operated on strings, because we used slicing. As such, now that we are aware of the time complexities of our sequence operations we should be careful not to use unnecessarily expensive operations!

Practice with `tuples`

Practice problem:Write a function `isSorted` that takes a single `tuple` parameter and returns `True` if the elements of that `tuple` are sorted in either non-decreasing *or* non-increasing order, and returns `False` otherwise.

Table of Contents

Checking membership

Consider we want to write the function `contains` that has two parameters, first a value and second a `LList`, and returns `True` if the given value is contained within the `LList` and `False` otherwise. How would we achieve this?

Checking membership

Consider we want to write the function `contains` that has two parameters, first a value and second a `LList`, and returns `True` if the given value is contained within the `LList` and `False` otherwise. How would we achieve this?

We would have to write a recursive function that compares each item of the `LList` with the given value. This is called *sequential search* because we search each item of our collection sequentially.

Sequential search on a `LList`

```
def contains(v, l):
  if isEmpty(l):
    return False
  if v == first(l):
    return True
  return contains(v, rest(l))
```

This function works perfectly fine, how many basic operations does it perform?

Sequential search on a `LList`

```
def contains(v, l):
  if isEmpty(l):
    return False
  if v == first(l):
    return True
  return contains(v, rest(l))
```

This function works perfectly fine, how many basic operations does it perform?

It depends!

Analysis and variability

We've already discussed time complexity of our functions, however each time we've calculated it we've calculated the exact number of operations a function would take. What about when the exact number of operations a function would take depends on the given arguments?

Analysis and variability

We've already discussed time complexity of our functions, however each time we've calculated it we've calculated the exact number of operations a function would take. What about when the exact number of operations a function would take depends on the given arguments?

Consider the `contains` function we just wrote — the number of operations it performs depends not only on the size of the `LList` but also the location of the element `v` within it!

Analysis and variability

We've already discussed time complexity of our functions, however each time we've calculated it we've calculated the exact number of operations a function would take. What about when the exact number of operations a function would take depends on the given arguments?

Consider the `contains` function we just wrote — the number of operations it performs depends not only on the size of the `LList` but also the location of the element `v` within it!

Consider how many operations the `contains` function executes if the element `v` is

Analysis and variability

We've already discussed time complexity of our functions, however each time we've calculated it we've calculated the exact number of operations a function would take. What about when the exact number of operations a function would take depends on the given arguments?

Consider the `contains` function we just wrote — the number of operations it performs depends not only on the size of the `LList` but also the location of the element `v` within it!

Consider how many operations the `contains` function executes if the element `v` is

- The first element of `l`

Analysis and variability

We've already discussed time complexity of our functions, however each time we've calculated it we've calculated the exact number of operations a function would take. What about when the exact number of operations a function would take depends on the given arguments?

Consider the `contains` function we just wrote — the number of operations it performs depends not only on the size of the `LList` but also the location of the element `v` within it!

Consider how many operations the `contains` function executes if the element `v` is

- The first element of `l`
- The third element of `l`

Analysis and variability

We've already discussed time complexity of our functions, however each time we've calculated it we've calculated the exact number of operations a function would take. What about when the exact number of operations a function would take depends on the given arguments?

Consider the `contains` function we just wrote — the number of operations it performs depends not only on the size of the `LList` but also the location of the element `v` within it!

Consider how many operations the `contains` function executes if the element `v` is

- The first element of `l`
- The third element of `l`
- The middle element of `l`

Analysis and variability

We've already discussed time complexity of our functions, however each time we've calculated it we've calculated the exact number of operations a function would take. What about when the exact number of operations a function would take depends on the given arguments?

Consider the `contains` function we just wrote — the number of operations it performs depends not only on the size of the `LList` but also the location of the element `v` within it!

Consider how many operations the `contains` function executes if the element `v` is

- The first element of `l`
- The third element of `l`
- The middle element of `l`
- Not within `l`

Best, worst, and average case

When considering the analysis of an algorithm we can consider one of three cases — the *best* case, the *worst* case, and the *average* case.

Best, worst, and average case

When considering the analysis of an algorithm we can consider one of three cases — the *best* case, the *worst* case, and the *average* case.

- The best case for `contains` is when `v` is the first element, in which case three basic operations are performed. So the best case is $O(1)$.

Best, worst, and average case

When considering the analysis of an algorithm we can consider one of three cases — the *best* case, the *worst* case, and the *average* case.

- The best case for `contains` is when `v` is the first element, in which case three basic operations are performed. So the best case is $O(1)$.
- The worst case for `contains` is when `v` is not an element of `l`, then we must compare it against every single element of `l`. As such a multiple of n operations are performed and the worst case is $O(n)$.

Best, worst, and average case

When considering the analysis of an algorithm we can consider one of three cases — the *best* case, the *worst* case, and the *average* case.

- The best case for `contains` is when `v` is the first element, in which case three basic operations are performed. So the best case is $O(1)$.
- The worst case for `contains` is when `v` is not an element of `l`, then we must compare it against every single element of `l`. As such a multiple of n operations are performed and the worst case is $O(n)$.
- The average case can be calculated by calculating the work done in each case summing that up and dividing by the number of cases. This comes out to $O(n)$ as well.

Being pessimistic

Each of the best, worst, and average cases can be important for algorithm analysis. However, quite commonly computer scientists opt to be pessimistic, and we typically write the worst case when specifying the complexity of a given function.

Being pessimistic

Each of the best, worst, and average cases can be important for algorithm analysis. However, quite commonly computer scientists opt to be pessimistic, and we typically write the worst case when specifying the complexity of a given function.

The benefits of being pessimistic are that if we need guarantees about complexity then considering the worst case means we have a guarantee the complexity will not be any worse than that.

Being pessimistic

Each of the best, worst, and average cases can be important for algorithm analysis. However, quite commonly computer scientists opt to be pessimistic, and we typically write the worst case when specifying the complexity of a given function.

The benefits of being pessimistic are that if we need guarantees about complexity then considering the worst case means we have a guarantee the complexity will not be any worse than that.

The average case is quite often more realistic, and more relevant for how functions are often used though.

Being pessimistic

Each of the best, worst, and average cases can be important for algorithm analysis. However, quite commonly computer scientists opt to be pessimistic, and we typically write the worst case when specifying the complexity of a given function.

The benefits of being pessimistic are that if we need guarantees about complexity then considering the worst case means we have a guarantee the complexity will not be any worse than that.

The average case is quite often more realistic, and more relevant for how functions are often used though.

Assume in this class if the complexity of a function is given, or asked for, then it is referring to the worst case complexity unless otherwise specified.

Membership of a `tuple`

If instead of writing `contains` to operate on a `LList` we were writing it to operate on a `tuple` could we perform better?

Membership of a `tuple`

If instead of writing `contains` to operate on a `LList` we were writing it to operate on a `tuple` could we perform better?

In the general case, no! No matter what you try and do in the worst case you'll have to look at every single element of the `tuple` in order to determine it does not contain the element you're looking for.

Writing `contains` for a `tuple`

```
def tupleContains(v, t):
  def tcHelper(i):
    if i >= len(t):
      return False
    if t[i] == v
      return True
    return tcHelper(i+1)
  return tcHelper(0)
```

Here is one implementation of `contains` written for a `tuple`.

Writing `contains` for a `tuple`

```
def tupleContains(v, t):
  def tcHelper(i):
    if i >= len(t):
      return False
    if t[i] == v
      return True
    return tcHelper(i+1)
  return tcHelper(0)
```

Here is one implementation of `contains` written for a `tuple`.

Is there ever a time that searching for an element could perform better than linearly?

Writing `contains` for a `tuple`

```
def tupleContains(v, t):
  def tcHelper(i):
    if i >= len(t):
      return False
    if t[i] == v
      return True
    return tcHelper(i+1)
  return tcHelper(0)
```

Here is one implementation of `contains` written for a `tuple`.

Is there ever a time that searching for an element could perform better than linearly? Not for the general case...

A slightly different problem

Searching an arbitrary collection of data for an element is going to take an amount of work that grows linearly with the collection size. What if the collection wasn't arbitrary though?

A slightly different problem

Searching an arbitrary collection of data for an element is going to take an amount of work that grows linearly with the collection size. What if the collection wasn't arbitrary though?

Consider a slightly modified problem. We have a collection of data that is *sorted* in non-decreasing order. Can we check for membership in this sorted collection in sublinear time?

A slightly different problem

Searching an arbitrary collection of data for an element is going to take an amount of work that grows linearly with the collection size. What if the collection wasn't arbitrary though?

Consider a slightly modified problem. We have a collection of data that is *sorted* in non-decreasing order. Can we check for membership in this sorted collection in sublinear time?

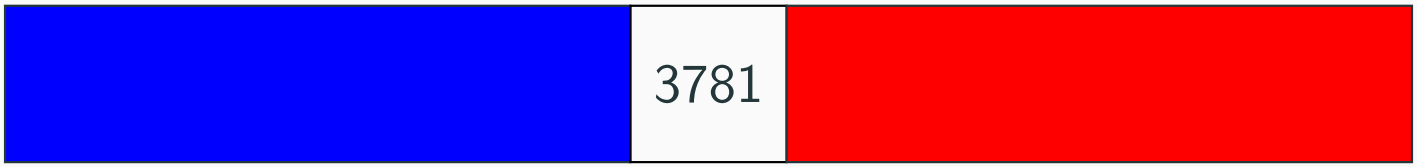
We have random access, yes we can!

Minimizing examined values

Consider we are trying to determine if the value 1023 exists in a sorted `tuple` that contains 5000 elements. Remembering that we can only look at one element at a time, which element guarantees it will maximize the amount of information we gain in the worst case?

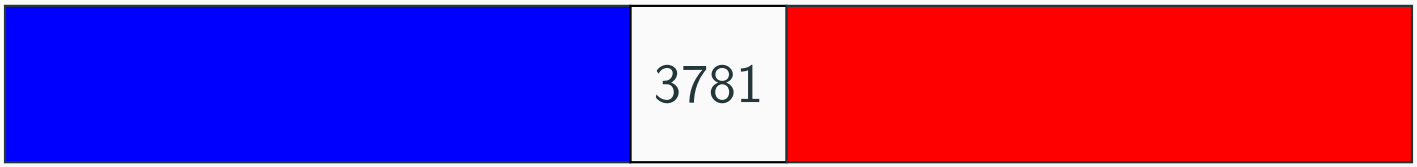
Minimizing examined values

Consider we are trying to determine if the value 1023 exists in a sorted `tuple` that contains 5000 elements. Remembering that we can only look at one element at a time, which element guarantees it will maximize the amount of information we gain in the worst case?



Minimizing examined values

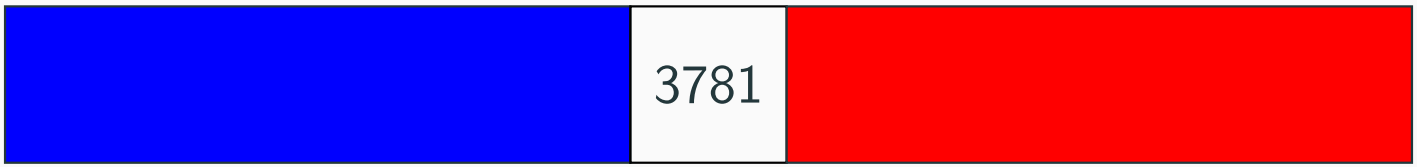
Consider we are trying to determine if the value 1023 exists in a sorted `tuple` that contains 5000 elements. Remembering that we can only look at one element at a time, which element guarantees it will maximize the amount of information we gain in the worst case?



The middle, it turns out, is a good choice. So we look at the value at our middle index of 2500, and see the value of 3781. What does that tell us about the values in our `tuple`?

Minimizing examined values

Consider we are trying to determine if the value 1023 exists in a sorted `tuple` that contains 5000 elements. Remembering that we can only look at one element at a time, which element guarantees it will maximize the amount of information we gain in the worst case?



The middle, it turns out, is a good choice. So we look at the value at our middle index of 2500, and see the value of 3781. What does that tell us about the values in our `tuple`?

The values in the `tuple` that appear *after* index 2500 are guaranteed not to be the value we're looking for. As such, we can eliminate that entire half of our `tuple`!

Binary search

In our previous slide we determined that first examining the halfway point of our `tuple` allowed us to eliminate half of all the values we needed to consider. What do we do after that?

Binary search

In our previous slide we determined that first examining the halfway point of our `tuple` allowed us to eliminate half of all the values we needed to consider. What do we do after that?

We can reuse the same logic. Since we now know only half of our `tuple` is releveant, we can halve it again!

Binary search

In our previous slide we determined that first examining the halfway point of our `tuple` allowed us to eliminate half of all the values we needed to consider. What do we do after that?

We can reuse the same logic. Since we now know only half of our `tuple` is releveant, we can halve it again!

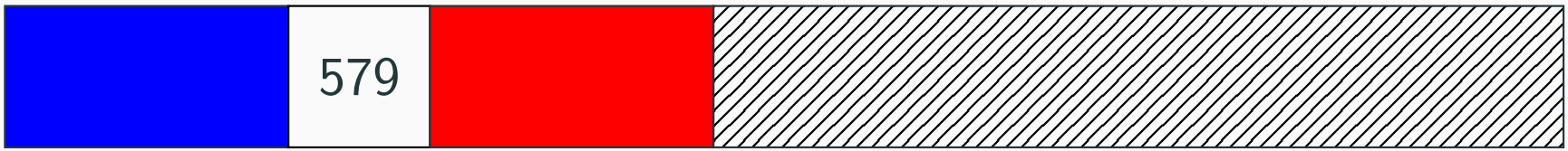
Repeating this process of halving until we find our element or we've ran out of values is the algorithm known as *binary search*.

Binary search example

Imagine again, we are searching for the value 1032 in a sequence.

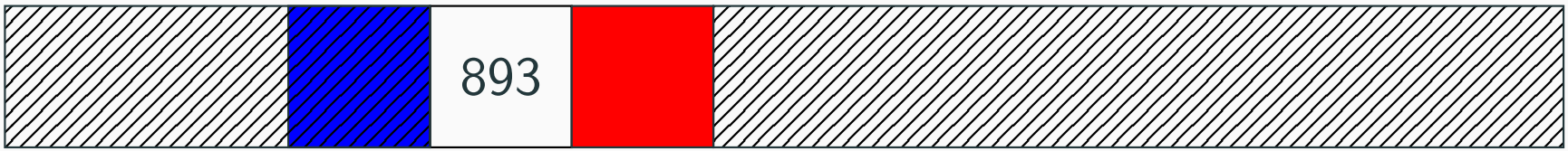
Binary search example

Imagine again, we are searching for the value 1032 in a sequence.



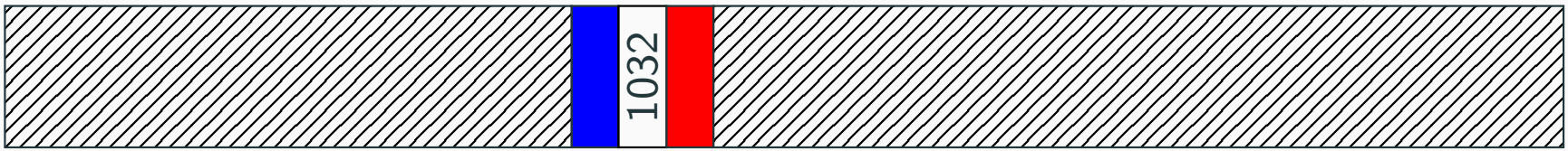
Binary search example

Imagine again, we are searching for the value 1032 in a sequence.



Binary search example

Imagine again, we are searching for the value 1032 in a sequence.



Implementing binary search

In our example of binary search we could observe that in fact three values could possibly change each step of the search. Those values are:

Implementing binary search

In our example of binary search we could observe that in fact three values could possibly change each step of the search. Those values are:

- The first index in of the slice of the `tuple` that is still relevant

Implementing binary search

In our example of binary search we could observe that in fact three values could possibly change each step of the search. Those values are:

- The first index in of the slice of the `tuple` that is still relevant
- The last index of the slice of the `tuple` that is still relevant

Implementing binary search

In our example of binary search we could observe that in fact three values could possibly change each step of the search. Those values are:

- The first index in of the slice of the `tuple` that is still relevant
- The last index of the slice of the `tuple` that is still relevant
- The next index we'd like to examine at ($\lfloor \frac{end - start}{2} \rfloor$)

Implementing binary search

In our example of binary search we could observe that in fact three values could possibly change each step of the search. Those values are:

- The first index in of the slice of the `tuple` that is still relevant
- The last index of the slice of the `tuple` that is still relevant
- The next index we'd like to examine at ($\lfloor \frac{end - start}{2} \rfloor$)

Then the algorithm we need to implement simply is the one that updates these values accordingly.

Binary search on a `tuple`

```
def bsContains(v, t):
```

Binary search on a `tuple`

```
def bsContains(v, t):
```

As with any efficient function working recursively on a `tuple` we will write a helper function to keep track of indices. In this case, it will keep track of our *low* and *high* indices.

Binary search on a `tuple`

```
def bsContains(v, t):
  def bsHelper(low, hi):
```

As with any efficient function working recursively on a `tuple` we will write a helper function to keep track of indices. In this case, it will keep track of our *low* and *high* indices.

Binary search on a `tuple`

```
def bsContains(v, t):
  def bsHelper(low, hi):
```

In writing our helper function we should first determine our base case, which is what?

Binary search on a `tuple`

```
def bsContains(v, t):
  def bsHelper(low, hi):
```

In writing our helper function we should first determine our base case, which is what?

If the values in our `tuple` that are still valid are within the range [*low*, *hi*] then when this range is empty is when we are finished!

Binary search on a `tuple`

```
def bsContains(v, t):
  def bsHelper(low, hi):
    if low > hi:
      return False
```

In writing our helper function we should first determine our base case, which is what?

If the values in our `tuple` that are still valid are within the range [*low*, *hi*] then when this range is empty is when we are finished!

Binary search on a `tuple`

```
def bsContains(v, t):
  def bsHelper(low, hi):
    if low > hi:
      return False
```

Next we should determine the index to look at. The middle value of a range can be chosen easily with the expression of $low + \lfloor \frac{hi - low}{2} \rfloor$. We can of course achieve the same behaviour with integer division.

Binary search on a `tuple`

```
def bsContains(v, t):
  def bsHelper(low, hi):
    if low > hi:
      return False
    mid = low + (hi-low)//2
```

Next we should determine the index to look at. The middle value of a range can be chosen easily with the expression of $low + \lfloor \frac{hi-low}{2} \rfloor$. We can of course achieve the same behaviour with integer division.

Binary search on a `tuple`

```
def bsContains(v, t):
  def bsHelper(low, hi):
    if low > hi:
      return False
    mid = low + (hi-low)//2
```

Next we must determine if the value at this index is the same as, less than, or more than our target value `v`.

Binary search on a `tuple`

```
def bsContains(v, t):
  def bsHelper(low, hi):
    if low > hi:
      return False
    mid = low + (hi-low)//2
    if t[mid] == v:
      ...
    if t[mid] > v:
      ...
    if t[mid] < v:
      ...
```

Next we must determine if the value at this index is the same as, less than, or more than our target value `v`.

Binary search on a `tuple`

```
def bsContains(v, t):
  def bsHelper(low, hi):
    if low > hi:
      return False
    mid = low + (hi-low)//2
    if t[mid] == v:
      ...
    if t[mid] > v:
      ...
    if t[mid] < v:
      ...
```

If the element is exactly the value we're looking for then we've found it, and have our final answer.

Binary search on a `tuple`

```
def bsContains(v, t):
  def bsHelper(low, hi):
    if low > hi:
      return False
    mid = low + (hi-low)//2
    if t[mid] == v:
      return True
    if t[mid] > v:
      ...
    if t[mid] < v:
      ...
```

If the element is exactly the value we're looking for then we've found it, and have our final answer.

Binary search on a `tuple`

```
def bsContains(v, t):
  def bsHelper(low, hi):
    if low > hi:
      return False
    mid = low + (hi-low)//2
    if t[mid] == v:
      return True
    if t[mid] > v:
      ...
    if t[mid] < v:
      ...
```

Now, if the element is larger than the one we're looking for then so is every value with an index larger than `mid`. As such, we update *hi* to be $mid - 1$

Binary search on a `tuple`

```
def bsContains(v, t):
  def bsHelper(low, hi):
    if low > hi:
      return False
    mid = low + (hi-low)//2
    if t[mid] == v:
      return True
    if t[mid] > v:
      return bsHelper(low, mid-1)
    if t[mid] < v:
      ...
```

Now, if the element is larger than the one we're looking for then so is every value with an index larger than `mid`. As such, we update *hi* to be $mid - 1$

Binary search on a `tuple`

```
def bsContains(v, t):
  def bsHelper(low, hi):
    if low > hi:
      return False
    mid = low + (hi-low)//2
    if t[mid] == v:
      return True
    if t[mid] > v:
      return bsHelper(low, mid-1)
    if t[mid] < v:
      ...
```

Finally, if the element is smaller than the one we're looking for then so is every value with an index smaller than `mid`. As such, we update *low* to be $mid + 1$.

Binary search on a `tuple`

```
def bsContains(v, t):
  def bsHelper(low, hi):
    if low > hi:
      return False
    mid = low + (hi-low)//2
    if t[mid] == v:
      return True
    if t[mid] > v:
      return bsHelper(low, mid-1)
    if t[mid] < v:
      return bsHelper(mid+1, hi)
```

Finally, if the element is smaller than the one we're looking for then so is every value with an index smaller than `mid`. As such, we update *low* to be $mid + 1$.

Binary search on a `tuple`

```
def bsContains(v, t):
  def bsHelper(low, hi):
    if low > hi:
      return False
    mid = low + (hi-low)//2
    if t[mid] == v:
      return True
    if t[mid] > v:
      return bsHelper(low, mid-1)
    if t[mid] < v:
      return bsHelper(mid+1, hi)
```

Now `bsHelper` is complete. What initial values should `bsContains` call `bsHelper` with?

Binary search on a `tuple`

```
def bsContains(v, t):
  def bsHelper(low, hi):
    if low > hi:
      return False
    mid = low + (hi-low)//2
    if t[mid] == v:
      return True
    if t[mid] > v:
      return bsHelper(low, mid-1)
    if t[mid] < v:
      return bsHelper(mid+1, hi)
  return bsHelper(0, len(t)-1)
```

In the initial call we want to consider that the value might be anywhere within the entire `tuple`! So we choose 0 for low and `len(t)-1` as hi.

Time complexity of binary search

What is the time complexity of binary search? We will present an intuitive informal proof.

Time complexity of binary search

What is the time complexity of binary search? We will present an intuitive informal proof.

Each individual application of `bsHelper` does a constant amount other than the one recursive call it makes. As such, we simply need to find how many recursive calls are made before the base case is reached.

Time complexity of binary search

What is the time complexity of binary search? We will present an intuitive informal proof.

Each individual application of `bsHelper` does a constant amount other than the one recursive call it makes. As such, we simply need to find how many recursive calls are made before the base case is reached.

In each recursive application `bsHelper` halves the number of elements in our `tuple` that *may* contain `v`. How many times can we divide a number by two before we reach zero?

Time complexity of binary search

What is the time complexity of binary search? We will present an intuitive informal proof.

Each individual application of `bsHelper` does a constant amount other than the one recursive call it makes. As such, we simply need to find how many recursive calls are made before the base case is reached.

In each recursive application `bsHelper` halves the number of elements in our `tuple` that *may* contain `v`. How many times can we divide a number by two before we reach zero?

$$log_2 n$$

Practice problem part 1

Practice problem: Write the function `findPairs` that has two parameters, the first being an arbitrary `tuple` of integers and the second being a target integer. Your function should return a `LList` of pairs where each pair is a `LList` of two integers. The pairs returned by your function should be all of the pairs of numbers in the given `tuple` that sum to `target`

For example, `findPairs((7, 3, 6, -2, 2), 5)` would return the `LList` `LL(LL(7, -2), LL(3, 2))`.

Once your function is complete determine its worst case time complexity.

Practice problem part 2

Practice problem: Write the function `findPairsSorted` that is the exact same as `findPairs` except that it assumes that the elements of its `tuple` parameter is sorted in non-decreasing order.

Try to come up with a solution that has a better worst case time complexity than your original `findPairs`, as this is possible given the new assumption.