Introduction to Efficiency

Rob Hackman

Fall 2025

University of Alberta

Table of Contents

The problem

Earlier we wrote the function `descendList`, which took an integer parameter `n` and built a `LList` of the values n to 0 in descending order.

```
def descendList(n):
  if n == 0:
    return cons(0, empty())
  return cons(n, descendList(n - 1))
```

What if we wanted the opposite, a `LList` of the values from 0 to n in ascending order?

Ascending order in one function — impossible!

Writing the function `ascendList` using only the tools we have and without writing an additional helper function is impossible! That is, you cannot construct the function

```
def ascendList(n):
    ...
```

with only using `empty` and `cons` and our other built-in `LList` functions. You would *need* to write a helper function.

One way to write `ascendList`

Let's try and write `ascendList` the way we've been typically writing recursive functions.

One way to write `ascendList`

Let's try and write `ascendList` the way we've been typically writing recursive functions.

```
def ascendList(n):
```

1) Choose our base case, when n is 0 we can easily build the `LList` That contains the elements from 0 to n in ascending order.

One way to write `ascendList`

Let's try and write `ascendList` the way we've been typically writing recursive functions.

```
def ascendList(n):
  if n == 0:
    return cons(0, empty())
```

2) Determine our method for stepping one closer to the base case, decrement *n* will progress us one step closer to the base case so we use that.

One way to write `ascendList`

Let's try and write `ascendList` the way we've been typically writing recursive functions.

```
def ascendList(n):
  if n == 0:
    return cons(0, empty())
  ror = ascendList(n-1)
```

3) Now, things get tricky. We operate by assuming our recursion works, and thus gives us the `LList` $(0, 1, ..., n-1)$. How can we build the list $(0, 1, ..., n-1, n)$ from that list?

One way to write `ascendList`

Let's try and write `ascendList` the way we've been typically writing recursive functions.

```
def ascendList(n):
  if n == 0:
    return cons(0, empty())
  ror = ascendList(n-1)
```

3) Now, things get tricky. We operate by assuming our recursion works, and thus gives us the `LList` $(0, 1, ..., n-1)$. How can we build the list $(0, 1, ..., n-1, n)$ from that list?

We can't! Not without a helper function. Since we have no way to *append* an item to a `LList`, only prepend with `cons`. So, let's write `append`.

One way to write `ascendList`

Let's try and write `ascendList` the way we've been typically writing recursive functions.

```
def append(elem, l):
```

a) Start writing append the same way we do any recursive function, identify our base case.

Observation: If we are appending an element to the empty list then appending is the same as prepending, so our work is trivial.

One way to write `ascendList`

Let's try and write `ascendList` the way we've been typically writing recursive functions.

```
def append(elem, l):
  if isEmpty(l):
    return cons(elem, empty())
```

b) How to step closer to the base case in our recursive case? If the base case is appending to an empty list, then we must make our `LList` smaller. Appending to the *rest* of the list seems like a natural choice.

One way to write `ascendList`

Let's try and write `ascendList` the way we've been typically writing recursive functions.

```
def append(elem, l):
  if isEmpty(l):
    return cons(elem, empty())
  ror = append(elem, rest(l))
```

c) Now assuming the recursive result is calculated correctly, and is the result of appending `elem` to the *rest* of our list, how do we build the final result?

Well if the recursion appends `elem` to the rest of our list, then the only thing that is missing is the *first* of our list. So, prepend that.

One way to write ascendList

Let's try and write ascendList the way we've been typically writing recursive functions.

```
def append(elem, l):
  if isEmpty(l):
    return cons(elem, empty())
  ror = append(elem, rest(l))
  return cons(first(l), ror)
```

d) Now that we are finished with writing append we can go back to solving ascendList that.

One way to write `ascendList`

Let's try and write `ascendList` the way we've been typically writing recursive functions.

```
def ascendList(n):
  if n == 0:
    return cons(0, empty())
  ror = ascendList(n-1)
```

4) Building our final answer from our recursive result is easy now, we simply append `n` to the end of the recursive result, which would be the ascending list of all elements 0 to $n - 1$.

One way to write `ascendList`

Let's try and write `ascendList` the way we've been typically writing recursive functions.

```
def ascendList(n):
  if n == 0:
    return cons(0, empty())
  ror = ascendList(n-1)
  return append(n, ror)
```

We've finished now... but is this a *good* solution?

One way to write `ascendList`

Let's try and write `ascendList` the way we've been typically writing recursive functions.

```
def ascendList(n):
  if n == 0:
    return cons(0, empty())
  ror = ascendList(n-1)
  return append(n, ror)
```

We've finished now... but is this a *good* solution?

To answer that we need to define what good means!

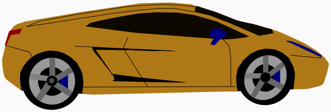
Table of Contents

A metric for “good”

What determines if one solution to a problem is better than another? What’s the best method for getting people from Edmonton to Vancouver?

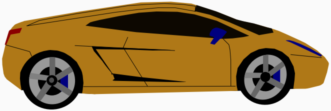
A metric for "good"

What determines if one solution to a problem is better than another? What's the best method for getting people from Edmonton to Vancouver?



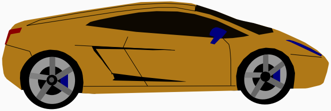
A metric for "good"

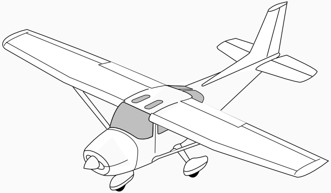
What determines if one solution to a problem is better than another? What's the best method for getting people from Edmonton to Vancouver?



A metric for “good”

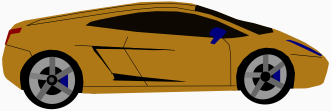
What determines if one solution to a problem is better than another? What’s the best method for getting people from Edmonton to Vancouver?

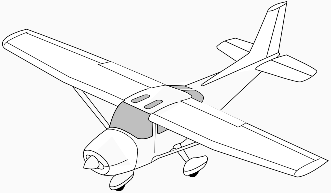


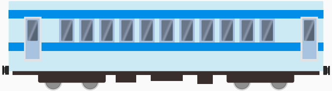


A metric for "good"

What determines if one solution to a problem is better than another? What's the best method for getting people from Edmonton to Vancouver?







Which is best?

Consider the following numbers about the four methods of travel:

- The sports car can drive two people from Edmonton to Vancouver in 10 hours
- The cargo van can drive eight people from Edmonton to Vancouver in 15 hours
- The private prop plane can fly six people from Edmonton to Vancouver in 3 hours
- The train can take one-thousand people from Edmonton to Vancouver in 18 hours

Which method of travel is best?

Which is best?

Consider the following numbers about the four methods of travel:

- The sports car can drive two people from Edmonton to Vancouver in 10 hours
- The cargo van can drive eight people from Edmonton to Vancouver in 15 hours
- The private prop plane can fly six people from Edmonton to Vancouver in 3 hours
- The train can take one-thousand people from Edmonton to Vancouver in 18 hours

Which method of travel is best?

We can't answer that question! There are different methods of comparison!

Comparing efficiency

If we rephrase the question from “which method of travel is best” to “which method of travel will get me to Vancouver the fastest” then answer is the prop plane, as that will get me there in the shortest amount of time.

Comparing efficiency

If we rephrase the question to “which method of travel moves the most people per unit of time to Vancouver” then the answer is the train!

Comparing efficiency

If we rephrase the question to "which method of travel costs the least amount of money for me to travel to Vancouver" then the answer is the cargo van!

Comparing efficiency

If we rephrase the question to "which method of travel will generate the most amount of ticket revenue for the local municipalities of Alberta and BC" then the answer is the sports car!

Efficiency of functions

The same is true for the question “which function is best?”, there are many different metrics we could use to measure.

The most common metric for the efficiency of a function is how long it takes for that function to run for a given input.

However, we do not define “how long” a function takes to run in terms of time, but rather in terms of how many “basic” operations it takes.

We must define what constitutes a basic operation, as particularly in Python it is not the case that any give expression or statement is a basic operation.

Basic operations in Python

As we learn new language features we'll discuss their runtimes. For now, some basic operations we know are:

- Our arithmetic operations[1] are all basic operations
- `empty`, `cons`, `first`, `rest`, and `isEmpty` are all basic operations
- Comparison operators on numbers are all basic operations
- Indexing a string is a basic operation
- Evaluating a statement that only includes expressions that are basic operations is a basic operation

[1] Actually, even this is not true in Python as it allows arbitrarily large numbers, but we will ignore that for now.

Examples of not basic operations

Some notable examples of expressions that we've seen that are *not* basic operations are

- String concatenation is not a basic operation
- String slicing is not a basic operation
- String repetition (multiplication) is not a basic operation
- Function calls are not basic operations, though depending on the function may effectively be![2]

[2] What this means will become clear as we learn how to evaluate functions.

Efficiency of `factorial`

Let's determine the efficiency of the `factorial` function we wrote. That is, let's determine how many basic operations are executed when `factorial` is called on some argument `n`

```
def factorial(n):
  if n == 0:
    return 1
  return n*factorial(n-1)
```

Efficiency of `factorial`

Let's determine the efficiency of the `factorial` function we wrote. That is, let's determine how many basic operations are executed when `factorial` is called on some argument `n`

```
def factorial(n):
  if n == 0:
    return 1
  return n*factorial(n-1)
```

How do we determine how many times a basic operations are executed? We count how many basic operations are executed each function call, and determine how many times the function is called recursively.

Efficiency of `factorial`

```
def factorial(n):
  if n == 0:
    return 1
  return n*factorial(n-1)
```

So how many basic operations are executed by each function call?

- One, for the comparison of `n` to 1

Efficiency of `factorial`

```
def factorial(n):
  if n == 0:
    return 1
  return n*factorial(n-1)
```

So how many basic operations are executed by each function call?

- One, for the comparison of `n` to 1
- One, for calculating `n-1`

Efficiency of `factorial`

```
def factorial(n):
  if n == 0:
    return 1
  return n*factorial(n-1)
```

So how many basic operations are executed by each function call?

- One, for the comparison of `n` to 1
- One, for calculating `n-1`
- One, for either the multiplication operation *or* `return 0`

Efficiency of `factorial`

```
def factorial(n):
  if n == 0:
    return 1
  return n*factorial(n-1)
```

So how many basic operations are executed by each function call?

- One, for the comparison of `n` to 1
- One, for calculating `n-1`
- One, for either the multiplication operation *or* `return 0`
- The recursive call to `factorial` is not basic, it will do the operations above, as well as yet another recursive call.

Efficiency of factorial

```
def factorial(n):
  if n == 0:
    return 1
  return n*factorial(n-1)
```

So how many basic operations are executed by each function call?

- One, for the comparison of `n` to 1
- One, for calculating `n-1`
- One, for either the multiplication operation *or* `return 0`
- The recursive call to `factorial` is not basic, it will do the operations above, as well as yet another recursive call.
- So, three basic operations for each call to `factorial`. How many calls occur?

Efficiency of `factorial`

```
def factorial(n):
  if n == 0:
    return 1
  return n*factorial(n-1)
```

For `factorial` called with an arbitrary $n > 0$ as an argument, there will be `n+1` function calls.

How do we know this? Since each recursive call subtracts 1 from its parameter, how many times can you subtract 1 from n before you reach 0? More formal proofs of this will show up in courses such as CMPUT 272 and CMPUT 201.

Efficiency of `factorial`

```
def factorial(n):
  if n == 0:
    return 1
  return n*factorial(n-1)
```

So, how many basic operations are executed when `factorial` is called on an arbitrary argument `n`? There are three basic operations per call, and will be a total of `n` calls, so the number of basic operations performed will be $3(n+1) = 3n + 3$.

Table of Contents

Ignoring scalars and smaller terms

As we discuss the number of operations a function executes, we are discussing for an arbitrary input size (n), and in the context of algorithm analysis we often only care about the limit as that n approaches infinity — their asymptotic growth rates.

As n trends towards infinity, scalars and more slowly growing terms do not matter, so we will ignore them.

That means when discussing our `factorial` function, we would ignore the scalar of 3 and the constant of 3 and say its *growth rate* is order n, or more simply *linear*.

Common function growth rates

Growth rate	Common name
c	constant
$\log_c n$	logarithmic
n	linear
$n \log_c n$	log-linear
n^2	quadratic
n^3	cubic
n^c	polynomial
c^n	exponential

In each of the growth rates above n refers to the size of the input, and c is some constant.

Big-O notation

When analyzing algorithms we will focus on the asymptotic growth rates of the resources[3] those functions require relative to their input size.

We now introduce Big-O notation as is commonly used in computer science. For a function f with domain a subset of the natural numbers and codomain the natural numbers then $O(f(x))$ is the infinite set of functions whose growth rate is at most the same as f.

[3] Most commonly time (measured by number of basic operations) or space (our computer's memory).

Membership of Big-O

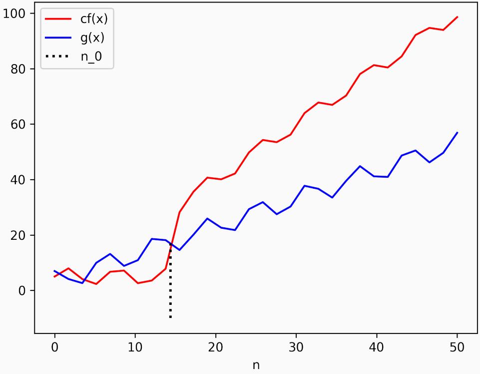
As $O(f(x))$ is the set of functions whose growth rates is no more than that of f, then the definition of membership is as such.

$g \in O(f(x)) \leftrightarrow \exists c \geq 1, n_0 \geq 1$ such that $\forall n \geq n_0, 0 \leq g(n) \leq cf(n)$

g is a member of $O(f(x))$ if and only if there exists a particular value n_0 and a constant scalar c such that $cf(n)$ is greater than or equal to $g(n)$ for all n greater than or equal to n_0.

Membership of Big-O visually

Visually, the definition of $g \in O(f(x))$ means that in the graph of $cf(x)$ and $g(x)$ there exists a point n_0 where $cf(x)$ will surpass $g(x)$ and stay above it.



Big-O not necessarily tight

If a function $g \in O(f(x))$ then that means that g grows no faster than f. This means it is entirely possible that g grows *slower* than f. For example, if $g(x) = 30 * x + 15$ then $g \in O(x^2)$.

Big-O not necessarily tight

If a function $g \in O(f(x))$ then that means that g grows no faster than f. This means it is entirely possible that g grows *slower* than f. For example, if $g(x) = 30 * x + 15$ then $g \in O(x^2)$.

For this reason, we say that big O notation denotes an upper bound, but not necessarily a *tight* upper bound.

Big-O not necessarily tight

If a function $g \in O(f(x))$ then that means that g grows no faster than f. This means it is entirely possible that g grows *slower* than f. For example, if $g(x) = 30 * x + 15$ then $g \in O(x^2)$.

For this reason, we say that big O notation denotes an upper bound, but not necessarily a *tight* upper bound.

For the cases of algorithm analysis is a non-tight bound particularly helpful? Almost all algorithms have a time complexity that is in $O(x^x)$, because most functions grow slower than tetrationally. However, if when defining the time complexity of our `factorial` function we said it was $O(x^x)$ that would not be very meaningful.

Big-O notation and growth rates

Growth rate	Common name
$O(1)$	constant
$O(\log_n)$	logarithmic
$O(n)$	linear
$O(n \log_n)$	log-linear
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(n^c)$	polynomial
$O(c^n)$	exponential

Note for constant time we simply use 1, and for logarithmic times the constant does not matter.

Big-Ω notation

While $O(f(x))$ denotes the set of functions that grow no faster than $f(x)$, $\Omega(f(x))$ denotes the set of functions that grow no *slower* than $f(x)$.

$g \in \Omega(f(x)) \leftrightarrow \exists c \geq 1, n_0 \geq 1$ such that $\forall n \geq n_0, 0 \leq cf(n) \leq g(n)$

g is a member of $\Omega(f(x))$ if and only if there exists a particular value n_0 and a constant scalar c such that $cf(n)$ is less than or equal to $g(n)$ for all n greater than or equal to n_0.

Big-Ω not necessarily tight

Once again if $g \in \Omega(f(x))$ it only guarantees that g grows no slower than $f(x)$. This also means that it's entirely possible that g grows much faster than $f(x)$. For example, if $g(x) = 5^x$ then $g \in \Omega(\log_2 x)$

Once again, of course an exponential function grows faster than a logarithmic one — this is not particularly helpful.

By your powers combined...

While alone Big-O and Big-Ω don't guarantee a tight bound, what about in conjunction?

By your powers combined...

While alone Big-O and Big-Ω don't guarantee a tight bound, what about in conjunction?

That is, what if we have a function f, and let the time complexity of a program we are trying to analyze be defined by some function g. If $g \in O(f(x))$ *and* $g \in \Omega(f(x))$ what does that tell us about g?

By your powers combined...

While alone Big-O and Big-Ω don't guarantee a tight bound, what about in conjunction?

That is, what if we have a function f, and let the time complexity of a program we are trying to analyze be defined by some function g. If $g \in O(f(x))$ *and* $g \in \Omega(f(x))$ what does that tell us about g?

It means that g grows no faster than f, but also that g grows no slower than f. This implies that the asymptotic growth rate of g *is* that of f! If g can belong to both $O(f(x))$ and $\Omega(f(x))$ then these must be tight bounds!

Big-Θ notation

$\Theta(f(x))$ notation denotes the set of functions whose asymptotic growth rate is *equal to* that of $f(x)$.

$$\begin{aligned} g \in \Theta(f(x)) \leftrightarrow & \exists c_1 \geq 1, c_2 \geq 1, n_0 \geq 1 \text{ such that} \\ & \forall n \geq n_0,\ 0 \leq c_1 f(n) \leq g(n) \leq c_2 f(n) \end{aligned}$$

Note that this really is just the logical conjunction of the definitions of g being a member of both $O(f(x))$ and $\Omega(f(x))$.

Complexity analysis

We will use these notations to talk about how much time (in terms of basic operations) our functions and programs take, as well as how much memory our programs use.

This will be one of our main ways of comparing algorithms for efficiency. We will continue to use these concepts throughout the class, and learn to understand them more deeply as we apply them.

Knowledge Check

Knowledge Check: Say for some function q you have found that it is in $O(p(x))$ by selecting the value of k for your constant scalar c and 10 for your n_0 in the Big-O definition. Additionally, you have found that q is in $\Omega(p(x))$ by selecting the value of t for your constant scalar c and 23 for your n_0 in the Big-Ω definition.

Show that q is in $\Theta(p(x))$ by selecting specific values for c_1, c_2, and n_0 in the definition for $\Theta(p(x))$.

Smaller terms and constants

Reminder: when talking about asymptotic growth rates we can simply ignore smaller terms and constant scalars.

Knowledge Check: The ignoring of smaller terms and constant scalars can seem unintuitive. For example let $f(x) = \frac{x^2}{2} - 10x - 500$ and $g(x) = 10x^2$. Prove that $g \in O(f)$, meaning that the asymptotic growth rate of g is no faster than f!

Table of Contents

Time complexity of `descendList`

Let us consider the time complexity of our `descendList` function.

```
def descendList(n):
  if n == 0:
    return cons(0, empty())
  return cons(n, descendList(n - 1))
```

Time complexity of `descendList`

Let us consider the time complexity of our `descendList` function.

```
def descendList(n):
  if n == 0:
    return cons(0, empty())
  return cons(n, descendList(n - 1))
```

- One basic operation for the comparison

Time complexity of descendList

Let us consider the time complexity of our descendList function.

```
def descendList(n):
  if n == 0:
    return cons(0, empty())
  return cons(n, descendList(n - 1))
```

- One basic operation for the comparison
- One basic operation for cons in either case

Time complexity of `descendList`

Let us consider the time complexity of our `descendList` function.

```
def descendList(n):
  if n == 0:
    return cons(0, empty())
  return cons(n, descendList(n - 1))
```

- One basic operation for the comparison
- One basic operation for `cons` in either case
- One basic operation for empty in the base case

Time complexity of `descendList`

Let us consider the time complexity of our `descendList` function.

```
def descendList(n):
  if n == 0:
    return cons(0, empty())
  return cons(n, descendList(n - 1))
```

- One basic operation for the comparison
- One basic operation for `cons` in either case
- One basic operation for empty in the base case
- One basic operation for $n - 1$ in the recursive case

Time complexity of `descendList`

Let us consider the time complexity of our `descendList` function.

```
def descendList(n):
  if n == 0:
    return cons(0, empty())
  return cons(n, descendList(n - 1))
```

- One basic operation for the comparison
- One basic operation for `cons` in either case
- One basic operation for empty in the base case
- One basic operation for $n - 1$ in the recursive case
- So, three operations for each call to the function

Time complexity of `descendList`

Let us consider the time complexity of our `descendList` function.

```
def descendList(n):
  if n == 0:
    return cons(0, empty())
  return cons(n, descendList(n - 1))
```

- One basic operation for the comparison
- One basic operation for `cons` in either case
- One basic operation for empty in the base case
- One basic operation for $n - 1$ in the recursive case
- So, three operations for each call to the function
- Once again, $n + 1$ calls to the function, n recursive calls and the one original call

Time complexity of `descendList`

Let us consider the time complexity of our `descendList` function.

```
def descendList(n):
  if n == 0:
    return cons(0, empty())
  return cons(n, descendList(n - 1))
```

The time complexity of our `descendList` function is defined by the function $3x + 3$ — the same as our `factorial` function.

While it is the time complexity of `descendList` that we are calculating, we would still typically just say $\mathit{descendList} \in \Theta(x)$.

Time complexity of `descendList`

Let us consider the time complexity of our `descendList` function.

```
def descendList(n):
  if n == 0:
    return cons(0, empty())
  return cons(n, descendList(n - 1))
```

What about `ascendList`?

Time complexity of `ascendList`

First, in order to calculate the time complexity of `ascendList` we must calculate the time complexity of `append`. Now, we are evaluating the runtime of a function whose parameter is a `LList`. In this case we will talk about our input size being n, the size of the `LList`.

```
def append(elem, l):
  if isEmpty(l):
    return cons(elem, empty())
  return cons(first(l), append(elem, rest(l)))
```

Time complexity of `ascendList`

```
def append(elem, l):
  if isEmpty(l):
    return cons(elem, empty())
  return cons(first(l), append(elem, rest(l)))
```

- One basic operation for `isEmpty`

Time complexity of `ascendList`

```
def append(elem, l):
  if isEmpty(l):
    return cons(elem, empty())
  return cons(first(l), append(elem, rest(l)))
```

- One basic operation for `isEmpty`
- One basic operation for `cons` in either case

Time complexity of `ascendList`

```
def append(elem, l):
  if isEmpty(l):
    return cons(elem, empty())
  return cons(first(l), append(elem, rest(l)))
```

- One basic operation for `isEmpty`
- One basic operation for `cons` in either case
- One basic operation for `empty` in the base case

Time complexity of `ascendList`

```
def append(elem, l):
  if isEmpty(l):
    return cons(elem, empty())
  return cons(first(l), append(elem, rest(l)))
```

- One basic operation for `isEmpty`
- One basic operation for `cons` in either case
- One basic operation for `empty` in the base case
- One basic operation for `first` in the recursive case

Time complexity of `ascendList`

```
def append(elem, l):
  if isEmpty(l):
    return cons(elem, empty())
  return cons(first(l), append(elem, rest(l)))
```

- One basic operation for `isEmpty`
- One basic operation for `cons` in either case
- One basic operation for `empty` in the base case
- One basic operation for `first` in the recursive case
- One basic operation for `rest` in the recursive case

Time complexity of `ascendList`

```
def append(elem, l):
  if isEmpty(l):
    return cons(elem, empty())
  return cons(first(l), append(elem, rest(l)))
```

- One basic operation for `isEmpty`
- One basic operation for `cons` in either case
- One basic operation for `empty` in the base case
- One basic operation for `first` in the recursive case
- One basic operation for `rest` in the recursive case
- Three basic operations for base case call, four basic operations each other call

Time complexity of `ascendList`

```
def append(elem, l):
  if isEmpty(l):
    return cons(elem, empty())
  return cons(first(l), append(elem, rest(l)))
```

- One basic operation for `isEmpty`
- One basic operation for `cons` in either case
- One basic operation for `empty` in the base case
- One basic operation for `first` in the recursive case
- One basic operation for `rest` in the recursive case
- Three basic operations for base case call, four basic operations each other call
- $n + 1$ calls to `append`, one of which is the base case call

Time complexity of `ascendList`

```
def append(elem, l):
  if isEmpty(l):
    return cons(elem, empty())
  return cons(first(l), append(elem, rest(l)))
```

- One basic operation for `isEmpty`
- One basic operation for `cons` in either case
- One basic operation for `empty` in the base case
- One basic operation for `first` in the recursive case
- One basic operation for `rest` in the recursive case
- Three basic operations for base case call, four basic operations each other call
- $n + 1$ calls to `append`, one of which is the base case call
- $4n + 3$ basic operations for `append`. So $append \in \Theta(n)$

Time complexity of `ascendList`

Now we can calculate the time complexity of `ascendList`

```
def ascendList(n):
  if n == 0:
    return cons(0, empty())
  return append(n, ascendList(n-1))
```

Time complexity of `ascendList`

Now we can calculate the time complexity of `ascendList`

```
def ascendList(n):
  if n == 0:
    return cons(0, empty())
  return append(n, ascendList(n-1))
```

- One basic operation for the comparison

Time complexity of `ascendList`

Now we can calculate the time complexity of `ascendList`

```
def ascendList(n):
  if n == 0:
    return cons(0, empty())
  return append(n, ascendList(n-1))
```

- One basic operation for the comparison
- Two basic operations in the base case for `cons` and `empty`

Time complexity of `ascendList`

Now we can calculate the time complexity of `ascendList`

```
def ascendList(n):
  if n == 0:
    return cons(0, empty())
  return append(n, ascendList(n-1))
```

- One basic operation for the comparison
- Two basic operations in the base case for `cons` and `empty`
- One basic operation in the recursive case for `n-1`

Time complexity of `ascendList`

Now we can calculate the time complexity of `ascendList`

```
def ascendList(n):
  if n == 0:
    return cons(0, empty())
  return append(n, ascendList(n-1))
```

- One basic operation for the comparison
- Two basic operations in the base case for `cons` and `empty`
- One basic operation in the recursive case for `n-1`
- One call to `append` for each recursive case

Time complexity of `ascendList`

Now we can calculate the time complexity of `ascendList`

```
def ascendList(n):
  if n == 0:
    return cons(0, empty())
  return append(n, ascendList(n-1))
```

- One basic operation for the comparison
- Two basic operations in the base case for `cons` and `empty`
- One basic operation in the recursive case for `n-1`
- One call to `append` for each recursive case
- $n+1$ calls to `descendList`, one of which is the base case call

Time complexity of `ascendList`

Now we can calculate the time complexity of `ascendList`

```
def ascendList(n):
  if n == 0:
    return cons(0, empty())
  return append(n, ascendList(n-1))
```

- One basic operation for the comparison
- Two basic operations in the base case for `cons` and `empty`
- One basic operation in the recursive case for `n-1`
- One call to `append` for each recursive case
- $n+1$ calls to `descendList`, one of which is the base case call
- $n(n+2)+3$ basic operations for `ascendList`. The n recursive case calls of `descendList` each do $n+2$ work, and the base case call does 3 work.

Time complexity of `ascendList`

Now we can calculate the time complexity of `ascendList`

```
def ascendList(n):
  if n == 0:
    return cons(0, empty())
  return append(n, ascendList(n-1))
```

So, `ascendList` has a time complexity of $n(n+2)+3 = n^2+2n+3$, which is quadratic! It seems intuitively true that we *should* be able to build a list of the values from 0 to n in linear time. So it would seem our implementation of `ascendList` is suboptimal!

Aside — thinking about `append`

Intrepid student that you are, you might ask:

> "Hold on, the calculation on the previous slide doesn't seem fair! Each call to `append` operates on a `LList` of length n, while the last `append` only operates on a `LList` of length 1! Clearly we can't ignore this and simplify it to n work for each `append` call!"

Knowledge Check: Yes we can. Prove to yourself that even if we consider the fact that each call to `append` is on a smaller and smaller `LList` that the time complexity of `ascendList` is still quadratic.

Thinking about `append`

To prove that `ascendList` would still be quadratic even if taking into account the reduced size of the `append` calls we will break down our function application further.

Thinking about `append`

To prove that `ascendList` would still be quadratic even if taking into account the reduced size of the `append` calls we will break down our function application further.

Furthermore, we will even be generous and assume that `append` has exactly a time complexity of n instead of the $4n + 3$ we calculated it to have. Despite even this further reduction we *still* won't end up with a sub-quadratic `ascendList`.

Thinking about `append`

To prove that `ascendList` would still be quadratic even if taking into account the reduced size of the `append` calls we will break down our function application further.

- The first call to `ascendList` does two basic operations, and the call to `append` on the recursive result, which will be a list of length n

Thinking about `append`

To prove that `ascendList` would still be quadratic even if taking into account the reduced size of the `append` calls we will break down our function application further.

- The first call to `ascendList` does two basic operations, and the call to `append` on the recursive result, which will be a list of length n
- The second call to `ascendList` does two basic operations, and the call to `append` on the recursive result, which will be a list of length $n - 1$

Thinking about `append`

To prove that `ascendList` would still be quadratic even if taking into account the reduced size of the `append` calls we will break down our function application further.

- The first call to `ascendList` does two basic operations, and the call to `append` on the recursive result, which will be a list of length n
- The second call to `ascendList` does two basic operations, and the call to `append` on the recursive result, which will be a list of length $n-1$
- The third call to `ascendList` does two basic operations, and the call to `append` on the recursive result, which will be a list of length $n-2$

Thinking about `append`

To prove that `ascendList` would still be quadratic even if taking into account the reduced size of the `append` calls we will break down our function application further.

- The first call to `ascendList` does two basic operations, and the call to `append` on the recursive result, which will be a list of length n
- ...
- Call $n-1$ to `ascendList` does two basic operations, and the call to `append` on the recursive result, which will be a list of length $n-(n-2)=2$

Thinking about `append`

To prove that `ascendList` would still be quadratic even if taking into account the reduced size of the `append` calls we will break down our function application further.

- The first call to `ascendList` does two basic operations, and the call to `append` on the recursive result, which will be a list of length n
- ...
- Call $n-1$ to `ascendList` does two basic operations, and the call to `append` on the recursive result, which will be a list of length $n-(n-2)=2$
- Call n to `ascendList` does two basic operations, and the call to `append` on the recursive result, which will be a list of length $n-(n-1)=1$

Thinking about `append`

So, ultimately we have $n+1$ calls to `ascendList`, the first call does $2+n$ work, the second does $2+n-1$ work, ..., the $n-1$ does $2+n-(n-2)=4$ work, the n call does $2+n-(n-1)=3$ work, and the $n+1$ call is the base case which does 2 work.

Thinking about `append`

So, ultimately we have $n + 1$ calls to `ascendList`, the first call does $2 + n$ work, the second does $2 + n - 1$ work, ..., the $n - 1$ does $2 + n - (n - 2) = 4$ work, the n call does $2 + n - (n - 1) = 3$ work, and the $n + 1$ call is the base case which does 2 work.

So the total operations performed by `ascendList` then is defined by $\sum_{i=0}^{n} 2 + i$

Thinking about `append`

So, ultimately we have $n + 1$ calls to `ascendList`, the first call does $2 + n$ work, the second does $2 + n - 1$ work, ..., the $n - 1$ does $2 + n - (n - 2) = 4$ work, the n call does $2 + n - (n - 1) = 3$ work, and the $n + 1$ call is the base case which does 2 work.

So the total operations performed by `ascendList` then is defined by $\sum_{i=0}^{n} 2 + i$

$$\sum_{i=0}^{n} 2 + i = 2n + \sum_{i=0}^{n} i$$

Thinking about `append`

So, ultimately we have $n+1$ calls to `ascendList`, the first call does $2+n$ work, the second does $2+n-1$ work, ..., the $n-1$ does $2+n-(n-2)=4$ work, the n call does $2+n-(n-1)=3$ work, and the $n+1$ call is the base case which does 2 work.

So the total operations performed by `ascendList` then is defined by $\sum_{i=0}^{n} 2+i$

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

Thinking about `append`

So, ultimately we have $n+1$ calls to `ascendList`, the first call does $2+n$ work, the second does $2+n-1$ work, ..., the $n-1$ does $2+n-(n-2)=4$ work, the n call does $2+n-(n-1)=3$ work, and the $n+1$ call is the base case which does 2 work.

So the total operations performed by `ascendList` then is defined by $\sum_{i=0}^{n} 2+i$

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

Thinking about `append`

So, ultimately we have $n+1$ calls to `ascendList`, the first call does $2+n$ work, the second does $2+n-1$ work, ..., the $n-1$ does $2+n-(n-2)=4$ work, the n call does $2+n-(n-1)=3$ work, and the $n+1$ call is the base case which does 2 work.

So the total operations performed by `ascendList` then is defined by $\sum_{i=0}^{n} 2+i$

$$\begin{aligned}\sum_{i=0}^{n} 2+i &= 2n+\sum_{i=0}^{n} i \\ &= 2n+\frac{n(n+1)}{2} \\ &= 2n+\frac{n^2}{2}+\frac{n}{2} \\ &= \frac{n^2}{2}+\frac{5n}{2}\end{aligned}$$

Thinking about `append`

So, ultimately we have $n + 1$ calls to `ascendList`, the first call does $2 + n$ work, the second does $2 + n - 1$ work, ..., the $n - 1$ does $2 + n - (n - 2) = 4$ work, the n call does $2 + n - (n - 1) = 3$ work, and the $n + 1$ call is the base case which does 2 work.

So the total operations performed by `ascendList` then is defined by $\sum_{i=0}^{n} 2 + i$

$$\begin{aligned}\sum_{i=0}^{n} 2 + i &= 2n + \sum_{i=0}^{n} i \\ &= 2n + \frac{n(n+1)}{2} \\ &= 2n + \frac{n^2}{2} + \frac{n}{2} \\ &= \frac{n^2}{2} + \frac{5n}{2}\end{aligned}$$

Which is still quadratic!

Improving `ascendList`

Now that we have identified our implementation of `ascendList` has a time complexity in $\Theta(n^2)$, and we believe it is possible to complete the task in linear time, what should we do?

Improving `ascendList`

Now that we have identified our implementation of `ascendList` has a time complexity in $\Theta(n^2)$, and we believe it is possible to complete the task in linear time, what should we do?

Consider the `descendList` function, and how it was able to easily achieve a linear runtime.

Improving `ascendList`

Now that we have identified our implementation of `ascendList` has a time complexity in $\Theta(n^2)$, and we believe it is possible to complete the task in linear time, what should we do?

Consider the `descendList` function, and how it was able to easily achieve a linear runtime.

`descendList` was simple because we were always prepending with `cons` rather than appending. This was doable because our parameter always told us the value to prepend, while the opposite is true with `ascendList`.

Measuring efficiency of functions

So, to measure the efficiency of a function we count how many basic operations it performs. How many basic operations does the following function execute?

```
def length(l):
  if isEmpty(l):
    return 0
  return 1 + length(rest(l))
```

Measuring efficiency of functions

So, to measure the efficiency of a function we count how many basic operations it performs. How many basic operations does the following function execute?

```
def length(l):
  if isEmpty(l):
    return 0
  return 1 + length(rest(l))
```

It depends on the argument `l`!

So, to measure the efficiency of a function we count how many basic operations it performs. How many basic operations does the following function execute?

```
def length(l):
  if isEmpty(l):
    return 0
  return 1 + length(rest(l))
```

It depends on the argument `l`!

When measuring the number of basic operations a function executes we must specify how many operations that function takes for a given size of input. We will use n as a free variable meant to represent the size of the functions input.

Measuring efficiency of functions

```
def length(l):
  if isEmpty(l):
    return 0
  return 1 + length(rest(l))
```

So how many basic operations does the above function execute?

Measuring efficiency of functions

```
def length(l):
  if isEmpty(l):
    return 0
  return 1 + length(rest(l))
```

So how many basic operations does the above function execute?

- The function must check `if isEmpty(l)` which is one basic operation.

Measuring efficiency of functions

```
def length(l):
  if isEmpty(l):
    return 0
  return 1 + length(rest(l))
```

So how many basic operations does the above function execute?

- The function must check `if isEmpty(l)` which is one basic operation.
- If the above is true then the statement `return 0` executes, which is one more basic operation.

Measuring efficiency of functions

```
def length(l):
  if isEmpty(l):
    return 0
  return 1 + length(rest(l))
```

So how many basic operations does the above function execute?

- The function must check `if isEmpty(l)` which is one basic operation.
- If the above is true then the statement `return 0` executes, which is one more basic operation.
- If the above is false then the statement `return 1 + length(rest(l))` executes. The arithmetic and `rest` are basic operations, but recursively calling `length` is not.

Measuring efficiency of functions

```
def length(l):
  if isEmpty(l):
    return 0
  return 1 + length(rest(l))
```

So how many basic operations does the above function execute?

- The function must check `if isEmpty(l)` which is one basic operation.
- If the above is true then the statement `return 0` executes, which is one more basic operation.
- If the above is false then the statement `return 1 + length(rest(l))` executes. The arithmetic and `rest` are basic operations, but recursively calling `length` is not.
- So each call to `length` executes about two basic operations, but how many times does `length` get called?

Measuring efficiency of functions

```
def length(l):
  if isEmpty(l):
    return 0
  return 1 + length(rest(l))
```

The function `length` calls itself once for each item in the argument `LList`, because that is the number of times we can call `rest` on a `LList` before we reach the empty list.

That means that if the length of our parameter `l` is n then the number of basic operations performed by our `length` function is $2n$.