CPSC 221

Basic Algorithms and Data Structures

May 15, 2015

Administrative stuff

Labs so far seemed to go ok. Hiccups with instructions on how to make things work on your own computer.

Even though no class this coming Monday, labs will resume on Tuesday.

So far

We’ve been looking at abstract data types. We’ve seen arrays (and the arraylist), linked lists, and stacks.

We’ll add two more ADTs to our repertoire today, and then we’ll begin to formalize the analysis we’ve been doing informally this week.

Queue

The queue is yet another container for a sequential collection of data. Like the stack, it’s great for storing “postponed obligations” or “postponed computations”.

Unlike the stack, the queue ADT permits data to be added only at one end of the sequence and removed from the other end.

Real world analogies:

> The queue for a bus, or tickets at the movies,
> or Timmies, or ...
> The waitlist to enrol in a course

Queue

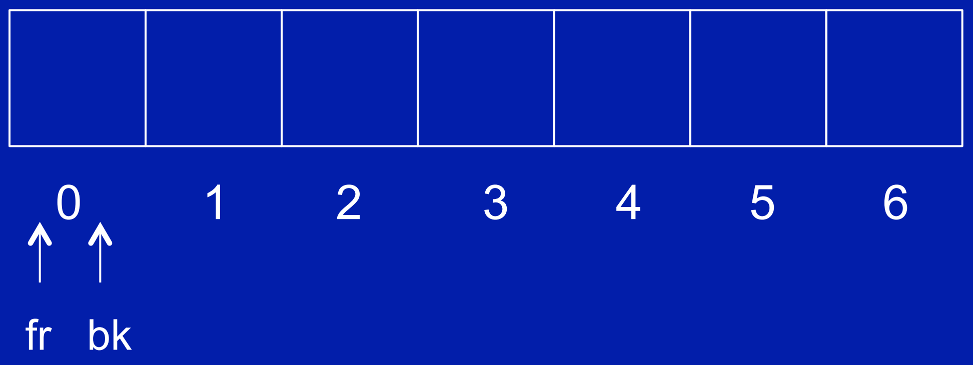
Queue operations:

enqueue(item)	-	add to back of queue
dequeue()	-	remove from front of queue
front()	-	return item at front without removing it
empty()	-	return true if queue empty else false
size()	-	return number of items in stack

Queue implementation

You could implement a queue in a couple of different ways. What might they be?

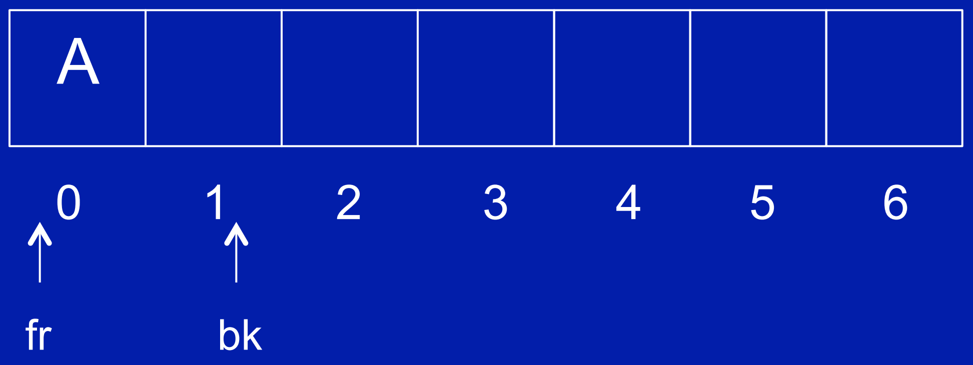
Queue implementation



You could implement a queue in a couple of different ways. What might they be?

Just like the stack, we could use an array or a linked list for implementation. Here's an example of some serious queue action.

Queue implementation



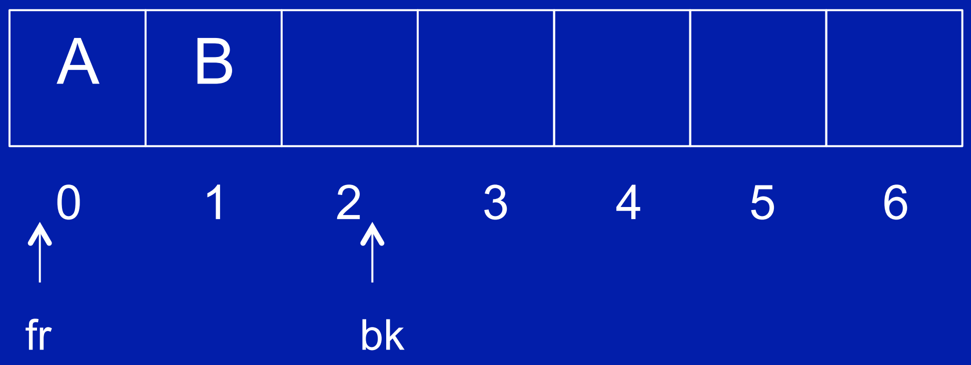
A						
0	1	2	3	4	5	6

fr

bk

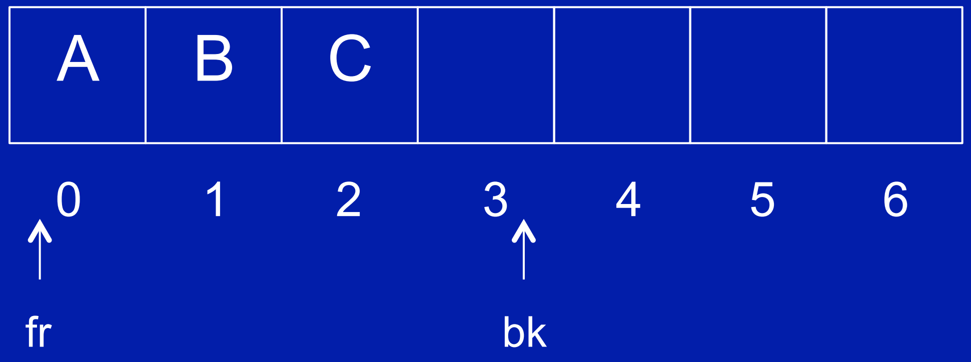
enqueue(A)

Queue implementation



enqueue(A)
enqueue(B)

Queue implementation



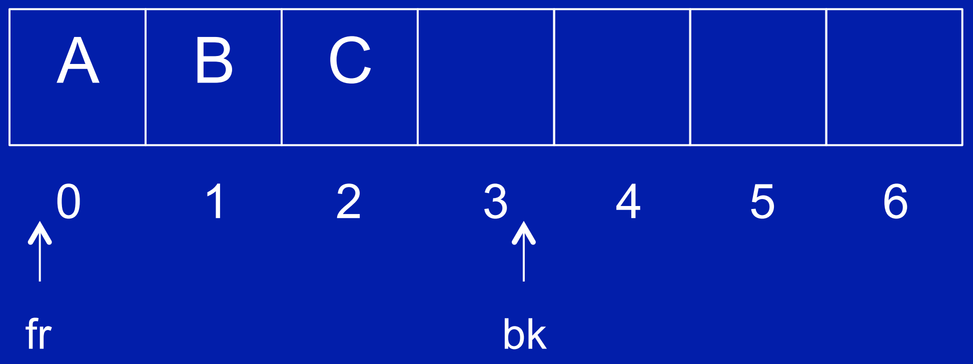
A	B	C				
0	1	2	3	4	5	6

fr

bk

enqueue(A)
enqueue(B)
enqueue(C)

Queue implementation



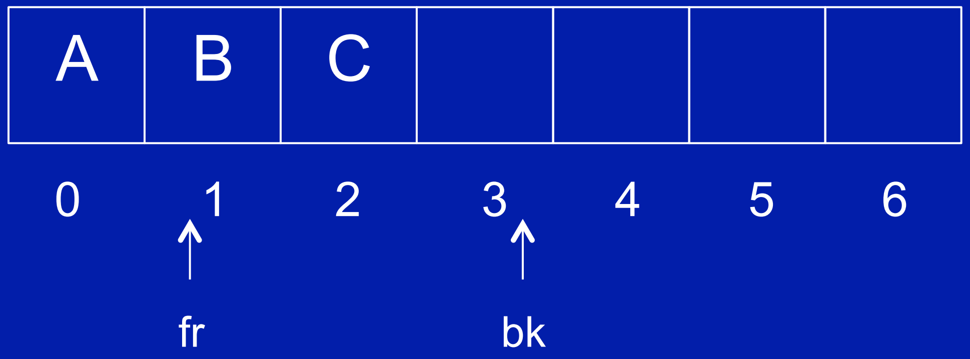
A	B	C				
0	1	2	3	4	5	6

fr

bk

enqueue(A)
enqueue(B)
enqueue(C)
front()
A

Queue implementation



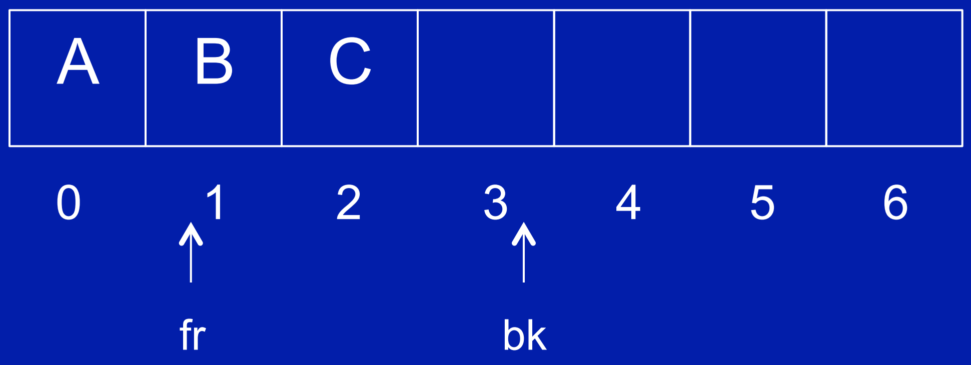
A	B	C				
0	1	2	3	4	5	6

fr

bk

enqueue(A)
enqueue(B)
enqueue(C)
front()
   A
dequeue()

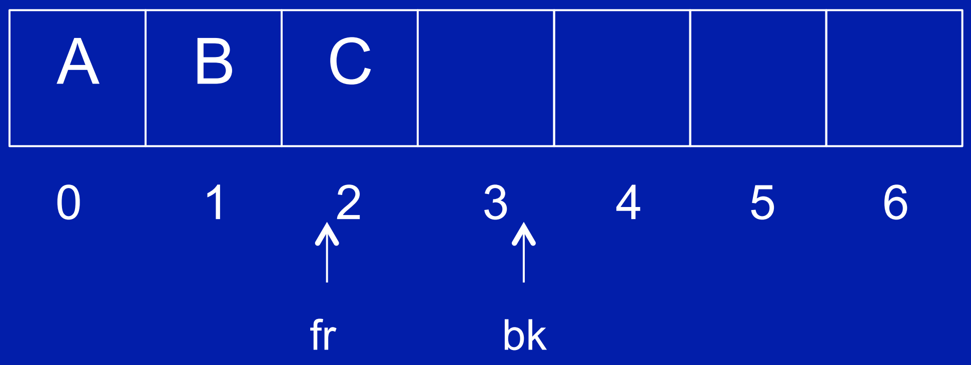
Queue implementation



A	B	C				
0	1	2	3	4	5	6

enqueue(A)
enqueue(B)
enqueue(C)
front()
  A
dequeue()
front()
  B

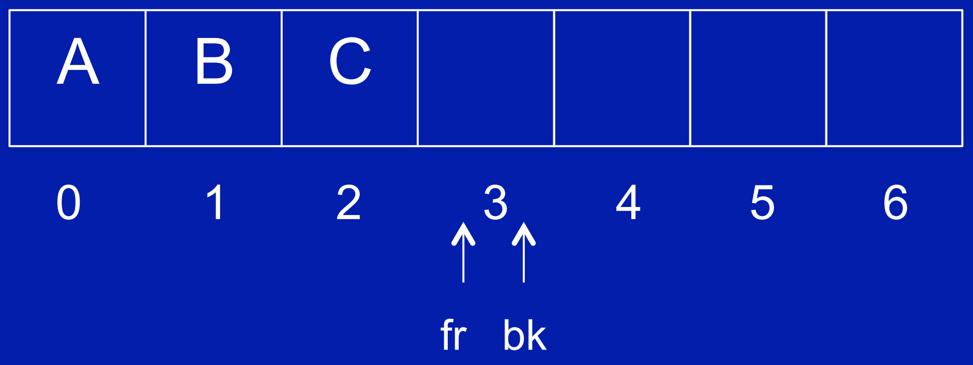
Queue implementation



A	B	C				
0	1	2	3	4	5	6

enqueue(A)
enqueue(B)
enqueue(C)
front()
  A
dequeue()
front()
  B
dequeue()

Queue implementation

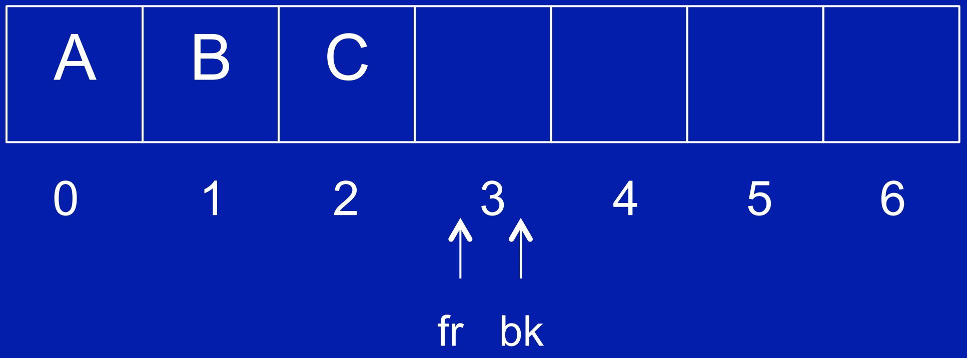


A	B	C				
0	1	2	3	4	5	6

fr bk

enqueue(A)
enqueue(B)
enqueue(C)
front()
   A
dequeue()
front()
   B
dequeue()
dequeue()

Queue implementation

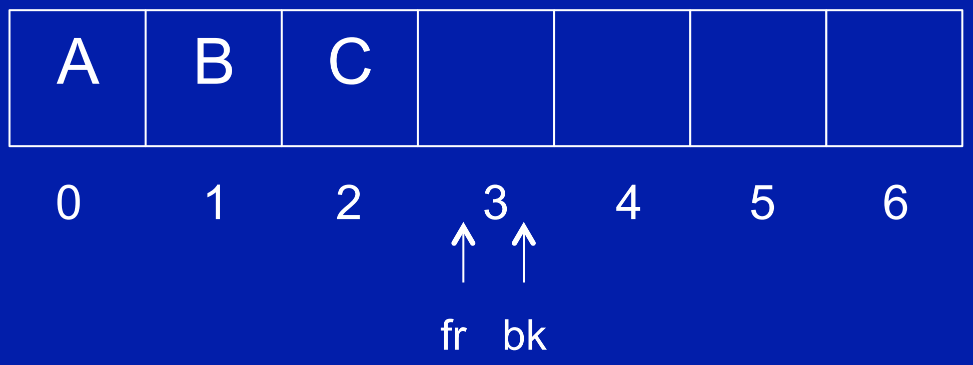


A	B	C				
0	1	2	3	4	5	6

fr bk

Now the queue is empty, but it only has about half the available space that it did when we started. How can we deal with that problem?

Queue implementation



A B C

0 1 2 3 4 5 6

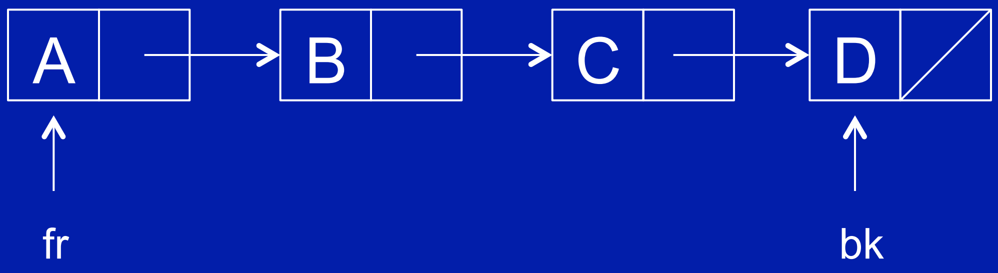
fr bk

One approach would be to implement a circular array. When the back pointer is incremented to point beyond the array, reset the back pointer to 0. Same with the front pointer.

```
enqueue(x)
{
  array[bk] = x
  bk = (bk + 1) % size
}
```

```
dequeue()
{
  fr = (fr + 1) % size
}
```

Queue implementation



A queue can also be implemented as a linked list.

```
enqueue(x)
{
  if (empty())
     fr = bk = new Node(x)
  else
     bk->next = new Node(x)
     bk = bk->next
}
```

```
dequeue()
{
  if (not(empty()))
     temp = fr
     fr = fr->next
     delete temp
}
```

These bits of pseudocode should give you an idea of how to write similar algorithms for the stack ADT from Wednesday. A home exercise for you.

Deque

One more linear sequential ADT.

A deque (pronounced “deck”) is a double-ended queue.

It acts like a queue, but we can add at either end and remove at either end.

A deque can also be implemented using either an array or a linked list.

It is an abstraction of the queue and stack.

And now you know.

Things you might have read

If you read your Koffman and Wolfgang textbook, you saw things like “a removal from the front [of an array-based queue] will be an O(n) process if...” or “stack operations using a linked data structure would be O(1).”

What does O(n) mean? O(1)?

We’re about to answer that. First, let’s tackle a related question...

Comparing algorithms

Why learn about comparing algorithms?

- You want to make intelligent choices. A poor choice may prevent the software you develop from completing its task in reasonable time or in reasonable space

What do you judge them on?

- Time (how long to run?)
- Space (how much memory does it use?)
- Other attributes
  - Expensive operations (e.g. I/O)
  - Elegance, cleverness
  - Energy, power (really? ask Google)
  - Ease of programming

Analyzing runtime

Iterative Fibonacci:

```
old2 = 1
old1 = 1
for (i = 3; i <= n; i++)
{
   result = old2 + old1
   old1 = old2
   old2 = result
}
```

How long does this take?
A second? A minute?

And what is this Fibonacci you’re talking about?

Analyzing runtime

Leonardo of Pisa, or Leonardo Pisano, or Leonardo Bonacci (1170 - 1250), also known as Fibonacci, came up with a model of growth in an idealized bunny (really) population.

Assuming that

- in the first month there is just one newly-born pair
- new-born pairs become fertile from their second month
- each month every fertile pair spawns a new pair, and
- the bunnies never die

Then if we have A pairs of fertile and newly-born bunnies in month N and we have B pairs in month N+1, then in month N+2 we'll have A+B pairs.

Analyzing runtime

The numbers are 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 and so on.

Fibonacci was wrong about the growth of bunny populations, but his numbers live on in mathematical history.

Here's how to compute the Fibonacci numbers:

```
fib(0) = 0
fib(1) = 1
fib(n) = fib(n - 1) + fib(n - 2)
```

This recursive version is horrifyingly slow (code it up in Racket if you want to see what “horrifyingly slow” means), so for efficiency we use an iterative version

Analyzing runtime

Iterative Fibonacci:

```
old2 = 1
old1 = 1
for (i = 3; i <= n; i++)
{
   result = old2 + old1
   old1 = old2
   old2 = result
}
```

How long does this take?
A second? A minute?

It depends on `n`

So we'll approximate the runtime as a function of `n`.

More generally, that function will be a function of the input.

Analyzing runtime

Iterative Fibonacci:

```
old2 = 1
old1 = 1
for (i = 3; i <= n; i++)
{
   result = old2 + old1
   old1 = old2
   old2 = result
}
```

There are lots of factors we could consider:

- What machine is it running on?
- What language is it written in?
- What compiler was used?
- How was it programmed?

But we want a basis for comparison that's independent of these implementation details.

Consequently, we want to count just “basic operations” like arithmetic, and memory access, while ignoring the details given above.

Analyzing runtime

Iterative Fibonacci:

```
old2 = 1
old1 = 1
for (i = 3; i <= n; i++)
{
   result = old2 + old1
   old1 = old2
   old2 = result
}
```

How many operations when this algorithm is run?

Analyzing runtime

Iterative Fibonacci:

```
old2 = 1
old1 = 1
for (i = 3; i <= n; i++)
{
   result = old2 + old1
   old1 = old2
   old2 = result
}
```

How many operations when this algorithm is run?

3 going into the loop

Analyzing runtime

Iterative Fibonacci:

```
old2 = 1
old1 = 1
for (i = 3; i <= n; i++)
{
   result = old2 + old1
   old1 = old2
   old2 = result
}
```

How many operations when this algorithm is run?

3 going into the loop

6 each time the loop is executed

Analyzing runtime

Iterative Fibonacci:

```
old2 = 1
old1 = 1
for (i = 3; i <= n; i++)
{
   result = old2 + old1
   old1 = old2
   old2 = result
}
```

How many operations when this algorithm is run?

3 going into the loop

6 each time the loop is executed

the loop is executed n - 2 times

Analyzing runtime

Iterative Fibonacci:

```
old2 = 1
old1 = 1
for (i = 3; i <= n; i++)
{
   result = old2 + old1
   old1 = old2
   old2 = result
}
```

How many operations when this algorithm is run?

3 going into the loop

6 each time the loop is executed

the loop is executed n - 2 times

1 going out of the loop

Analyzing runtime

Iterative Fibonacci:

```
old2 = 1
old1 = 1
for (i = 3; i <= n; i++)
{
   result = old2 + old1
   old1 = old2
   old2 = result
}
```

How many operations when this algorithm is run?

3 going into the loop

6 each time the loop is executed

the loop is executed n - 2 times

1 going out of the loop

3 + (6)(n-2) + 1

Analyzing runtime

Iterative Fibonacci:

```
old2 = 1
old1 = 1
for (i = 3; i <= n; i++)
{
   result = old2 + old1
   old1 = old2
   old2 = result
}
```

How many operations when this algorithm is run?

3 going into the loop

6 each time the loop is executed

the loop is executed n - 2 times

1 going out of the loop

3 + (6)(n-2) + 1

If we're ignoring details, does it make sense to be so precise?

Analyzing runtime

Iterative Fibonacci:

```
old2 = 1
old1 = 1
for (i = 3; i <= n; i++)
{
   result = old2 + old1
   old1 = old2
   old2 = result
}
```

How many operations when this algorithm is run?

3 going into the loop

6 each time the loop is executed

the loop is executed n - 2 times

1 going out of the loop

3 + (6)(n-2) + 1

If we're ignoring details, does it make sense to be so precise? It's educational now, but later we'll see that we can do this more simply and ignore the details

Run time as a function of input

The run time of iterative Fibonacci is (depending on the details of how we count and our implementation):

$3 + (6)(n - 2) + 1$ which simplifies to $6n - 8$

Since we've abstracted away exactly how long the different operations take, and on what computer we're running, does it make sense to say $6n - 8$ instead of $6n - 10$ or $5n - 8$ or ...?

What matters here is n. As n gets bigger, the run time grows linearly in proportion to n. (We'll formalize this in the very near future.) The 6 and the 8 are just details.

Run time as a function of input

What if there are many possible inputs, as in the case of linear search in an array? Does that change anything?

```
boolean find(char target_char, char[] array)
{
  int n = array.length;
  for (i = 0; i < n; i++)
  {
     if array[i] == target_char
        return true;
  }
  return false;
}
```

What if the item is the first in the list? The last in the list? Not in the list? Which run time should we report?

Which run time?

There are different kinds of analysis we could report:

- Best Case
- Worst Case
- Average Case (Expected Time)
- Common Case
- Amortized
- and so on...

Which run time?

There are different kinds of analysis we could report:

- Best Case
- Worst Case
- Average Case (Expected Time)
- Common Case
- Amortized
- and so on...

Best Case isn’t all that useful, because it doesn’t happen often

Which run time?

There are different kinds of analysis we could report:

- Best Case
- Worst Case
- Average Case (Expected Time)
- Common Case
- Amortized
- and so on...

Worst Case is useful but pessimistic

Which run time?

There are different kinds of analysis we could report:

- Best Case
- Worst Case
- Average Case (Expected Time)
- Common Case
- Amortized
- and so on...

Average Case is useful but can be hard to do right

Which run time?

There are different kinds of analysis we could report:

- Best Case
- Worst Case
- Average Case (Expected Time)
- Common Case
- Amortized
- and so on...

Common Case is useful but poorly defined

Which run time?

There are different kinds of analysis we could report:

- Best Case
- Worst Case
- Average Case (Expected Time)
- Common Case
- Amortized
- and so on...

Amortized is useful and you'll see this in more advanced courses

Big-O notation

Whatever we decide to report, the common means of reporting it in the world of algorithm analysis is through Big-O (pronounced “big oh”) notation.

Going back to the iterative Fibonacci example, we calculated the function for the run time to be 6n - 8. Then we argued that the 6 and the 8 are noise for our purposes.

One way to say all that is “The running time for the Fibonacci algorithm finding the nth Fibonacci number is on the order of n.”

In Big-O notation, that’s just *T(n) is O(n)*

Big-O notation

T(n) is O(n)

T(n) is our shorthand for the runtime of the function being analyzed.

The O in O(n) means “order of magnitude”, so Big-O notation is clearly, and intentionally, not precise. It’s a formal notation for the approximation of the time (or space) requirements of running algorithms.

In the current context, we might say “the run time of iterative Fibonacci is Oh of n” or just “iterative Fibonacci is Oh of n”.

Big-O arithmetic

There's a formal mathematical definition for Big-O that we're obliged to discuss:

T(n) is O(f(n)) if there are two positive constants, n_0 and c, and a function f(n) such that cf(n) >= T(n) for all n > n_0

Does your head hurt yet?

Or, as your book puts it, as n gets sufficiently large (larger than n_0), there is some constant c for which the processing time will always be less than or equal to cf(n), so cf(n) is an upper bound on the performance. The performance will never be worse than cf(n) and may be better.

Feeling better now? I didn't think so.

Big-O arithmetic

One more time. Here’s what this

T(n) is O(f(n)) if there are two positive constants, n_0 and c, and a function f(n) such that cf(n) >= T(n) for all n > n_0

really means in a practical sense:

If you want to show that T(n) is O(f(n)) then find two positive constants, n_0 and c, and a function f(n) that satisfy the constraints above. For example...

Big-O arithmetic

For our iterative Fibonacci example, $T(n) = 6n - 8$. We want to show that $T(n)$ is $O(n)$. So we set up our inequality like this:

$$6n - 8 \le cn$$
$$6n \le cn + 8$$
$$n \le cn/6 + 8/6$$

Now pick some value for c. Let's use 6 to cancel the denominator.

$$n \le 6n/6 + 8/6$$
$$n \le n + 4/3$$

Big-O arithmetic

n <= 6n/6 + 8/6
n <= n + 4/3

Now we pick some value for n_0, substitute it for n, and see if the inequality holds. Hmmm, let’s try 1.

1 <= 1 + 4/3

1 is always less than or equal to 1 + 4/3, so in choosing c = 6 and n_0 = 1, we have shown that 6n - 8 is O(n) because 6n - 8 <= 6n for n >= 1. T(n) is O(n). Yippee!

That wasn’t that hard. Let’s do another.

Big-O arithmetic

Examining some code, we determine that $T(n) = 3n^2 + 6n$ and we think that $T(n)$ is $O(n^2)$. How do we prove it?

$$\begin{aligned} 3n^2 + 6n &<= cn^2 \\ 3n\,(n + 2) &<= cn^2 \\ n + 2 &<= cn/3 \\ n &<= cn/3 - 2 \end{aligned}$$

Now we let $c = 3$ to cancel the denominator. That gives us $n <= n - 2$. That won’t work. Let’s try $c = 6$, a multiple of 3, that will also simplify things by cancelling the denominator. That gives $n <= 2n - 2$. That’s much more promising...

Big-O arithmetic

Examining some code, we determine that $T(n) = 3n^2 + 6n$ and we think that $T(n)$ is $O(n^2)$. How do we prove it?

$$\begin{aligned} 3n^2 + 6n &<= cn^2 \\ 3n\,(n + 2) &<= cn^2 \\ n + 2 &<= cn/3 \\ n &<= cn/3 - 2 \\ n &<= 2n - 2 \text{ with } c = 6 \end{aligned}$$

If we let $n_0 = 1$ and substitute it for n, we get

$$\begin{aligned} 1 &<= 2 - 2 \\ 1 &<= 0 \end{aligned}$$

That won’t do.

Big-O arithmetic

Examining some code, we determine that $T(n) = 3n^2 + 6n$ and we think that $T(n)$ is $O(n^2)$. How do we prove it?

$$3n^2 + 6n <= cn^2$$

$$3n\,(n + 2) <= cn^2$$

$$n + 2 <= cn/3$$

$$n <= cn/3 - 2$$

$$n <= 2n - 2 \text{ with } c = 6$$

If we let $n_0 = 2$ and substitute it for n, we get

$$2 <= 4 - 2$$

$$2 <= 2$$

That will do nicely.

Big-O arithmetic

Examining some code, we determine that $T(n) = 3n^2 + 6n$ and we think that $T(n)$ is $O(n^2)$. How do we prove it?

$$\begin{aligned} 3n^2 + 6n &<= cn^2 \\ 3n\,(n + 2) &<= cn^2 \\ n + 2 &<= cn/3 \\ n &<= cn/3 - 2 \\ n &<= 2n - 2 \text{ with } c = 6 \end{aligned}$$

Choosing $c = 6$ and $n_0 = 2$, we have shown that $3n^2 + 6n$ is $O(n^2)$ because $3n^2 + 6n <= 6n^2$ for $n >= 2$.

Big-O arithmetic

The Big-O definition simply(?) says that there is a point n_0 such that for all values of n that are past this point, T(n) is bounded by some multiple of f(n). Thus, if the running time T(n) of an algorithm is $O(n^2)$, we are guaranteeing that at some point we can bound the running time by a quadratic function (a function whose high-order term involves n^2).

Big-O says there’s a function that is an upper bound to the worst case performance for the algorithm.

Big-O arithmetic

Note however that if T(n) is linear and not quadratic, you could still say that the running time is O(n^2). It's technically correct because the inequality holds. However, O(n) would be the more precise claim because it's an even lower upper bound.

Common growth rates

Big-O is for expressing how run time or memory requirements grow as a function of the problem size. Your book has a nice table listing commonly-encountered rates.

Big-O	*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
$O(2^n)$	Exponential
$O(n!)$	Factorial

Common growth rates

Big-O is for expressing how run time or memory requirements grow as a function of the problem size. Your book has a nice table listing commonly-encountered rates.

Big-O	*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(2^n)	Exponential
O(n!)	Factorial

These aren't going to run in reasonable time for any but very small n. Exponential describes that horrifyingly slow version of fibonacci.

Common growth rates

Big-O is for expressing how run time or memory requirements grow as a function of the problem size. Your book has a nice table listing commonly-encountered rates.

Big-O	*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(2^n)	Exponential
O(n!)	Factorial

Often with a loop nested within a loop which is nested within yet another loop.

Common growth rates

Big-O is for expressing how run time or memory requirements grow as a function of the problem size. Your book has a nice table listing commonly-encountered rates.

Big-O	*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(2^n)	Exponential
O(n!)	Factorial

Often with a loop in a loop. Typical of naive sorting, where each element in a collection is compared to every other element.

Common growth rates

Big-O is for expressing how run time or memory requirements grow as a function of the problem size. Your book has a nice table listing commonly-encountered rates.

Big-O	*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(2^n)	Exponential
O(n!)	Factorial

Typical of better sorting.

Common growth rates

Big-O is for expressing how run time or memory requirements grow as a function of the problem size. Your book has a nice table listing commonly-encountered rates.

Big-O	*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(2^n)	Exponential
O(n!)	Factorial

Typical of inefficient, exhaustive search.

Common growth rates

Big-O is for expressing how run time or memory requirements grow as a function of the problem size. Your book has a nice table listing commonly-encountered rates.

Big-O	*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(2^n)	Exponential
O(n!)	Factorial

Better search algorithms reside in this neighbourhood.

Common growth rates

Big-O is for expressing how run time or memory requirements grow as a function of the problem size. Your book has a nice table listing commonly-encountered rates.

Big-O	*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
$O(2^n)$	Exponential
O(n!)	Factorial

How much time to add something to the front of a list, for example.

What exactly are we trying to do?

Sometimes it’s called *complexity analysis*.

Complexity = the rate at which the storage or time grows as a function of the problem size.

The absolute or real growth depends on the machine used to execute the program, the compiler used to construct the program, the cost of individual operations, and many other factors. We would like to have a way of describing the inherent complexity of a program (or piece of a program), independent of machine/compiler considerations. This means that we must not try to describe the absolute time or storage needed. We must instead concentrate on a "proportionality" approach, expressing the complexity in terms of its relationship to some known function. This type of analysis is also known as *asymptotic analysis*.

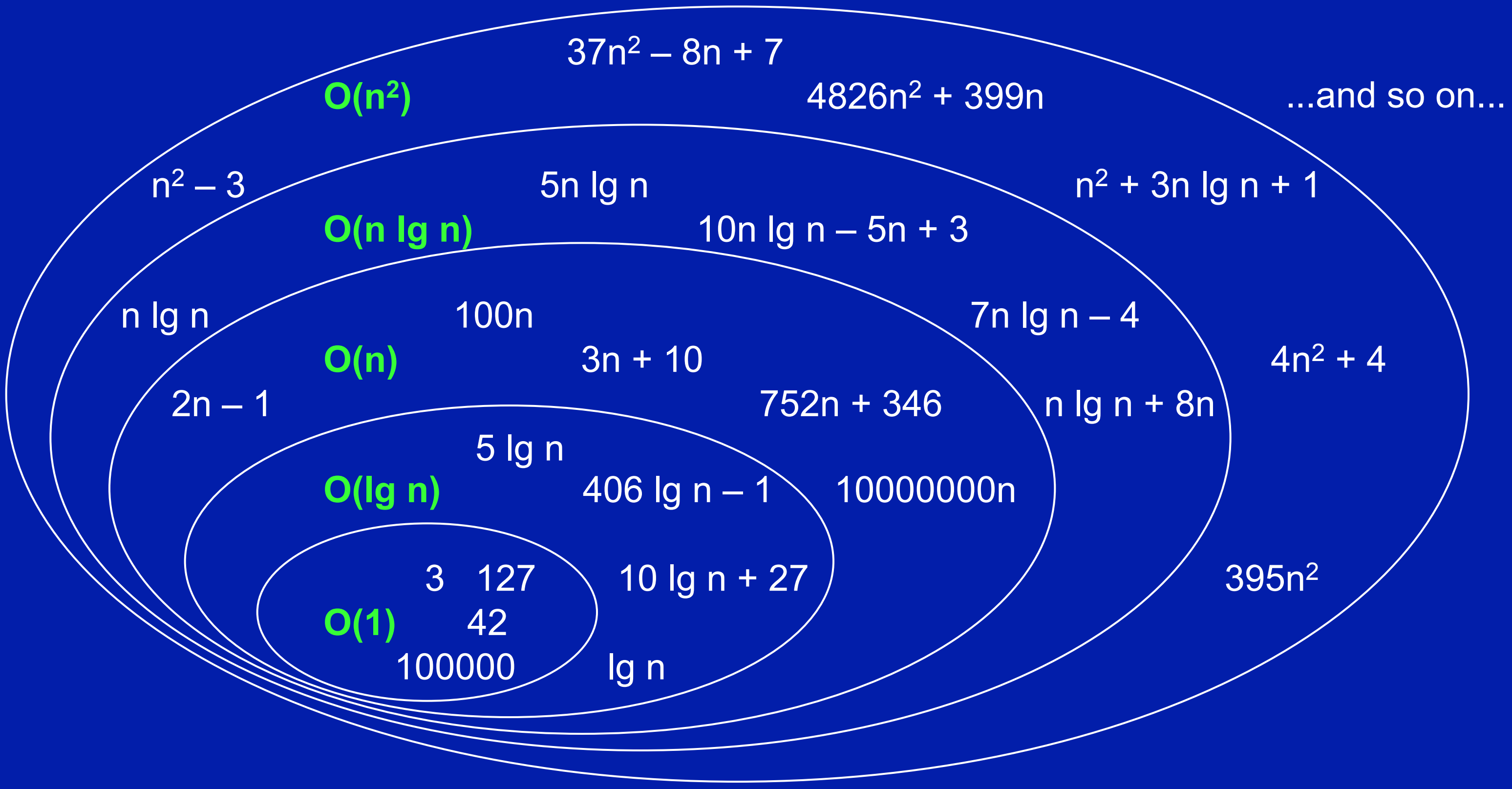
What exactly are we trying to do?

Asymptotic analysis means that we're studying the behaviour of functions f(n) as n gets very large, approaching ∞.

Asymptotic analysis is based on the idea that as the problem size grows, the complexity can be described as a simple proportionality to some known function. This idea is incorporated in the Big-O notation for asymptotic performance. When we say T(n) is O(n^2), we mean that the growth of the run time is proportional to n^2, and that there is at least one specific function, cn^2, that is an upper bound to the growth of the run time.

We're just trying to classify algorithms into these standard complexity categories. We're not trying to say which algorithm among many within the category is better...that's where the constants that we ignore while doing the analysis come back into play again.

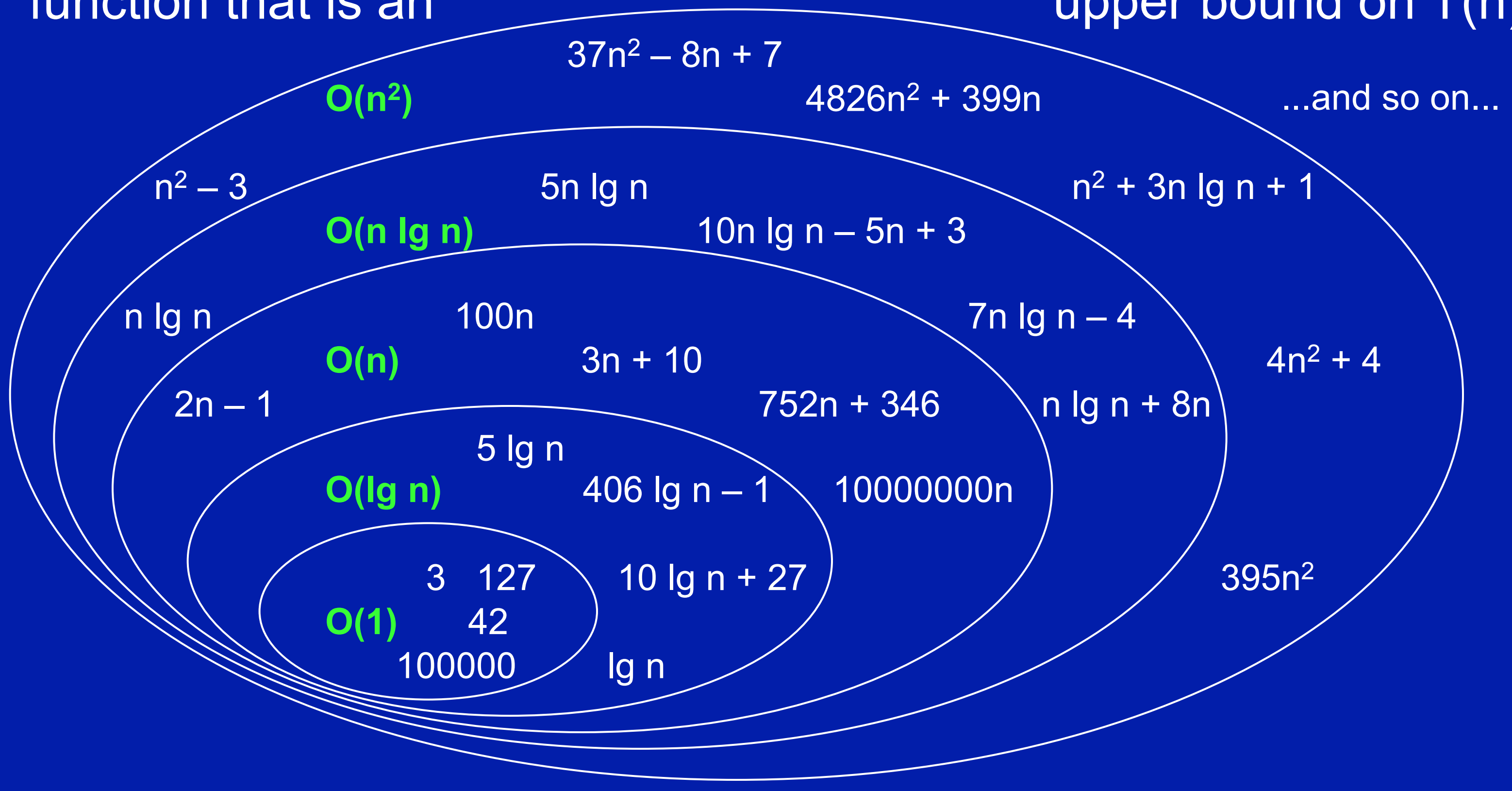
What exactly are we trying to do?



O(g(n)) is the set of functions with smaller or same order of growth as g(n)

What exactly are we trying to do?

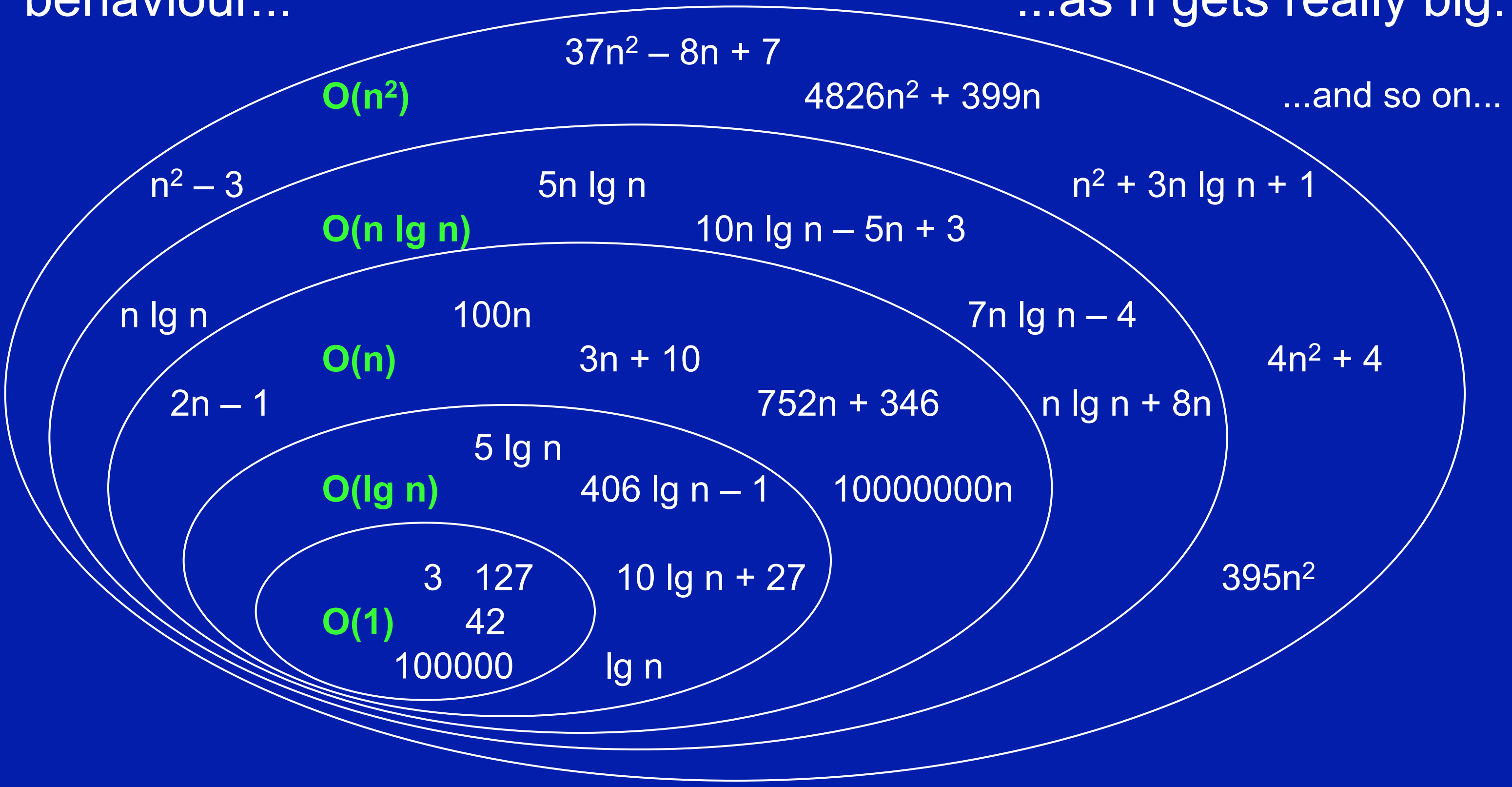
For any f(n) with run time T(n), we're trying to categorize the run time behaviour by placing it in the smallest possible group that contains a function that is an upper bound on T(n)



O(g(n)) is the set of functions with smaller or same order of growth as g(n)

What exactly are we trying to do?

If you start with an algorithm that inherently has $O(n^2)$ complexity, there's no combination of constants that will give it $O(n \lg n)$ behaviour... ...as n gets really big.



$37n^2 – 8n + 7$

$O(n^2)$ $4826n^2 + 399n$ ...and so on...

$n^2 – 3$ $5n \lg n$ $n^2 + 3n \lg n + 1$

$O(n \lg n)$ $10n \lg n – 5n + 3$

$n \lg n$ $100n$ $7n \lg n – 4$

$O(n)$ $3n + 10$ $4n^2 + 4$

$2n – 1$ $752n + 346$ $n \lg n + 8n$

$5 \lg n$

$O(\lg n)$ $406 \lg n – 1$ $10000000n$

3 127 $10 \lg n + 27$ $395n^2$

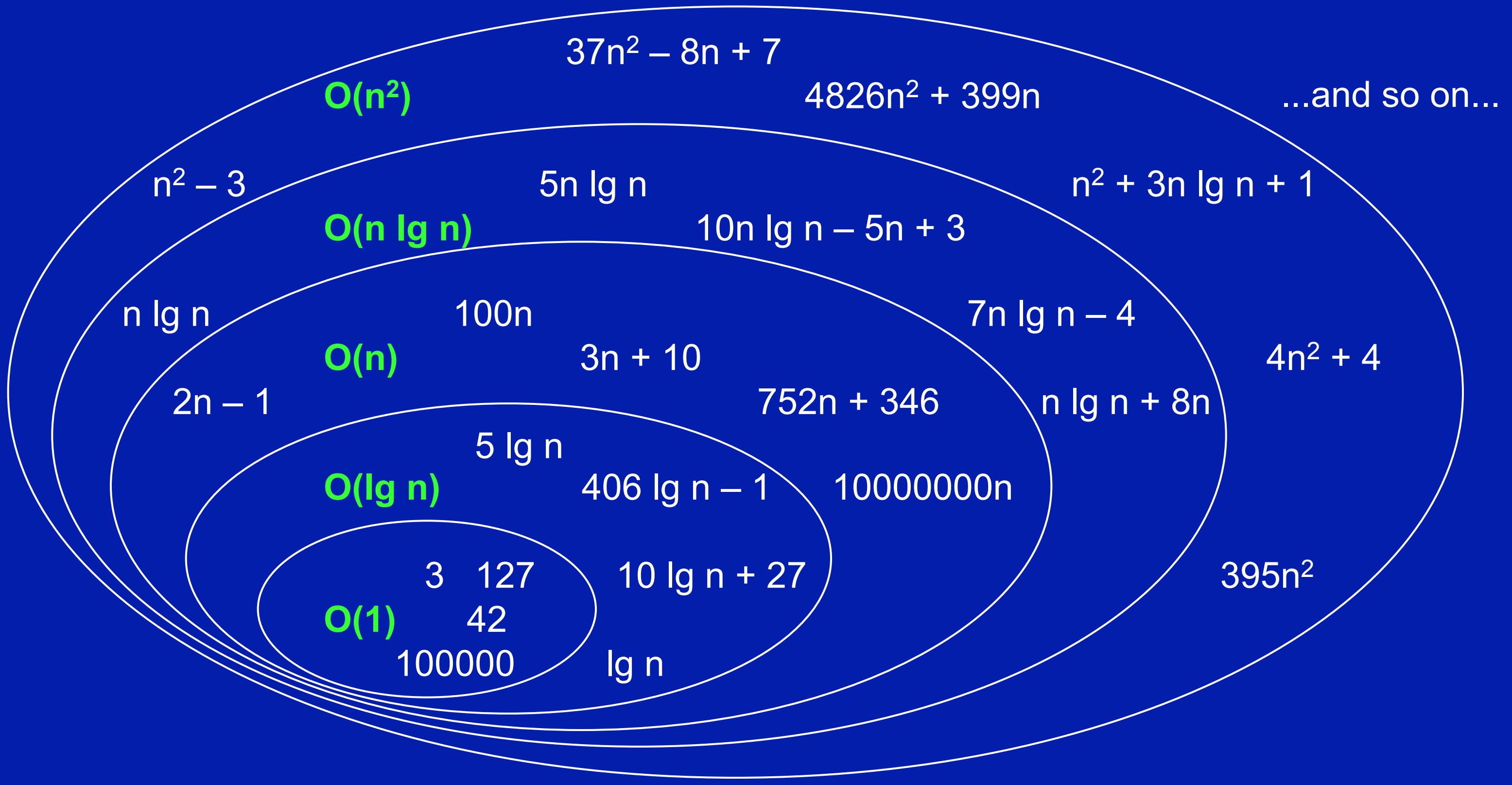
$O(1)$ 42

100000 $\lg n$

$O(g(n))$ is the set of functions with smaller or same order of growth as $g(n)$

What exactly are we trying to do?

However, there may be a limited range of values of n for which the $O(n^2)$ algorithm gives better performance than the $O(n \lg n)$ algorithm.



$37n^2 - 8n + 7$

$O(n^2)$ $4826n^2 + 399n$ ...and so on...

$n^2 - 3$ $5n \lg n$ $n^2 + 3n \lg n + 1$

$O(n \lg n)$ $10n \lg n - 5n + 3$

$n \lg n$ $100n$ $7n \lg n - 4$

$O(n)$ $3n + 10$ $4n^2 + 4$

$2n - 1$ $752n + 346$ $n \lg n + 8n$

$5 \lg n$

$O(\lg n)$ $406 \lg n - 1$ $10000000n$

3 127 $10 \lg n + 27$ $395n^2$

$O(1)$ 42

100000 $\lg n$

$O(g(n))$ is the set of functions with smaller or same order of growth as $g(n)$