University of Waterloo
David R. Cheriton
School of Computer Science

Unit 4 – Analysis of Algorithms

We explore the idea of *efficiency* of algorithms. What does it mean for one algorithm to be faster than another? How do we compare the cost of two different algorithms? What techniques are available for figuring out this cost?

We look at asymptotic notation for analyzing algorithms. This provides a tool for comparing the cost functions of algorithms (or any functions for that matter). We look at how to determine and to simplify these cost functions.

Recall The Simon Game

- Simple Simon:
  - Just one round of play—Simon picks one sequence of length n, and the player must repeat back the sequence.
- Full Simon:
  - n Simple Simon rounds
  - In each round i, the player must press a sequence of length i.
  - Sequence in round i is a prefix of sequence in round $i + 1$.

Playing Time for Simon

- How long does it take for a person to play Simple Simon?
  Let n be number of colours presented and $P_s(n)$ the number of presses required.
  - $P_s(1) = 1, P_s(2) = 2, \ldots$
  - $P_s(n) = n$
- How long does it take for a person to play Full Simon?
  Let n be number of rounds and $P_f(n)$ the number of presses required.
  - $P_f(1) = 1$, $P_f(2) = 3$, $P_f(3) = 6, \ldots$
  - $P_f(n) = 1 + 2 + 3 + \ldots + n = n \cdot (n+1)/2 = n^2/2 + n/2$
  - Less precise, but simpler: $n^2/2 \leqslant P_f(n) \leqslant n^2$
- Note that these are *worst case* times. A game may end earlier (i.e., if you lose)
- In general, in CS 136, we will consider **worst case** time: what is the maximum amount of time required on an input of length n?
  - Other options: best case, average case

Playing Time for Guess-a-number

I'm thinking of a number between 1 and n...

- How many guesses do you need? I'll tell you if you're too high, too low, or you win.
- Think about $G(n)$, number of guesses to win a game of size n in the worst case.
- Depends on your strategy!
- Guess 1, guess 2, ...? For this strategy, $G(n) = n$

Playing Time for Guess-a-number (2)

Smarter approach: Guess in the middle $\lfloor(\ell + h)/2\rfloor$ of the range you know the number is in. This divides the size of game in two!

n	1	2	3	4	...	7	8	...	15	16	17	...
$G(n)$	1	2	2	3	...	3	4	...	4	5	5	...

- Each guess divides the size of the game in two.
- After m guesses, the size of the game is $\frac{n}{2^m}$ (at most).
- How many times can I guess before the game has only one number left?
- $\frac{n}{2^m} = 1 \Rightarrow m = \log_2(n)$
- $G(n) = \log_2(n) + 1$ (add 1 for the last guess)

This is called *Binary Search*, encountered in CS115/135.

Efficiency

"My algorithm can sort a list of a million numbers in 324 ms running in DrRacket 5.1.3.9 on top of MacOS 10.7 on a 3.4 GHz Core i7 with 16GB of RAM."

- Some questions can be answered "easily":
  - What if I double/half the clock speed?
- Some are potentially difficult but out of our scope for now:
  - What if I use a different implementation? Python? C?
  - Different **O**perating **S**ystem (OS)?
- Some can be answered *independently* of other considerations:
  - What about a list of a million and one? Two million?
  - What if the list is "almost" sorted?
  - What if the list is completely backwards?

We want an *abstract* notion of efficiency that does not rely on knowing specific details, yet is informative.

Efficiency

Two Main Abstractions/Simplifications:

1. Step Counting:
   - How many "steps" does an algorithm take?
   - Independent of clock speed, operating system, version of DrRacket, etc., each "step" is a tick of our "abstract clock."
   - To convert back to "real time":

$$\begin{aligned} time &\approx \mathbf{steps} \times \left(\frac{instructions}{step} \times \frac{cycles}{instruction} \times \frac{seconds}{cycle} \right) \\ &\approx \mathbf{steps} \times constant \end{aligned}$$

   - But we generally don't know or care about the constant.
   - We'll still refer to the number of steps as **running time**.

Efficiency

Two Main Abstractions/Simplifications:

1. Step Counting
2. Equivalence Classes for Functions: *"My algorithm takes exactly* $17n \log_2 n + 53$ *steps to sort a list of* n *numbers. Oh no wait I counted wrong hold on:* $16n \log_2 n + 103$*."*
   - Probably nobody cares about this difference. A different computer might change it again.
   - We will express in mathematics how much we don't care using *asymptotic notation* or *"Big-O notation."*

Step Counting

```
(define (len lst)
  (cond
    [(empty? lst) 0]
    [else (+ 1 (len (rest lst)))]))
```

- Let $T(n)$ be the number of substitution steps required by the above algorithm for a list of length n.

Step Counting

```
(define (len lst)
  (cond
    [(empty? lst) 0]
    [else (+ 1 (len (rest lst)))]))
```

- `(len '() )  ;; 3 Steps`
- `(len '(1) )  ;; 11 Steps`
- `(len '(1 2) )  ;; 19 Steps`
- `(len '(1 2 3) ) ;; 27 Steps`
- $T(n) = \ ?$
- $T(n) = 8n + 3$

Step Counting

```
(define (len lst)
  (cond
    [(empty? lst) 0]
    [else (+ 1 (len (rest lst)))]))
```

- $T(n) = 8n + 3$
- Might have guessed $7n + 3$...
- Counting is finicky, does not scale, and the "details" don't really matter anyway.
- **Goal:** *approximate* running time as a function of problem size.

Beginnings of "Order" notation

Don't care about the difference between 7 and 1

- Approximation: For any quantity that is a constant (i.e. doesn't depend on n), we'll write $O(1)$.

Don't care about the difference between $17n$ and n

- Approximation: For any quantity that looks like $a \cdot n$ (where a is a constant) we'll write $O(n)$.

Don't care about "low order" terms:

- For a sum of functions $f(n) + g(n)$, and f is $O(n)$ and g is $O(1)$, then I only care about the $O(n)$.

Order notation

- Hides constant factors: $4n$ is $O(n)$
- Hides "lower order terms": $4n + 5$ is $O(n)$

$O(n)$ stands-in for any function which grows linearly with n.

- $3n^2 + 2$ is $O(n^2)$
- $3n^2 + 2$ is *not* $O(n)$

"Tabular Method": Recursive Functions

```
(define (len lst)
  (cond
    [(empty? lst) 0] ;; Q1 :  A1
    [else (+ 1 (len (rest lst)))])) ;; Q2 :  A2
```

- For each expression Q_i:
  - #Q: How many times is the question evaluated?
  - steps per Q: How many steps does the evaluation take?
  - #A: How many times is the answer evaluated?
  - steps per A: How many steps does the evaluation take (not counting recursion)?
  - $\text{Total}(Q_i) = \#Q \times (\text{steps per } Q) + \#A \times (\text{steps per } A)$
- Total time for function $= \sum \text{Total}(Q_i)$

"Tabular Method": Recursive Functions

```
(define (len lst)
  (cond
    [(empty? lst) 0] ;; Q1 :  A1
    [else (+ 1 (len (rest lst)))])) ;; Q2 :  A2
```

- For a list of length n:

Q	# Q	steps per Q	# A	steps per A	Total
(1)	$n+1$	$O(1)$	1	$O(1)$	$O(n)$
(2)	n	$O(1)$	n	$O(1)$	$O(n)$
					$O(n)$ time

- Q1 is evaluated $n+1$ times: when the length of `lst` is $n, n-1, \ldots, 0$.
- The whole thing takes $O(n)$ steps, or $an+b$ steps for some constants a and b that don't matter.

"Tabular Method": Loops

```
void printline(int n) {
  int i=0;              // S1
  while (i<n) {         // S2
    printf("*");        // S3
    i=i+1;              // S4
  }
  printf("\n");         // S5
}
```

- #S: How many times is each **statement** evaluated?
- time per S: How many steps does each statement take?

For an integer n:

S	# S	time per S	Total
S1	1	$O(1)$	$O(1)$
S2	$n+1$	$O(1)$	$O(n)$
S3	n	$O(1)$	$O(n)$
S4	n	$O(1)$	$O(n)$
S5	1	$O(1)$	$O(1)$
			$O(n)$ time

Print a triangle

```
void printtriangle(int n) {
  int i=n;                  // S1
  while (i>0) {             // S2
     printline(i);          // S3
     i=i-1;                 // S4
  }
}
```

- For an integer n:

S	# S	time per S	Total
S1	1	$O(1)$	$O(1)$
S2	$n+1$	$O(1)$	$O(n)$
S3	n	$O(i)$	*???*
S4	n	$O(1)$	$O(n)$
			???

- `printline(i)` takes $O(i)$ time.
- But i is at most n, so we write $O(n)$.

Print a triangle

```
void printtriangle(int n) {
  int i=n;                  // S1
  while (i>0) {             // S2
     printline(i);          // S3
     i=i-1;                 // S4
  }
}
```

- For an integer n:

S	# S	time per S	Total
S1	1	$O(1)$	$O(1)$
S2	$n+1$	$O(1)$	$O(n)$
S3	n	$O(n)$	$O(n^2)$
S4	n	$O(1)$	$O(n)$
			$O(n^2)$ time

- Whole thing takes $O(n^2)$ time, or $an^2 + bn + c$ time for some constants a, b, c that don't matter.
- We could have done a more exact analysis (`printline(i)` takes $O(i)$ operations) but it would not matter much: – why?

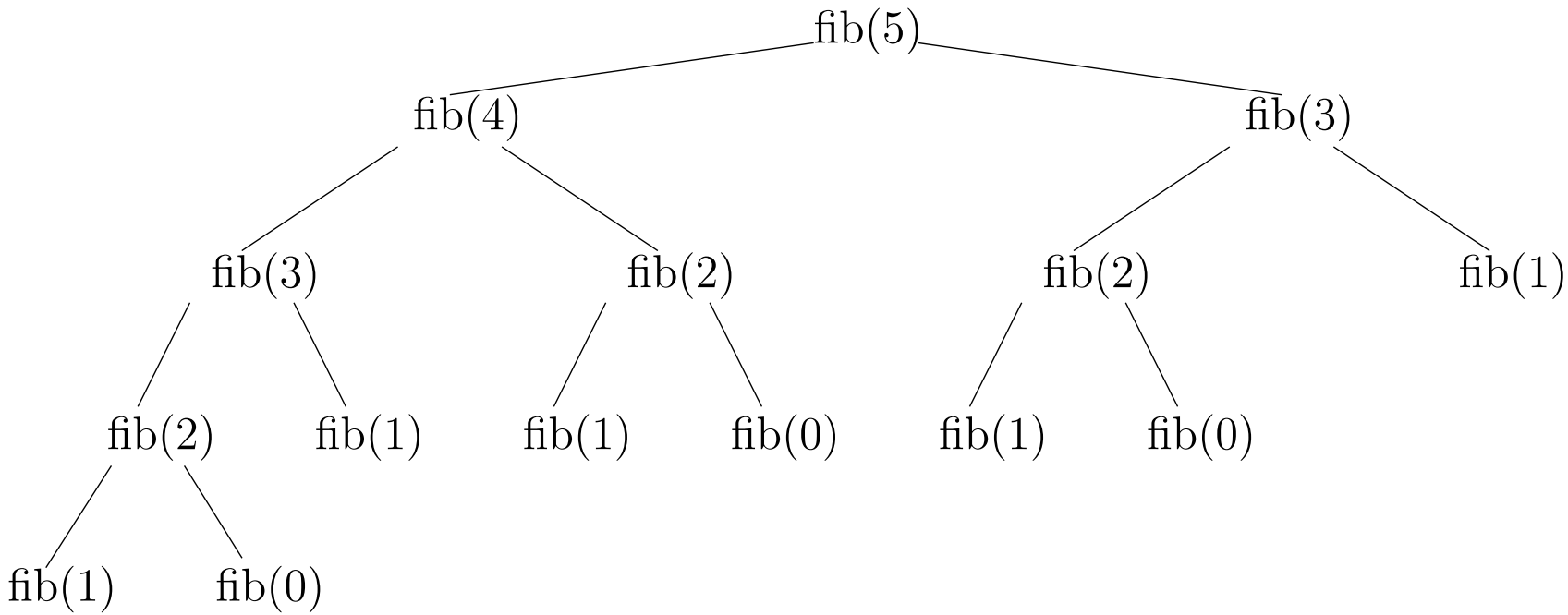
Fibonacci

```
int fib(int n) {
  if (n == 0) return 0;
  else if (n == 1) return 1;
  else
    return fib(n-1) + fib(n-2);
}
```

Just count the number of times $T(n)$ that `fib(n)` is called:

- $T(0) = T(1) = 1$
- $T(n) = 1 + T(n-1) + T(n-2)$ for $n > 1$.

Fibonacci



- $T(n) \leqslant 2^n$ (tree has fewer nodes than a perfect binary tree of height n)
- $T(n) \geqslant 2^{n/2}$ (shortest path of tree has length $n/2$)

Each call to `fib(n)` requires $O(1)$ time plus the cost of the recursive calls.

A call to `fib(n)` takes $O(T(n))$ time.

Better Fibonacci

```
int fibhelper(int n, int this, int next) {
  if (n == 0)
     return this;
  else
     return fibhelper(n-1, next, this+next);
}

int fib(int n) {
  return fibhelper(n,0,1);
}
```

Just count the number of times $T(n)$ that `fibhelper` is called

- $T(n) = 1 + T(n-1) = 1 + 1 + T(n-2) = i + T(n-i)$
- ➡ $T(n) = n + T(0) = n + 1$

A call to `fib(n)` takes $O(n)$ time.

Which one do you want?

- $O(n)$ versus $O(2^n)$
- Say $n = 1000$. Take about 1000 steps or about $2^{1000} \approx 10^{300}$ steps?

Iterative Fibonacci

```
int fib(int n) {
  int this = 0;
  int next = 1;
  while (n > 0) {

    next = next + this;
    this = ???;
    n = n - 1;
  }
  return this;
}
```

Iterative Fibonacci

```
int fib(int n) {
  int this = 0;
  int next = 1;
  int oldnext;
  while (n > 0) {
    oldnext = next;
    next = next + this;
    this = oldnext;
    n = n - 1;
  }
  return this;
}
```

- Loop executes n times
- Time per loop is $O(1)$ operations
- Total cost is $O(n)$ operations

Asymptotic Notation

- Also known as *Order Notation* or *Big-Oh notation*.
- Formalizes a notion of "less-than-or-equal-to" for functions
- $O(g(n))$ is the *set* of functions whose "order" is equal to or less than that of $g(n)$.
- Notation is inconsistently used.
  - $f(n) = O(g(n))$ (common, but we won't use it here)
  - $f(n)$ is $O(g(n))$
  - $f(n) \in O(g(n))$
  - $f(n)$ is in $O(g(n))$
- "Order" captures of notion of "how fast does f grow when n is large?", ignoring constant factors.

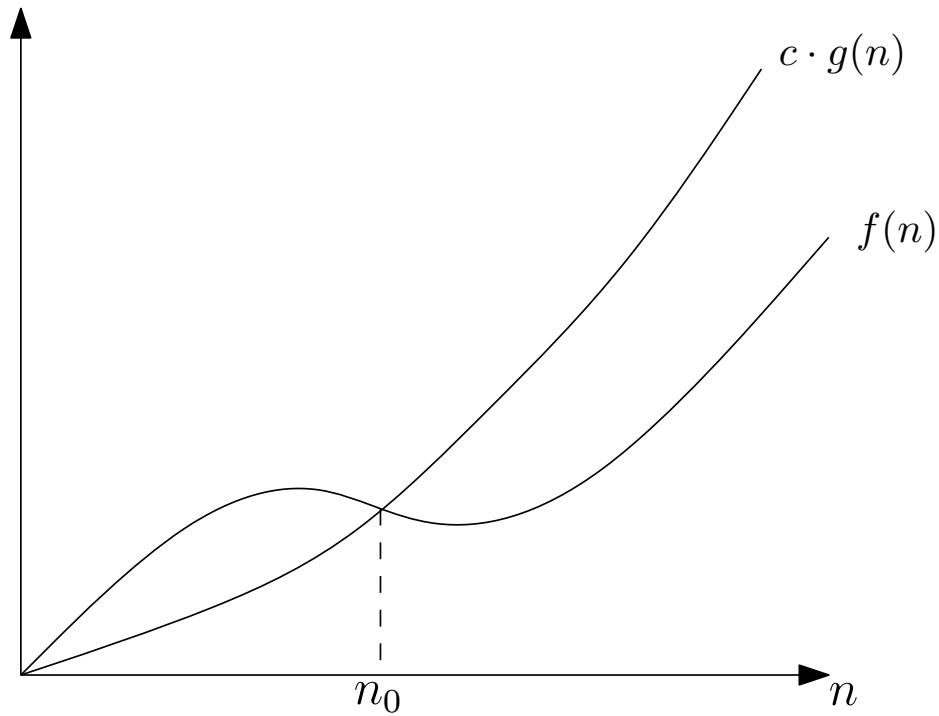
Asymptotic Notation (2)

Let $f(n)$, $g(n)$ be functions from $\mathbb{N}$ to $\mathbb{N}$ (i.e., their input is a positive number and they output a positive number).

In increasing levels formality:

- $f(n) \in O(g(n))$ if for large enough n, $f(n)$ grows no faster than $g(n)$ if we ignore constant factors and "unimportant" (lower order) terms
- $f(n) \in O(g(n))$ if I can pick a positive constant c so that for big enough n, $c \cdot g(n)$ is bigger than $f(n)$
- $f(n) \in O(g(n))$ if there exist positive constants c and n_0 such that for all $n \geqslant n_0$ we have $f(n) \leqslant c \cdot g(n)$

Asymptotic Notation (3)



$f(n) \in O(g(n))$ if there exist positive constants c and n_0 such that for all $n \geqslant n_0$ we have $f(n) \leqslant c \cdot g(n)$

Showing $f(n) \in O(g(n))$

Suppose that $f(n), g(n)$ are functions from $\mathbb{N} \to \mathbb{N}$.
Want to establish that $f(n) \in O(g(n))$

"$f(n) \in O(g(n))$ if there exist positive constants c and n_0 such that for all $n \geqslant n_0$ we have $f(n) \leqslant c \cdot g(n)$"

- **Goal: Produce a c and n_0 that satisfy the definition.**
- Example: is $n + 2 \in O(n)$?

Showing $f(n) \in O(g(n))$

"$f(n) \in O(g(n))$ if there exist positive constants c and n_0 such that for all $n \geqslant n_0$ we have $f(n) \leqslant c \cdot g(n)$"

- $f(n) = n + 2$, $g(n) = n$
- Goal: Produce a c and n_0 such that $n + 2 \leqslant cn$ for all $n \geqslant n_0$

Assume $n_0 = 1$

$$n + 2 \leqslant cn$$

$$1 + 2/n \leqslant c \text{ divide by } n > 0$$

We need $c \geqslant 1 + 2/n$. We know $1 + 2 \geqslant 1 + 2/n$, since $n > 0$. Then, choose $c = 1 + 2 = 3$. If we pick $c = 3$, $n_0 = 1$

$$3 \leqslant c$$

$$1 + 2 \leqslant c$$

$$1 + 2/n \leqslant c \text{ because } n \text{ is positive}$$

$$n + 2 \leqslant cn \text{ multiply through by } n$$

$\Rightarrow n + 2 \in O(n)$

Showing $f(n) \in O(g(n))$

"$f(n) \in O(g(n))$ if there exist positive constants c and n_0 such that for all $n \geqslant n_0$ we have $f(n) \leqslant c \cdot g(n)$"

- **Goal: Produce a c and n_0 that satisfy the definition.**
- Start with hypothesis $n_0 = 1$, inequality $f(n) \leqslant c \cdot g(n)$
- Simplify algebraically until you get $\text{simplefunction}(n) \leqslant c$, (possibly increasing n_0 as you go to make life easier).
- Choose c to satisfy $\text{simplefunction}(n) \leqslant c$, choose n_0 to be the n_0 you ended up with.
- To formally *prove*, state your c and n_0 and run the steps in reverse, i.e., assume n_0 and c are the values you chose in the previous step and work backwards to get $f(n) \leqslant c \cdot g(n)$ for all $n \geqslant n_0$.

Showing $f(n) \in O(g(n))$

http://www.wolframalpha.com/ is handy for plotting
(but a plot is not a proof)

Read the handout on order notation.

Examples

- $f(n) \in O(g(n))$ if I can pick a constant c so that no matter how big n is, $c \cdot g(n)$ bigger than $f(n)$
- $f(n) \in O(g(n))$ if there exist positive constants c, n_0 such that for all $n \geqslant n_0$ we have $f(n) \leqslant c \cdot g(n)$

- Is $2n \in O(3n)$? Yes. $c = 1, n_0 = 1$
- Is $3n \in O(2n)$? Yes. $c = 2, n_0 = 1$
- If $f(n) \in O(g(n))$ and $g(n) \in O(f(n))$, we say f and g are of "the same order." [Aside: We write $f(n) \in \Theta(g(n))$.]
- Is $a \cdot n \in O(n)$ for some constant $a \geqslant 1$? Yes. $c = a, n_0 = 1$

Examples (2)

- $f(n) \in O(g(n))$ if I can pick a constant c so that no matter how big n is, $c \cdot g(n)$ bigger than $f(n)$
- $f(n) \in O(g(n))$ if there exist positive constants c, n_0 such that for all $n \geqslant n_0$ we have $f(n) \leqslant c \cdot g(n)$

- Is $a \cdot n + b \in O(n)$? ($a > 0$) Yes. $c = a + |b|, n_0 = 1$

Examples (3)

- $f(n) \in O(g(n))$ if I can pick a constant c so that no matter how big n is, $c \cdot g(n)$ bigger than $f(n)$
- $f(n) \in O(g(n))$ if there exist positive constants c, n_0 such that for all $n \geqslant n_0$ we have $f(n) \leqslant c \cdot g(n)$

- Is $2n + 10 \in O(n^2)$? Yes. $c = 12, n_0 = 1$
- Is $3n^2 - 6n \in O(n^2)$?
  - Yes. $3n^2 - 6n \leqslant 3n^2$, so let $c = 3, n_0 = 1$.
- Is $3n^2 + 6n \in O(n^2)$?
  - Yes. $3n^2 + 6n \leqslant 3n^2 + 6n^2 \leqslant 9n^2$, so let $c = 9, n_0 = 1$.
- Is $4n \in O(n \log n)$? Yes. $c = 4, n_0 = 2$

Exercises

- $f(n) \in O(g(n))$ if I can pick a constant c so that no matter how big n is, $c \cdot g(n)$ bigger than $f(n)$
- $f(n) \in O(g(n))$ if there exist positive constants c, n_0 such that for all $n \geqslant n_0$ we have $f(n) \leqslant c \cdot g(n)$

Show the following by finding suitable constants c and n_0 in each case:

- $a \cdot f(n) \in O(f(n))$
- $a \cdot n + b \in O(n^2)$ for $a > 0$
- $2n \log n + 10n \in O(n \log n)$
- $2n \log n + 2n \in O(n^2)$

Negative Example

- $f(n) \in O(g(n))$ if there exist positive constants c, n_0 such that for all $n \geqslant n_0$ we have $f(n) \leqslant c \cdot g(n)$

- Is $3n^2 - 6n \in O(n)$? No.
- No matter what c and n_0 you choose, I can always find n such that $3n^2 - 6n \leqslant cn$ is not true.
- In general: $f(n) \notin O(g(n))$ iff for **any** c, and **any** n_0 I can produce $n \geqslant n_0$ for which $f(n) > c \cdot g(n)$.

Showing $f(n) \notin O(g(n))$

- $f(n) \notin O(g(n))$ iff for **any** c, and **any** n_0 I can produce $n \geqslant n_0$ for which $f(n) > c \cdot g(n)$.

- How do we prove $f(n) \notin O(g(n))$?
- Assume $f(n) > c \cdot g(n)$, solve for n.
- $3n^2 - 6n \notin O(n)$

$$3n^2 - 6n > cn$$
$$3n - 6 > c \text{ assume } n > 0$$
$$n > (c+6)/3$$

Given *any* c and n_0, we can choose any $n > \max\{n_0, (c+6)/3\}$ and work our way backwards to show $f(n) > cg(n)$.

Again, read the handout on order notation.

The “best estimate”

Two true statements:

- $3n + 2 \in O(n)$
- $3n + 2 \in O(n^2)$

Which is more informative?

- Typically looking for the “least upper bound”
- The “best estimate”

The "nicest" order

Three true statements

- $14n^2 + 5n - 7 \in O(13n^2 + 12n + 5)$
- $14n^2 + 5n - 7 \in O(13n^2)$
- $14n^2 + 5n - 7 \in O(n^2)$

Which is the most informative?

- Typically look for the simplest upper bound
- For polynomials choose the highest degree term
- For sums of functions, choose the highest order term if possible.

Equivalence Classes

- If $f(n) \in O(g(n))$ and $g(n) \in O(f(n))$ then f and g have the "same order."
- We write $f(n) \in \Theta(g(n))$ (or $g(n) \in \Theta(f(n))$.)
- Sort of like, if $a \leqslant b$ and $b \leqslant a$ then $a = b$.
- Sets of functions of the same order are called "equivalence classes."

Equivalence Classes for Functions

Consider three algorithms, with the following running times:

$$\begin{aligned} T_1(n) &= 3n \\ T_2(n) &= 10000n + 4 \\ T_3(n) &= n + 42 \end{aligned}$$

Consider comparing runtime for $n = 500$ with $n = 1000$:

$$\begin{aligned} T_1(1000)/T_1(500) &= (3 \cdot 1000)/(3 \cdot 500) \\ &= 2 \\ T_2(1000)/T_2(500) &= (10000 \cdot 1000 + 4)/(10000 \cdot 500 + 4) \\ &\approx 1.9999992... \\ T_3(1000)/T_3(500) &= (1000 + 42)/(500 + 42) \\ &\approx 1.92250923... \end{aligned}$$

In all cases, doubling n approximately doubles running time.

Equivalence Classes for Functions

- Functions of the form $T(n) = a \cdot n + b$ (where $a > 0$) are all kind of the same for our purposes
  - This follows from the definition of $O(...)$
- $T(2n)/T(n) \approx 2$, and gets closer the bigger n is. (Try it.)
- In all cases, run time grows *linearly* in the input size n.

Equivalence Classes for Functions

Consider three algorithms, with the following running times:

$$\begin{aligned} T_1(n) &= 3n^2 \\ T_2(n) &= 10000n^2 + 100n + 4 \\ T_3(n) &= n^2 + 42 \end{aligned}$$

Consider comparing runtime for $n = 500$ with $n = 1000$:

$$\begin{aligned} T_1(1000)/T_1(500) &= (3 \cdot 1000^2)/(3 \cdot 500^2) \\ &= 4 \\ T_2(1000)/T_2(500) &= (10000 \cdot 1000^2 + 100 \cdot 1000 + 4)/ \\ &\quad (10000 \cdot 500^2 + 100 \cdot 500 + 4) \\ &\approx 3.9996000... \\ T_3(1000)/T_3(500) &= (1000^2 + 42)/(500^2 + 42) \\ &\approx 3.99949608... \end{aligned}$$

In all cases, doubling n approximately **quadruples** running time.

Equivalence Classes for Functions

- Functions of the form $T(n) = a \cdot n^2 + b \cdot n + c$ (where $a > 0$) are all kind of the same for our purposes
  - This follows from the definition of $O(...)$
- $T(2n)/T(n) \approx 4$, and gets closer the bigger n is. (Try it.)
- In all cases, run time grows *quadratically* in the input size n.

- *They are fundamentally different from the linear functions.*
- They are "bigger" than linear functions in a way that doesn't depend on a, b, and c.
- $O(n) \subseteq O(n^2)$, but $O(n^2) \nsubseteq O(n)$

Cost of doubling input size

$T(n)$	$O(T(n))$	$T(2n)/T(n)$ as $n \to \infty$
a	$O(1)$	1
$a \log(n)$	$O(\log n)$	$\approx$[1] 1
$an + b$	$O(n)$	≈ 2
$an \log(n)$	$O(n \log n)$	$\approx$[2] 2
$an^2 + bn + c$	$O(n^2)$	≈ 4
$an^3 + bn^2 + cn + d$	$O(n^3)$	≈ 8
$a2^n + \text{poly}(n)$	$O(2^n)$	$\approx 2^n$

The functions above are divided into equivalence classes according to "how fast do they grow?"

The $O(...)$ formalism gives us these classes in a rigorous way.

---

1 $\frac{T(2n)}{T(n)} = \frac{a \log 2n}{a \log n} = \frac{\log n + \log 2}{\log n} = 1 + \frac{1}{\log n} \to 1$ when $n \to \infty$

2 $\frac{T(2n)}{T(n)} = \frac{2an \log 2n}{an \log n} = \frac{2 \log n + 2 \log 2}{\log n} = 2 + \frac{2}{\log n} \to 2$ when $n \to \infty$

Some Algorithms and Orders

$O(T(n))$	$T(2n)/T(n),\ n \to \infty$	Alg.
$O(1)$	1	Basic arithmetic operations
$O(\log n)$	≈ 1	Binary search
$O(n)$	≈ 2	Unordered search
$O(n \log n)$	≈ 2	Merge sort
$O(n^2)$	≈ 4	Insertion sort
$O(n^3)$	≈ 8	Matrix inversion[3]
$O(2^n)$	$\approx 2^n$	Travelling Salesman Problem[4]

[3] Gauss–Jordan elimination. There are faster algs.

[4] Dynamic programming