Efficiency

Readings: None

Algorithms

An ***algorithm*** is step-by-step description of *how* to solve a "problem".

Algorithms are not restricted to computing. For example, every day you might use an algorithm to select which clothes to wear.

For most of this course, the "problems" will be function descriptions (*interfaces*), and we will work with *implementations* of algorithms that solve those problems.

The word *algorithm* is named after Muḥammad ibn Mūsā al-Kwārizmī ($\approx$ 800 A.D.).

Problem: Write a Racket function to determine if the sum of a list of positive integers is greater than a specific positive integer.

```
;; sum>?: (listof Int) Int-> Boolean
;;   PRE:  all Ints are positive
;;   POST: produces #t if sum of lon is > k,
;;         #f otherwise
;; (sum>? lon k) determines if sum of lon is > k
```

algorithm 1: (total sum) Calculate the sum of `lon`. Produce `#t` if the sum is greater than `k`, `#f` otherwise.

algorithm 2: (recursive) if `lon` is empty, produce `#f`. If `k` is less than the first element of `lon` (`f`) produce `#t`, otherwise, solve this problem with `(k-f)` and `lon` with `f` removed.

We can *implement* the two algorithms:

```
(define (tsum>? lon k)
  (> (foldl + 0 lon) k))

(define (rsum>? lon k)
  (cond [(empty? lon) #f]
        [(< k (first lon)) #t]
        [else (rsum>? (rest lon) (- k (first lon)))]))
```

Both algorithms solve the problem.

How do we determine which one is “better”?

What do we mean by “better”?

How do we **compare** algorithms?

There are many objective and subjective methods for comparing algorithms:

- How easy is it to understand?
- How easy is it to implement?
- How robust is it?
- How accurate is it?
- How adaptable is it? (Can it be used to solve similar problems?)
- **How fast (efficient) is it?**

In this course, we use ***efficiency*** to objectively compare algorithms.

Efficiency

The most common measure of efficiency is ***time efficiency***, or **how long** it takes an algorithm to solve a problem. Unless we specify otherwise, we will **always mean *time efficiency***.

Another efficiency measure is ***space efficiency***, or how much space (memory) an algorithm requires to solve a problem. We briefly discuss space efficiency at the end of this module.

The *efficiency* of an algorithm may depend on its *implementation*.

To avoid any confusion, we always measure the efficiency of a specific implementation.

Running time

To *quantify* efficiency, we are interested in measuring the ***running time*** of an algorithm.

What **unit of measure** should we use? Seconds?

"My algorithm can sort one billion integers in 9.037 seconds".

- What *year* did you make this statement?
- What machine & model did you use? (With how much RAM?)
- What computer language & operating system did you use?
- Was that the actual CPU time, or the total time elapsed?
- How accurate is the time? Is the 0.037 relevant?

Measuring *running times* in seconds can be problematic.

What are the alternatives?

In Racket, we can measure the total number of (substitution) **steps** required to apply a function.

```
   (rsum>? '(10 5 1) 11)
=> ...                          ;; skipping 9 steps
=> (rsum>? '(5 1) 1)
=> (cond [(empty? '(5 1)) #f]
         [(< 1 (first '(5 1))) #t]
         [else (rsum>? (rest '(5 1)) (- 1 (first '(5 1))))])
=> (cond [#f  #f] [(< 1 (first '(5 1))) #t] ...)
=> (cond [(< 1 (first '(5 1))) #t] ...)
=> (cond [(< 1 5) #t] ...)
=> (cond [#t #t] ...)
=> #t                           ;; 16 total steps
```

We have to use caution when measuring Racket steps, as some built-in functions can be deceiving. For example, the built-in functions `foldl` and `last` may *appear* to only require one substitution step, but we must consider how the functions are internally implemented, and how many "hidden" steps there are.

We will revisit this issue later.

> Do not worry about precisely calculating the number of steps in a Racket expression. We will be introducing some simplification shortcuts soon.

In C, one measure might be how many *machine code* instructions are executed. Unfortunately, this is very hard to measure and is highly dependent on the machine and the environment.

A popular efficiency measurement in C is the number of **operators** executed, or *operations*. We also call each C operation a *step*.

```
sum = 0;              // 1
i = 0;                // 1
while (i < 5) {       // 6
  sum = sum + i;      // 10
  i = i + 1;          // 10
}
```

Like counting Racket steps, this can be a little tedious.

Input size

What is the *running time* (number of steps) required for this implementation of `sum`?

```
;; (sum lon) finds the sum of all numbers in lon
(define (sum lon)
  (cond [(empty? lon) 0]
        [else (+ (first lon) (sum (rest lon)))]))
```

The running time **depends on the length** of the list (`lon`).

If there are n items in the list, it requires $7n + 2$ steps.

We are always interested in the running time *with respect to* the **size of the input**.

Traditionally, the variable n is used to represent the size of the input. m and k are also popular when there is more than one input.

Often, n is obvious from the context, but if there is any ambiguity you should clearly state what n represents.

For example, with lists of strings, n may represent the number of strings in the list, or it may represent the length of all of the strings in the list.

The *running time* of an implementation is a **function** of n, and is written as $T(n)$.

There may also be another ***attribute*** of the input that is important in addition to size.

For example, with *trees*, we use n to represent the number of nodes in the tree and h to represent the *height* of the tree.

In advanced algorithm analysis, n may represent the number of *bits* required to represent the input, or the length of the *string* necessary to describe the input.

Worst case analysis

Let's measure the running time of our `tsum>?` implementation (using the `sum` helper function instead of the deceiving `foldl`):

```
(define (tsum>? lon k)
  (define (sum lst)
    (cond [(empty? lst) 0]
          [else (+ (first lst) (sum (rest lst)))]))
  (> (sum lon) k))
```

The running time of `tsum>?` is $T(n) = 7n + 4$, where n is the length of the list.

Next, we'll measure the running time of our `rsum>?` implementation:

```
(define (rsum>? lon k)
  (cond [(empty? lon) #f]
        [(< k (first lon)) #t]
        [else (rsum>? (rest lon) (- k (first lon)))]))
```

Unfortunately, without knowing `k` and `lon`, we cannot determine how many steps are required.

```
(rsum>? '(10 9 8 7 6 5 4 3 2 1)  1)  ;;   5 steps
(rsum>? '(10 9 8 7 6 5 4 3 2 1) 60)  ;; 102 steps
```

The ***best case*** is when *only the first element* in the list is "visited".

The ***worst case*** is when the result is `#f` and *all* of the list elements are visited.

`rsum>?`: $T(n) = 5$ (best case), $10n + 2$ (worst case)

For `tsum>?`, the best case is the same as the worst case.

`tsum>?`: $T(n) = 7n + 4$.

Which implementation is more efficient?

Is it more "fair" to compare against the best case or the worst case?

Typically, we want to be conservative (*pessimistic*) and use the *worst case*. Unless otherwise specified, the running time of an algorithm is the **worst case running time**.

Comparing the worst case, the `tsum>?` implementation ($7n + 4$) is more efficient than `rsum>?` ($10n + 2$).

> We may also be interested in the *average* case running time, but that analysis is typically much more complicated.

Big O notation

In practice, we are not concerned with the difference between the running times $(7n + 4)$ and $(10n + 2)$.

We are interested in the ***order*** of a running time.

The order is the "dominant" term in the running time without any constant coefficients.

The dominant term in both $(7n + 4)$ and $(10n + 2)$ is n, and so they are both "*order* n".

To represent *orders*, we use ***Big O notation*** (which we define more formally later). Instead of "*order* n", we use $O(n)$.

The "dominant" term is the term that *grows* the largest when n is very large ($n \to \infty$). The *order* is also known as the *"growth rate"*.

In this course, there are only a few orders that we will encounter (arranged from smallest to largest):

$$O(1)\ \ O(\log n)\ \ O(n)\ \ O(n \log n)\ \ O(n^2)\ \ O(n^3)\ \ O(2^n)$$

For example:

- 2013 is $O(1)$
- $100000 + n$ is $O(n)$
- $n + n \log n$ is $O(n \log n)$
- $999n + 0.01n^2$ is $O(n^2)$
- $\frac{n(n+1)(2n+1)}{6}$ is $O(n^3)$
- $n^3 + 2^n$ is $O(2^n)$

When comparing algorithms, the algorithm with the lowest *order* is more efficient.

For example, an $O(n \log n)$ implementation is more efficient than an $O(n^2)$ implementation.

If two algorithms have the same *order*, they are considered equivalent.

Both `tsum>?` and `rsum>?` are $O(n)$, so they are equivalent.

In CS 240 and CS 341 you will study orders and Big O notation much more rigourously.

Big O arithmetic

When *adding* two orders, the result is the largest of the two orders.

- $O(\log n) + O(n) = O(n)$
- $O(1) + O(1) = O(1)$

When *multiplying* two orders, the result is the product of the two orders.

- $O(\log n) \times O(n)$ is $O(n \log n)$
- $O(1) \times O(n) = O(n)$

Algorithm analysis

An important skill in Computer Science is the ability to ***analyze*** a function and determine the *order* of the running time.

With experience and intuition, determining the order will become second nature: *"clearly, `sum` is* $O(n)$*"*

```
(define (sum lon)
  (cond [(empty? lon) 0]
        [else (+ (first lon) (sum (rest lon)))]))
```

In this course, we help you gain experience and work toward building your intuition. We also introduce some helpful tools.

Analyzing simple functions

First, consider **simple** functions (without recursion or iteration).

```
int max(int a, int b) {
  return (a > b) ? a : b;
}
```

In C, all operations (operator executions) are $O(1)$.

Without iteration, there must be a fixed number of operators, so the running time of all operations is:

$$O(1) + O(1) + \ldots + O(1) = O(1)$$

If a simple function calls no other functions, the running time is:

$$T(n) = O(1)$$

If a simple function `f` calls `g` and `h`, the running time of `f` is:

$$T_f(n) = O(1) + T_g(n) + T_h(n)$$

which we know is:

$$T_f(n) = \max(T_g(n), T_h(n))$$

If the parameters of `f` determine which functions are called, remember to use the **worst case**.

```
int f(int n) {
  if (n < 256) {
    return fast(n);
  } else {
    return slow(n);
  }
}
```

$$T_f(n) = T_{slow}(n)$$

In Racket functions, there are no operations and only functions. We must determine the largest running time of all of the functions called.

Consider the following two implementations:

```
;; (single? lst) determines if lst has only one element

(define (a-single? lst)
  (= 1 (length lst)))

(define (b-single? lst)
  (and (not (empty? lst)) (empty? (rest lst))))
```

The running time of `a` is $O(n)$, while the running time of `b` is $O(1)$.

When using functions that are built-in or provided by modules (libraries) you should always be aware of the running times.

Racket running times (lists)

$O(1)$: `cons cons? empty empty? list list?`
`rest first second...tenth`

$O(n)$: `length last reverse list-ref member`$^{\diamond}$
`remove drop-right take-right append`†
`filter`* `map`* `foldl`* `foldr`* `build-list`*

$O(n \log n)$: `sort`

$^{\diamond}$: we discuss `member` later

†: where n is the length of the *first* list

*: when used with a $O(1)$ function, e.g.: (`filter even? lst`)

Racket running times (numeric)

In C, all arithmetic operations are $O(1)$.

When working with Racket *small* integers (valid C integers), the Racket numeric functions are also $O(1)$.

However, because Racket can handle arbitrarily large numbers, the story is a little more complicated. For example, the running time to add two *large* numbers is $O(\log n)$, where n is the largest number.

Regardless, in this course we assume that all built-in numeric functions in Racket are $O(1)$.

Racket running times (equality)

Racket's `equal?` is deceiving: its running time is $O(n)$, where n is the "size" of the smallest argument.

For numeric arguments, we assume `equal?` and `=` are $O(1)$.

Equality illustrates another symbol/string difference: `symbol=?` is $O(1)$, while `string=?` is $O(n)$ (where n is the length of the string).

We discuss the running times of Racket string functions when we introduce C strings.

Because `(member e lst)` depends on `equal?`, its running time is $O(nm)$ where n is the length of the `lst` and m is the size of `e`.

Recurrence relations

Next we consider the running times of recursive functions.

First, we define the running time recursively, which is known in mathematics as a ***recurrence relation***. For example:

$$T(n) = O(n) + T(n-1)$$

We can then look up the recurrence relation in a table to determine the *closed-form* (non-recursive) running time:

$$T(n) = O(n) + T(n-1) = O(n^2)$$

In later courses, you will *derive* the closed-form solutions and *prove* their correctness.

Procedure for recursive functions

1. Identify the order of the function *excluding* any recursion
2. Determine the size of the input for the next recursive call(s)
3. Write the full *recurrence relation* (combine step 1 & 2)
4. Look up the closed-form solution in a table

```
(define (sum lon)
  (cond [(empty? lon) 0]
        [else (+ (first lon) (sum (rest lon)))]))
```

1. $O(1)$
2. $n - 1$
3. $T(n) = O(1) + T(n-1)$
4. $T(n) = O(n)$ (see following table)

The recurrence relations we will encounter in this course are:

$T(n) = O(1) + T(n-1)$	$= O(n)$
$T(n) = O(n) + T(n-1)$	$= O(n^2)$
$T(n) = O(n^2) + T(n-1)$	$= O(n^3)$
$T(n) = O(1) + T(n/2)$	$= O(\log n)$
$T(n) = O(1) + 2T(n/2)$	$= O(n)$
$T(n) = O(n) + 2T(n/2)$	$= O(n \log n)$
$T(n) = O(1) + 2T(n-1)$	$= O(2^n)$

This table will be provided on exams.

Examples: recurrence relations

In the following slides we present several examples. For simplicity and convenience (and to avoid any issues with `equal?`) we use lists of integers.

```
(define (member? e lon)
  (cond [(empty? lon) #f]
        [(= e (first lon)) #t]
        [else (member? e (rest lon))]))
```

$$T(n) = O(1) + T(n-1) = O(n)$$

```
(define (slow-member? e lon)
  (cond [(zero? (length lon)) #f]   ;; oops!
        [(= e (first lon)) #t]
        [else (slow-member? e (rest lon))]))
```

$$T(n) = O(n) + T(n-1) = O(n^2)$$

```
(define (slow-has-duplicates? lon)
  (cond [(empty? lon) #f]
        [(slow-member? (first lon) (rest lon)) #t]
        [else (slow-has-duplicates? (rest lon))]))
```

$$T(n) = O(n^2) + T(n-1) = O(n^3)$$

```
(define (has-duplicates? lon)
  (cond [(empty? lon) #f]
        [(member? (first lon) (rest lon)) #t]
        [else (has-duplicates? (rest lon))]))
```

$$T(n) = O(n) + T(n-1) = O(n^2)$$

```
(define (has-same-adjacent? lon) ;; O(n)
  (cond [(or (empty? lon) (empty? (rest lon))) #f]
        [(= (first lon) (second lon)) #t]
        [else (has-same-adjacent? (rest lon))]))

(define (faster-has-duplicates? lon)
  (has-same-adjacent? (sort lon <)))
```

$$T(n) = O(n \log n) + O(n) = O(n \log n)$$

```
(define (find-max lon)
  (cond
    [(empty? (rest lon)) (first lon)]
    [(> (first lon) (find-max (rest lon))) (first lon)]
    [else (find-max (rest lon))]))
```

$$T(n) = O(1) + 2T(n-1) = O(2^n)$$

```
(define (max2 a b)
  (if (> a b) a b))

(define (fast-max lon)
  (max2 (first lon) (fast-max (rest lon))))
```

$$T(n) = O(1) + T(n-1) = O(n)$$

```
// small_pow2(n) returns the smallest power of 2
//   that is greater than n

int small_pow2(int n) {
  if (n == 0) {
    return 1;
  } else {
    return 2 * small_pow2(n / 2);
  }
}
```

$$T(n) = O(1) + T(n/2) = O(\log n)$$

Sorting algorithms

No introduction to efficiency is complete without a discussion of **sorting algorithms**.

In this course we discuss several sorting algorithms, including ***insertion sort***, which was introduced in CS 135.

The built-in Racket function `sort` is $O(n \log n)$. How does it compare to *insertion sort* and others?

In *insertion sort*, the `first` element of the list is *inserted* into the *sorted* `rest` of the list.

```
(define (insert n slon)
  (cond
    [(empty? slon) (cons n empty)]
    [(<= n (first slon)) (cons n slon)]
    [ else (cons (first slon) (insert n (rest slon)))]))
```

$$T(n) = O(1) + T(n-1) = O(n)$$

```
(define (ins-sort lon)
  (cond
    [(empty? lon) empty]
    [else (insert (first lon) (ins-sort (rest lon)))]))
```

$$T(n) = O(n) + T(n-1) = O(n^2)$$

In ***selection sort***, the smallest element in the list (`m`) is *selected* to be the first element in the new sorted list, followed by the sorted list with `m` removed.

```
(define (sel-sort lon)
  (define m (find-min lon))  ; find-min is O(n)
  (cons m (sel-sort (remove m lon))))
```

$$T(n) = O(n) + T(n-1) = O(n^2)$$

With insertion sort, the *best case* is when the list is already sorted, and the insertion is always at the first position. The best case is $O(n)$.

With selection sort, the *best case* is still $O(n^2)$.

In ***merge sort***, the list is split into two separate lists. Each list is sorted, and then the two sorted lists are `merge`d together.

This approach is known as ***divide and conquer***, where a problem is *divided* into two (or more) smaller problems. Once the smaller problems are completed (*conquered*), the results are combined to solve the original problem.

For *merge sort*, we need a function to `merge` two sorted lists:

```
(define (merge slon1 slon2)
  (cond
    [(empty? slon1) slon2]
    [(empty? slon2) slon1]
    [(< (first slon1) (first slon2))
     (cons (first slon1) (merge (rest slon1) slon2))]
    [else (cons (first slon2)
                (merge slon1 (rest slon2)))]))
```

If the size of the two lists are m and p, then the recursive calls are either with $(m-1)$ and p or with m and $(p-1)$.

However, if we define $n = m + p$ (the combined size of both lists), then each recursive call is of size $(n-1)$.

$$T(n) = O(1) + T(n-1) = O(n)$$

Now, we can complete `merge-sort`:

```
(define (merge-sort lon)
  (define len (length lon))
  (define mid (quotient len 2))
  (define left (drop-right lon mid))   ; O(n)
  (define right (take-right lon mid))  ; O(n)
  (cond [(<= len 1) lon]
        [else (merge (merge-sort left)
                     (merge-sort right))]))
```

$$T(n) = O(n) + 2T(n/2) = O(n \log n)$$

In Racket, the built-in function `sort` uses `merge-sort`.

The final sorting algorithm we discuss in this module is ***quick sort***.

In *quick sort*, an element of the list is selected as a "pivot". The list is then *divided* into two sublists: a list of elements *less than* (or equal to) the pivot, and a list of elements *greater than* the pivot. Each sublist is sorted (*conquered*), and then appended together (along with the original pivot).

Quicksort is also known as Hoare's quicksort (named after the author) or the partition-exchange sort.

```
(define (quick-sort lon)
  (cond [(empty? lon) empty]
    [else (define pivot (first lon))
          (define less (filter (lambda (x)
                                 (<= x pivot)) (rest lon)))
          (define greater (filter (lambda (x)
                                    (> x pivot)) (rest lon)))
          (append (quick-sort less)
                  (list pivot)
                  (quick-sort greater))]))
```

When the pivot is in "the middle" it splits the sublists equally, so:

$$T(n) = O(n) + 2T(n/2) = O(n \log n)$$

But that is the *best case*...

In the worst case, the "pivot" is the smallest (or largest element), so one of the sublists is empty and the other is of size $(n-1)$.

In the worst case, the performance of quick sort is:

$$T(n) = O(n) + T(n-1) = O(n^2)$$

Despite its worst case behaviour, quick sort is still very popular and in widespread use.

The average case behaviour is quite good, and there are some simple methods that can be used to improve the selection of the pivot.

Sorting summary

Algorithm	best case	worst case
insertion sort	$O(n)$	$O(n^2)$
selection sort	$O(n^2)$	$O(n^2)$
merge sort	$O(n \log n)$	$O(n \log n)$
quick sort	$O(n \log n)$	$O(n^2)$

Iterative analysis

The final type of analysis we consider is ***iterative analysis***, where we use **summations** instead of *recurrence relations*. For example:

```
for (i = 0; i < n; i++) {
  printf("*");
}
```

$$T(n) = \sum_{i=0}^{n-1} O(1) = O(n)$$

Because we are primarily interested in *orders*, we simply write:

$\sum_{i=1}^{n}$ instead of $\sum_{i=0}^{n-1}$, $\sum_{i=1}^{10n}$, or $\sum_{i=1}^{\frac{n}{2}}$

Procedure for loops

1. Work from the *innermost* loop to the *outermost*
2. Determine the number of iterations in the loop (in the worst case)
3. Relate the number of iterations to the size of the input (n)

   (or possibly to an outer loop counter)
4. Determine the running time per iteration
5. Write the summation and simplify the expression

```
sum = 0;
for (i = 0; i < n; i++) {
  sum += i;
}
```

$$\sum_{i=1}^{n} O(1) = O(n)$$

Common summations:

$$\sum_{i=1}^{\log n} O(1) = O(\log n)$$

$$\sum_{i=1}^{n} O(1) = O(n)$$

$$\sum_{i=1}^{n} O(n) = O(n^2)$$

$$\sum_{i=1}^{n} O(i) = O(n^2)$$

$$\sum_{i=1}^{n} O(i^2) = O(n^3)$$

The summation index and final value should reflect the *number of iterations* and the *size of the input*, and do not necessarily reflect the actual loop counter values.

```
k = n;                  // n is size of the input
while (k > 0) {
  printf("*");
  k -= 10;
}
```

There are $n/10$ iterations. Because we are only interested in the *order*, $n/10$ and n are equivalent.

$$\sum_{i=1}^{n} O(1) = O(n)$$

When the loop counter changes *geometrically*, the number of iterations is often logarithmic.

```
k = n;                  // n is size of the input
while (k > 0) {
  printf("*");
  k /= 10;
}
```

There are $\log_{10}$ iterations.

$$\sum_{i=1}^{\log n} O(1) = O(\log n)$$

When working with *nested* loops, evaluate the *innermost* loop first.

```
for (j = 0; j < n; j++) {
  for (i = 0; i < j; i++) {
    printf("*");
  }
  printf("\n");
}
```

Inner loop: $\sum_{i=1}^{j} O(1) = O(j)$

Outer loop: $\sum_{j=1}^{n} O(j) = O(n^2)$

Big O revisited

We now revisit *Big O notation* and define it more formally.

First, $O(n)$ is the **set** of all functions whose "order" is **less than or equal** to n.

For example,

$3n \in O(n)$

$100n + 9999 \in O(n)$

But also,

$5 \in O(n)$

$3 \log n + 100 \in O(n)$

Generally, $O(g(n))$ is the **set** of all functions whose "order" is less than or equal to $g(n)$.

By this definition,

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

While it's technically correct to say that most functions you encounter are $O(2^n)$, that is not very useful information to communicate.

In this course, we always want the **most appropriate** order, or in other words, the *smallest* correct order.

$0.01n^3 + 1000n^2 \in O(n^3)$

A slightly more formal definition of Big O is:

$$f(n) \in O(g(n)) \Leftrightarrow f(n) \leq c \cdot g(n)$$

for large n and some positive integer c

This definition makes it clear why we *"ignore"* constant coefficients.

For example,

$9n \in O(n)$ for $c = 10$
$(9n \leq 10n)$

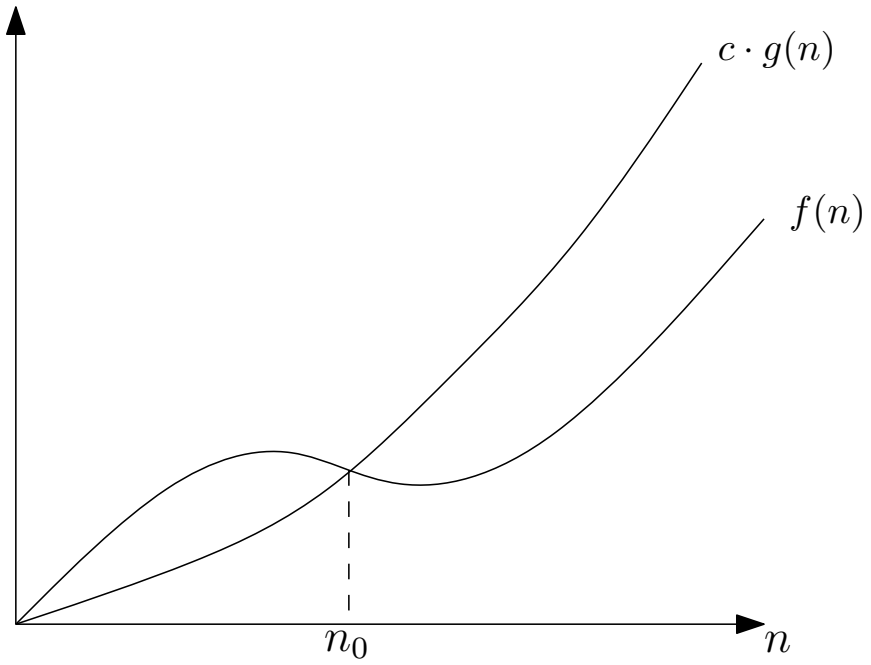
and

$0.01n^3 + 1000n^2 \in O(n^3)$ for $c = 1001$
$(0.01n^3 + 1000n^2 \leq 1001n^3)$

The full definition of Big O is:

$$f(n) \in O(g(n)) \Leftrightarrow \exists c > 0, n_0 \forall n > n_0, f(n) \leq c \cdot g(n)$$

$f(n)$ is in $O(g(n))$ if there exists a positive c and n_0 such that for any value of $n > n_0$, $f(n) \leq c \cdot g(n)$.



Big O describes the *asymptotic* behaviour of a function.

This is **different** than describing the **worst case** behaviour of a algorithm.

Many confuse these two topics but they are completely **separate concepts**. You can asymptotically define the best case and the worst case behaviour of an algorithm.

For example, the best case insertion sort is $O(n)$, while the worst case is $O(n^2)$.

In later CS courses, the formal definition of Big O is used to *prove* algorithm behaviour more rigourously. In this course, we only expect a basic understanding of the asymptotic nature of Big O.

There are other asymptotic functions in addition to Big O.

$$f(n) \in \omega(n) \Leftrightarrow c \cdot g(n) < f(n)$$
$$f(n) \in \Omega(n) \Leftrightarrow c \cdot g(n) \leq f(n)$$
$$f(n) \in \Theta(n) \Leftrightarrow c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$
$$f(n) \in O(n) \Leftrightarrow f(n) \leq c \cdot g(n)$$
$$f(n) \in o(n) \Leftrightarrow f(n) < c \cdot g(n)$$

$O(n)$ is often used when $\Theta(n)$ is more appropriate.

Contract update

You should include the `TIME` (efficiency) of each function that is not $O(1)$ and is not *obviously* $O(1)$.

If there is any ambiguity as to how n is measured, it should be specified.

```
;; merge-sort: (listof Int) -> (listof Int)
;;   PRE:  true
;;   POST: produces lon sorted in ascending order
;;   TIME: O(n logn), n is the length of lon
(define (merge-sort lon) ...)
```

Space complexity

This term we will not discuss any space complexity issues.

We will briefly discuss space complexity in the dynamic memory module.

Goals of this module

You should be comfortable with the terminology for: algorithm, time efficiency, running time, order

You should be comfortable determining the order of an expression

You should have a basic understanding of Big O notation and how n is used to represent the size of the input

You should understand the concept of "worst case" running time, and be able to reason about the worst case for a given implementation

You should be able to reason about the running time for many built-in functions, and how to avoid common design mistakes with expensive operations such as `length`

You should be able to analyze a recursive function, determine its recurrence relation, and look up its closed-form running time in a provided lookup table.

You should understand the four sorting algorithms presented.

You should be able to analyze an iterative function and determine its running time.

You should be able to analyze your own code to ensure it achieves a desired running time.

You should have a basic understanding of the formal definition of big O notation its asymptotic behaviour.

You should understand the running time changes to our contract notation