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" are function descriptions (*interfaces*) and we work with *implementations* of algorithms that solve those problems.

The word *algorithm* is named after Muḥammad ibn Mūsā al-Khwārizmī ($\approx$ 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>? lon k) determines if sum of lon is > k
;; sum>?: (listof Int) Int-> Bool
;; requires: all elements are positive
```

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

algorithm 2: If the first element of `lon` (call this `f`) is greater than `k`, produce `#t`, otherwise reduce `k` by `f` and then repeat for each remaining element. If no more elements remain, produce `#f`.

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 **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* of an algorithm.

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 revisit this issue later.

Do not worry about *precisely* calculating the number of steps in a Racket expression. **You are not expected to count exact the number of steps in an expression.**

We introduce 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***.

For consistency, 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 **T**ime* 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

Consider the running time of our `tsum>?` implementation (using the `sum` helper function instead of the deceiving `foldl`).

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

(define (tsum>? lon k)
  (> (sum lon) k))
```

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

Now consider 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
```

For `rsum>?`, the ***best case*** is when *only the first element* in the list is "visited" and the ***worst case*** is when *all* of the list elements are visited.

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

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

`tsum>?`	$T(n) = 7n + 4$	(all cases)

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 with any constant coefficients removed.

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***. Instead of "*order* n", we use $O(n)$.

We define Big O notation more formally later.

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, we encounter only a few orders

(arranged from smallest to largest):

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

example: orders

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

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

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

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

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

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.

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

There is no “universally accepted” Big O notation.

In many textbooks, **and in this introductory course**, the notation

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

is acceptable.

In other textbooks, and in other courses, this notation may be too informal.

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

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 becomes second nature: *"clearly, the running time of* `sum` *is* $O(n)$*"*

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

In this course, our goal is to give you experience and work toward building your intuition. We also introduce some helpful introductory tools to help perform analysis.

Analyzing simple functions

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

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

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

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

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

For a simple function that calls no other functions, the running time is $O(1)$, otherwise it depends on the other functions called.

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 % 2) {
    return fast(n);
  } else {
    return slow(n);
  }
}
```

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

Consider the following two implementations.

```
;; (single? lst) determines if lst
;;   has exactly 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 a function that is built-in or provided by a module (library) you should always be aware of the running time.

Racket running times (lists)

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

$O(n)$: `length last reverse member`$^{\diamond}$ `remove`$^{\diamond}$
`list-ref`^{i} `drop-right`^{i} `take-right`^{i} `append`^{1}
`filter`* `map`* `foldl`* `foldr`* `build-list`* `apply`*

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

$^{\diamond}$: we discuss `member` later (`remove` is similar)

^{i}: $O(i)$, where i is the position, *e.g.,* (`list-ref lst i`)

^{1}: where n is the length of the *first* list (being appended to)

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

Array efficiency

One of the significant differences between arrays and lists is that any element of an array can be accessed in constant time regardless of the index or the length of the array.

To access the i-th element in a list (*e.g.*, `list-ref`) is $O(i)$.

To access the i-th element in an array (*e.g.*, `a[i]`) is always $O(1)$.

Racket has a *vector* data type that is very similar to arrays in C.

```
(define v (vector 4 8 15 16 23 42))
```

Like C's arrays, any element of a vector can be accessed by the `vector-ref` function in $O(1)$ time.

For the list construction function `list` or the mathematical function `max`, you might expect their running times to be $O(n)$.

However, these functions do not accept lists, but rather a finite (constant) number of parameters.

```
(list 1 2 3 4 5 6 7 8 9 10)
(max 4 8 15 16 23 42)
```

Because the number of parameters is a constant, their running times are $O(1)$.

Racket running times (numeric)

In C, all arithmetic operations are $O(1)$ and in this course we assume that all built-in numeric functions in Racket are $O(1)$.

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

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

Racket running times (equality)

We assume `=` (numeric arguments) is $O(1)$. `symbol=?` is $O(1)$, but `string=?` is $O(n)$, where n is the length of the smallest string*.

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

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`. `(remove e lst)` is similarly $O(mn)$.

* This highlights another difference between symbols & strings.

C running times

The <`string.h`> library functions `strlen`,`strcpy` and `strcmp` are $O(n)$, where n is the length of the string (or the smallest string for `strcmp`).

The <`stdio.h`> library functions `printf` and `scanf` are $O(1)$ except when working with strings (`"%s"`), in which case they are $O(n)$, where n is the length of the string.

Note that the string literal used with `printf` must always be constant length (*i.e.,* `printf("literal")`).

Recurrence relations

To determine the running time of a recursive function we must determine the ***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 *derive* the closed-form solutions and *prove* their correctness.

The recurrence relations we encounter in this course are:

$T(n) = O(1) + T(n - k_1)$	$= O(n)$
$T(n) = O(n) + T(n - k_1)$	$= O(n^2)$
$T(n) = O(n^2) + T(n - k_1)$	$= O(n^3)$
$T(n) = O(1) + T(\frac{n}{k_2})$	$= O(\log n)$
$T(n) = O(1) + k_2 \cdot T(\frac{n}{k_2})$	$= O(n)$
$T(n) = O(n) + k_2 \cdot T(\frac{n}{k_2})$	$= O(n \log n)$
$T(n) = O(1) + T(n - k_1) + T(n - k_1')$	$= O(2^n)$

where $k_1, k_1' \geq 1$ and $k_2 > 1$

This table will be provided on exams.

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. non-recursive functions: $O(1)$ (`empty?`, `first`, `rest`)
2. size of the recursion: $n - 1$ (`rest lon`)
3. $T(n) = O(1) + T(n-1)$ (combine 1 & 2)
4. $T(n) = O(n)$ (table lookup)

Examples: recurrence relations

For simplicity and convenience (and to avoid any `equal?` issues) we use lists of integers in these examples.

```
(define (member1 e lon)
  (cond [(empty? lon) #f]
        [(= e (first lon)) #t]
        [else (member1 e (rest lon))]))
```

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

```
(define (member2 e lon)
  (cond [(zero? (length lon)) #f]
        [(= e (first lon)) #t]
        [else (member2 e (rest lon))]))
```

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

```
(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 (fast-max lon)
  (cond [(empty? (rest lon)) (first lon)]
        [else (max (first lon) (fast-max (rest lon)))]))
```

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

```
(define (clean-max lon)
  (apply max lon))
```

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

Although not common, we can use recursion with arrays.

```
int rsum_array(const int *a, int len) {
  if (len == 0) {
    return 0;
  }
  return a[0] + rsum_array(a + 1, len - 1);
}
```

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

With arrays, it is much more common to use **iteration**.

Iterative analysis

iterative analysis uses **summations**, not *recurrence relations*.

```
for (i = 1; i <= n; ++i) {
  printf("*");
}
```

$$T(n) = \sum_{i=1}^{n} O(1) = \underbrace{O(1) + \ldots + O(1)}_{n} = n \times O(1) = O(n)$$

Because we are primarily interested in *orders*,

$\sum_{i=0}^{n-1} O(x)$, $\sum_{i=1}^{10n} O(x)$, or $\sum_{i=1}^{\frac{n}{2}} O(x)$ are equivalent* to $\sum_{i=1}^{n} O(x)$

* unless x is exponential (*e.g.*, $O(2^n)$).

Procedure for iteration

1. Work from the *innermost* loop to the *outermost*
2. Determine the number of iterations in the loop (in the worst case) in relation to the size of the input (n) or an outer loop counter
3. Determine the running time per iteration
4. Write the summation(s) 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 should reflect the *number of iterations* in relation to the *size of the input* and does 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 $\log_{10} n$ 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: $\sum_{i=0}^{j-1} O(1) = O(j)$

Outer loop: $\sum_{j=0}^{n-1} (O(1) + O(j)) = O(n^2)$

```
bool member(int item, const int a[], int len) {
  for (int i=0; i < len; ++i) {
    if (a[i] == item) {
      return true;
    }
  }
  return false;
}
```

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

```
bool has_duplicates(const int a[], int len) {
  for (int i=0; i < len - 1; ++i) {
    for (int j=i+1; j < len; ++j) {
      if (a[i] == a[j]) return true;
    }
  }
  return false;
}
```

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

Outer loop: $\sum_{i=0}^{n-2} (O(1) + O(n-i)) = O(n^2)$

Note: $\sum_{i=1}^{n} (n-i) = \sum_{i=0}^{n-1} i = O(n^2)$

Do **NOT** put the `strlen` function within a loop.

```
int char_count(char c, const char *s) {
  int count = 0;
  for (int i=0; i < strlen(s); ++i) {    // BAD !!!!
    if (s[i] == c) ++count;
  }
  return count;
}
```

By using an $O(n)$ function (`strlen`) inside of the loop, the function becomes $O(n^2)$ instead of $O(n)$.

Unfortunately, this mistake is common amongst beginners.

This will be harshly penalized on assignments & exams.

Sorting algorithms

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

In this course we discuss several sorting algorithms in C (with arrays) and in Racket (with lists).

In this Section, we only sort **numbers**. Like the built-in Racket `sort` function, these sort functions would be more useful if they were provided with a comparison function. In Section 12 we discuss how this can be done in C.

In ***insertion sort***, we start with an empty (sorted) sequence, and then **insert** each element into the sorted sequence, maintaining the order after each insert. The Racket version has been seen in previous course(s).

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

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

In the following C implementation of insertion sort, `a[0]...a[i-1]` is sorted, and then we look for the correct position `pos` to insert `a[i]` into, "shifting" all of the elements `a[pos]...a[i-1]` one position greater.

```
void insertion_sort(int a[], int len) {
  for (int i=1; i < len; ++i) {
    int val = a[i];
    int pos = i;
    while ((pos > 0) && (a[pos-1] > val)) {
      a[pos] = a[pos-1];
      --pos;
    }
    a[pos] = val;
  }
}
```

$$T(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} O(1) = O(n^2)$$

Selection Sort

In ***selection sort***, the smallest element is *selected* to be the first element in the new sorted sequence, and then the next smallest element is selected to be the second element, and so on.

```
(define (selection-sort lon)
  (cond [(empty? (rest lon)) lon]
        [else (define m (find-min lon))  ; O(n)
              (cons m (selection-sort (remove m lon)))]))
```

Selection sort is $O(n^2)$, even in the best case.

Note: Insertion sort is $O(n)$ in the best case (when the list is already sorted).

In this C implementation of selection sort, the elements `a[0]...a[i-1]` are sorted and we must find the position (`pos`) of the next smallest element, and then "swap" it with `a[i]`.

```
void selection_sort(int a[], int len) {
  for (int i=0; i < len-1; ++i) {
    int pos = i;
    for (int j = i+1; j < len; ++j) {
      if (a[j] < a[pos]) pos = j;
    }
    swap(&a[i], &a[pos]);  // see Section 05
  }
}
```

This C implementation is also $O(n^2)$.

Merge Sort

In ***merge sort***, the list is split into two separate lists. After the two lists are sorted they 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.

We only present merge sort in Racket.

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 $[(m-1)$ and $p]$ or $[m$ 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))]))
```

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

The built-in Racket function `sort` uses `merge-sort`.

Quick Sort

In *quick sort*, an element 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 partition-exchange sort or as Hoare's quicksort (named after the author).

```
(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))]))
```

In our C implementation of quick sort, we:

- select the first element of the array as our “pivot”
- move (“swap”) all elements that are larger than the pivot to the back of the array
- move (“swap”) the pivot into the correct position
- recursively sort the “smaller than” sub-array and the “larger than” sub-array

The core quick sort function `quick_sort_range` has parameters for the range of elements (`first` and `last`) to be sorted, so a wrapper function is required.

```
void quick_sort_range(int a[], int len, int first, int last) {

  if (last <= first) return; // size is <= 1

  int pivot = a[first];        // first element is the pivot
  int pos = last;              // where to put next larger

  for (int i = last; i >= first + 1; --i) {
    if (a[i] >= pivot) {
      swap(&a[pos], &a[i]);
      --pos;
    }
  }
  swap(&a[first], &a[pos]);    // put pivot in correct place
  quick_sort_range(a, len, first, pos-1);
  quick_sort_range(a, len, pos+1, last);
}

void quick_sort(int a[], int len) {
  quick_sort_range(a, len, 0, len-1);
}
```

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 $(n-1)$.

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

Despite its worst case behaviour, quick sort is still popular and in widespread use. The average case behaviour is quite good and there are straightforward methods that can be used to improve the selection of the pivot.

It is part of the C standard library (see Section 12).

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

Binary search

Earlier we saw a C `member` function that is $O(n)$.

```
bool member(int item, int a[], int len) {
  for (int i=0; i < len; ++i) {
    if (a[i] == item) {
      return true;
    }
  }
  return false;
}
```

But what if the array was previously *sorted*?

We can use **binary search** to write an $O(\log n)$ `sorted-member` function.

```
bool sorted_member(int item, const int a[], int len) {
  int low = 0;
  int high = len-1;
  while (low <= high) {
    int mid = low + (high - low) / 2;
    if (a[mid] == item) {
      return true;
    } else if (a[mid] < item) {
      low = mid + 1;
    } else {
      high = mid - 1;
    }
  }
  return false;
}
```

The range (`high-low`) starts out at n (`len`) and decreases by $\frac{1}{2}$ each iteration, so the running time is $O(\log n)$.

Big O revisited

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

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

$$n^2 \in O(n^{100})$$
$$n^3 \in O(2^n)$$

While you can say that n^2 is in the set $O(n^{100})$, it's not very useful information.

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

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 $c = 10, \quad 9n \leq 10n$

and

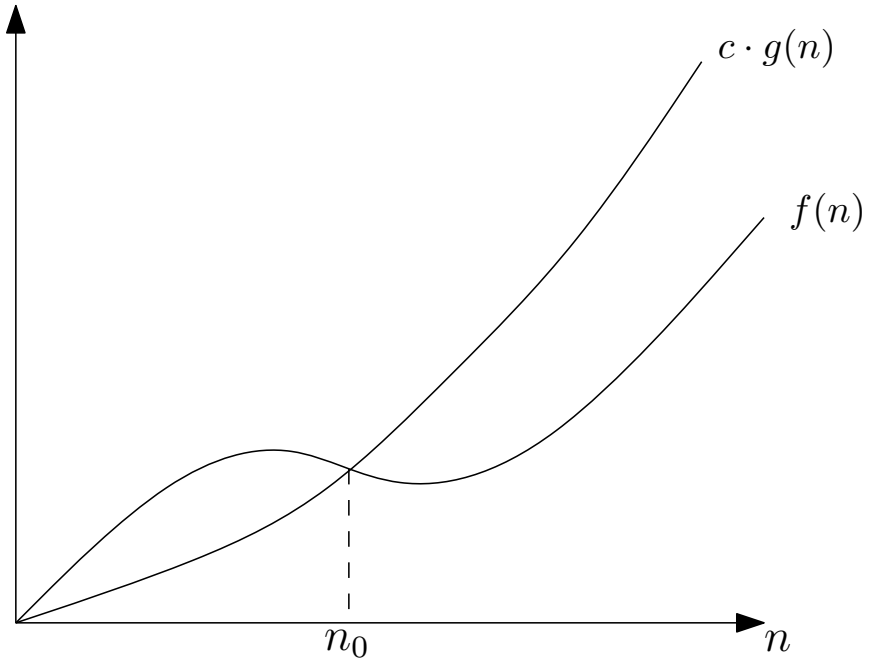
$0.01n^3 + 1000n^2 \in O(n^3)$

for $c = 1001, \quad 0.01n^3 + 1000n^2 \leq 1001n^3$

The full definition of Big O is

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

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



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)
;; time: O(n*logn), n is the length of lon
(define (merge-sort lon) ...)
```

Space complexity

The ***space complexity*** of an algorithm is the amount of **additional memory** that the algorithm requires to solve the problem.

While we are mostly interested in **time complexity**, there are circumstances where space is more important.

If two algorithms have the same time complexity but different space complexity, it is likely that the one with the lower space complexity is faster.

Consider the following two Racket implementations of a function to sum a list of numbers.

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

(define (asum lst)
  (define (asum/acc lst sofar)
    (cond [(empty? lst) sofar]
          [else (asum/acc (rest lst)
                          (+ (first lst) sofar))]))
  (asum/acc lst 0))
```

Both functions produce the same result and both functions have a time complexity $T(n) = O(n)$.

The significant difference is that `asum` uses *accumulative* recursion.

If we examine the substitution steps of `sum` and `asum`, we get some insight into their differences.

```
(sum '(1 1 1))
=> (+ 1 (sum '(1 1)))
=> (+ 1 (+ 1 (sum '(1))))
=> (+ 1 (+ 1 (+ 1 (sum empty))))
=> 3

(asum '(1 1 1))
=> (asum/acc '(1 1 1) 0)
=> (asum/acc '(1 1) 1)
=> (asum/acc '(1) 2)
=> (asum/acc empty 3)
=> 3
```

The `sum` expression "grows" and eventually has $O(n)$ +'s before the final result can be calculated. However, the `asum` expression does not grow and is $O(1)$.

The measured run-time of `asum` is *significantly* faster than `sum` (in an experiment with a list of one million `1`'s, over `40` times faster).

`sum` uses $O(n)$ space, whereas `asum` uses $O(1)$ space.

But **both** functions make the **same** number of recursive calls, how is this explained?

The difference is that `asum` uses **tail recursion**.

A function is ***tail recursive*** if the recursive call is always the **last expression** to be evaluated (the "tail").

Typically, this is achieved by using accumulative recursion and providing a partial result as one of the parameters.

With tail recursion, the previous stack frame can be **reused** for the next recursion (or the previous frame can be discarded before the new stack frame is created).

Tail recursion is more space efficient and avoids stack overflow.

Many modern C compilers detect and take advantage of tail recursion.

Goals of this Section

At the end of this section, you should be able to:

- use the new terminology introduced (*e.g.,* algorithm, time efficiency, running time, order)
- compute the order of an expression
- explain and demonstrate the use of Big O notation and how n is used to represent the size of the input
- determine the "worst case" running time for a given implementation

- deduce the running time for many built-in functions
- avoid common design mistakes with expensive operations such as `strlen` or `length`
- analyze a recursive function, determine its recurrence relation and look up its closed-form running time in a provided lookup table
- analyze an iterative function and determine its running time
- explain and demonstrate the use of the four sorting algorithms presented

- analyze your own code to ensure it achieves a desired running time
- describe the formal definition of Big O notation and its asymptotic behaviour
- explain space complexity, and how it relates to tail recursion
- use running times in your contracts