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.).

There are many objective and subjective methods for comparing algorithms:

- How easy is it to understand?
- How easy is it to implement?
- How accurate is it?
- How robust is it? (Can it handle errors well?)
- 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 measurement? Is the 0.037 relevant?

Measuring *running times* in seconds can be problematic.

What are the alternatives?

Typically, we measure the number of **elementary operations** required to solve the problem.

- In C, we can count the number of operations, or in other words, the number of *operators* executed.
- In Racket, we can count the total number of (substitution) **steps** required, although that can be deceiving for built-in functions[†].

[†] We will revisit the issue of built-in functions later.

You are not expected to count the exact the number of operations.

We only count operations in these notes for illustrative purposes.

We introduce some simplification shortcuts soon.

Input size

What is the number of operations executed for this implementation?

```
int sum_array(const int a[], int len) {
  int sum = 0;
  int i = 0;
  while (i < len) {
    sum = sum + a[i];
    i = i + 1;
  }
  return sum;
}
```

The running time **depends on the length** of the array.

If there are n items in the array, it requires $7n + 3$ operations.

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** (or **length**) 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 also important.

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.

Algorithm Comparison

Problem: Write a function to determine if an array of positive integers contains at least `e` even numbers and `o` odd numbers.

```
// check_array(a, len, e, o) determines if array a
//   contains at least e even numbers and
//   at least o odd numbers
// requires: len > 0
//           elements of a > 0
//           e, o >= 0
```

Homer, Bart and Lisa are debating the best algorithm (strategy) for implementing `check_array`.

Bart just wants to count the total number of odd numbers in the entire array.

```
bool bart(const int a[], int len, int e, int o) {
  int odd_count = 0;
  for (int i = 0; i < len; i = i + 1) {
    odd_count = odd_count + (a[i] % 2);
  }
  return (odd_count >= o) && (len - odd_count >= e);
}
```

According to Lisa, if there are n elements in the array, $T(n) = 8n + 7$.

Remember, you are not expected to calculate this precisely.

Homer is lazy, and he doesn't want to check all of the elements in the array if he doesn't have to.

```
bool homer(const int a[], int len, int e, int o) {
  // only loop while it's still possible
  while (len > 0 && e + o <= len) {
    if (a[len - 1] % 2 == 0) {  // even case:
      if (e > 0) {
        e = e - 1;              // only decrement e if e > 0
      }
    } else if (o > 0) {
      o = o - 1;
    }
    if (e == 0 && o == 0) {
      return true;
    }
    len = len - 1;
  }
  return false;
}
```

The problem with analyzing Homer's code is that it depends not just on the length of the array, but on the contents of the array and the parameters e and o.

```
int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

// these will be fast:
bool fast1 = homer(a, 10, 0, 11);    // false;
bool fast2 = homer(a, 10, 1, 0);     // true;

// these will be slower:
bool slow1 = homer(a, 10, 5, 5);     // true;
bool slow2 = homer(a, 10, 6, 4);     // false;
```

For Homer's code, the ***best case*** is when it can `return` immediately, and the ***worst case*** is when *all* of the array elements are visited.

For Bart's code, the best case is the same as the worst case.

`homer`	$T(n) = 4$	(best case)
	$T(n) = 17n + 1$	(worst case)
`bart`	$T(n) = 8n + 7$	(all cases)

Which implementation is more efficient?

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

Worst case running time

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, Bart's implementation ($8n + 7$) is more efficient than Homer's ($17n + 1$).

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 $(8n + 7)$ and $(17n + 1)$.

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 $(8n + 7)$ and $(17n + 1)$ 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

- $2016 = O(1)$
- $100000 + n = O(n)$
- $n + n \log n = O(n \log n)$
- $999n + 0.01n^2 = O(n^2)$
- $\frac{n(n+1)(2n+1)}{6} = O(n^3)$
- $n^3 + 2^n = O(2^n)$

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 Homer's and Bart's implementations 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.

In this course, our goal is to give you experience and work toward building your intuition:

```
int sum_array(const int a[], int len) {
  int sum = 0;
  for (int i = 0; i < len; ++i) {
    sum += a[i];
  }
  return sum;
}
```

"Clearly, each element is visited once, so the running time of `sum_array` *is* $O(n)$*"*.

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.

```
// sum_array(const int a[], int len) sums the elements
//   of array a
// time: O(n), n is the len of a
```

Analyzing simple functions

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

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

If no other functions are called, there must be a fixed number of operators. Each operator is $O(1)$, so the running time is:

$$O(1) + O(1) + \overset{\text{[fixed \# of times]}}{\cdots} + O(1) = O(1)$$

If a simple function calls other functions, its running time will depend on those functions.

Built-in functions

Consider the following two implementations.

```
// is_len_two(s) determines if the length of s is exactly 2

bool is_len_two_a(const char *s) {
  return strlen(s) == 2;
}

bool is_len_two_b(const char *s) {
  return s[0] && s[1] && (s[2] == 0);
}
```

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.

C running times (strings & I/O)

<`string.h`> functions (*e.g.,* `strlen`, `strcpy`) are $O(n)$, where n is the length of the string. For `strcmp`, n is the length of the smallest string.

<`stdio.h`> functions `printf` and `scanf` are $O(1)$, except when working with strings (`"%s"`), which 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")`).

Racket running times (numeric)

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 it is 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 (lists)

Elementary list functions are $O(1)$:

`cons cons? empty empty? rest first second tenth`

List functions that process the full list are typically $O(n)$:

`length last reverse append`

Abstract list functions (*e.g.,* `map`, `filter`) depend on the consumed function, but are $O(n)$ for straightforward $O(1)$ functions.

The exception is Racket's `sort`, which is $O(n \log n)$.

Racket running times (equality)

We can assume `=` (numeric equality) 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`.

[†] This highlights another difference between symbols & strings.

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 an **array** (*e.g.*, `a[i]`) is always $O(1)$.

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

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.

Iterative analysis

Iterative analysis uses **summations**.

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

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

Because we are primarily interested in *orders*,

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

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

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.

$$\sum_{i=1}^{n/10} O(1) = O(n)$$

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

Inner loop: $\sum_{j=0}^{i-1} O(1) = O(i)$

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

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

```
int char_count(char c, 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.

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)

Revisiting sorting algorithms

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

First we will analyze two recursive sorting algorithms in Racket.

For simplicity, we only consider sorting **numbers**.

When sorting strings or large data structures, you must also include the time to compare each element.

> When analyzing sorting algorithms, one measure of running time is the number of comparisons.

Insertion sort

Recall ***insertion sort*** (from CS 135), where we start with an empty (sorted) sequence, and then **insert** each element into the sorted sequence, maintaining the order after each insert.

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

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

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 is another example of a *divide and conquer* algorithm.

The lists are *divided* into two smaller problems, which are then sorted (*conquered*). The results are combined to solve the original problem.

For now, we can only easily implement 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`.

Selection sort

Recall our C implementation of selection sort:

```
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]);
  }
}
```

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

Quick sort

In our C implementation of quick sort, we:

1. select the first element of the array as our "pivot". $O(1)$
2. move all elements that are larger than the pivot to the back of the array. $O(n)$.
3. move ("swap") the pivot into the correct position. $O(1)$.
4. recursively sort the "smaller than" sub-array and the "larger than" sub-array. $T(?)$

The analysis of step 4 is a little trickier.

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

In Section 08, we implemented binary search on a sorted array.

```
int find_sorted(int item, const int a[], int len) {
  // ...
  while (low <= high) {
    int mid = low + (high - low) / 2;
    // ...
    if (a[mid] < item) {
      low = mid + 1;
    } else {
      high = mid - 1;
  //...
```

In each iteration, the size of the search range (n = `high - low`) was halved, so the running time is:

$$T(n) = \sum_{i=1}^{\log_2 n} O(1) = O(\log n)$$

Algorithm Design

In this introductory course, the algorithms we develop will be mostly straightforward.

To provide some insight into *algorithm design*, we will introduce a problem that is simple to describe, but hard to solve efficiently.

We will present four different algorithms to solve this problem, each with a different running time.

The maximum subarray problem

Problem: Given an array of integers, find the **maximum sum** of any *contiguous* sequence (subarray) of elements.

For example, for the following array:

31	-41	59	26	-53	58	97	-93	-23	84

the maximum sum is 187:

31	-41	59	26	-53	58	97	-93	-23	84

This problem has many applications, including *pattern recognition* in *artificial intelligence*.

Solution A: $O(n^3)$

```
// for every start position i and ending position j
// loop between them (k) summing elements

int max_subarray(const int a[], int len) {
  int maxsofar = 0;
  for (i = 0; i < len; ++i) {
    for (j = i; j < len; ++j) {
      int sum = 0;
      for (k = i; k <= j; ++k) {
        sum += a[k];
      }
      maxsofar = max(maxsofar, sum);
    }
  }
  return maxsofar;
}
```

$$T(n) = \sum_{i=1}^{n} \sum_{j=i}^{n} \sum_{k=i}^{j} O(1) = O(n^3)$$

Solution B: $O(n^2)$

```
// for every start position i,
// check if the sum from i...j is the max

int max_subarray(const int a[], int len) {
  int maxsofar = 0;
  for (i = 0; i < len; ++i) {
    int sum = 0;
    for (j = i; j < len; ++j) {
      sum += a[j];
      maxsofar = max(maxsofar, sum);
    }
  }
  return maxsofar;
}
```

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

Solution C: $O(n \log n)$

We will only describe this recursive *divide and conquer* approach.

1. Find the midpoint position m. $O(1)$
2. Find (a) the maximum subarray from `(0...m-1)`, and
   (b) the maximum subarray from `(m+1...len-1)`. $2T(n/2)$
3. Find (c) the maximum subarray that includes m. $O(n)$
4. Find the maximum of (a), (b) and (c). $O(1)$

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

Solution D: $O(n)$

```
// for each position i, keep track of
// the maximum subarray ending at i

int max_subarray(const int a[], int len) {
  int maxsofar = 0;
  int maxendhere = 0;
  for (i = 0; i < len; ++i) {
    maxendhere = max(maxendhere + a[i], 0);
    maxsofar = max(maxsofar, maxendhere);
  }
  return maxsofar;
}
```

In this introductory course, you are not expected to be able to come up with this solution yourself.

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))))
=> (+ 1 (+ 1 (+ 1 0)))
=> (+ 1 (+ 1 1))
=> (+ 1 2)
=> 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" to $O(n)$ +'s, but the `asum` expression does not use any additional space.

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.

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.

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 $O(n^2)$.

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 number c

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

For example,

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

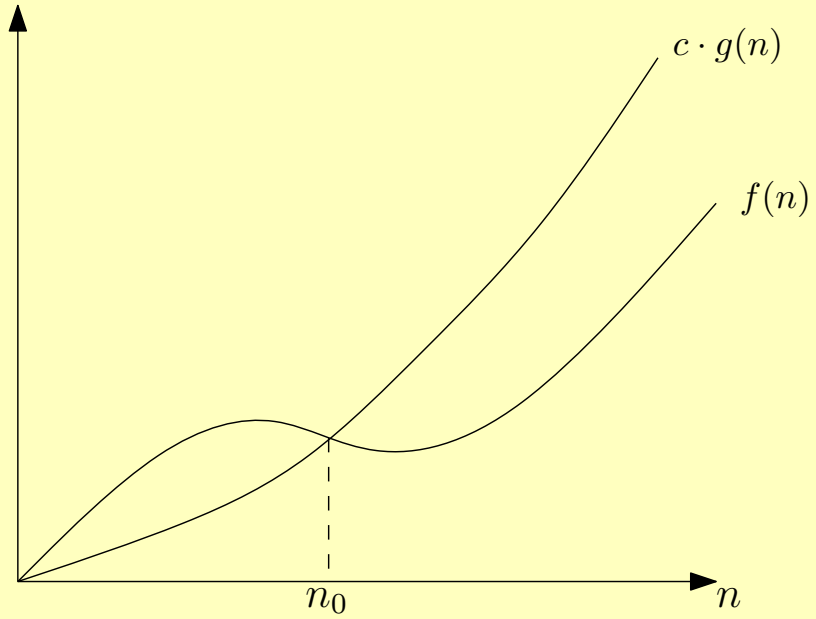
$0.01n^3 + 1000n^2 \in O(n^3)$

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

The full definition of Big O is

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

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



In later CS courses, you will use the formal definition of Big O to *prove* algorithm behaviour more rigourously.

There are other asymptotic functions in addition to Big O.

(for each of the following, $\exists n_0 > 0, \forall n \geq n_0 \ldots$)

$f(n) \in \omega(g(n)) \Leftrightarrow \forall c > 0, c \cdot g(n) \leq f(n)$
$f(n) \in \Omega(g(n)) \Leftrightarrow \exists c > 0, c \cdot g(n) \leq f(n)$
$f(n) \in \Theta(g(n)) \Leftrightarrow \exists c_1, c_2 > 0, c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$
$f(n) \in O(g(n)) \Leftrightarrow \exists c > 0, f(n) \leq c \cdot g(n)$
$f(n) \in o(g(n)) \Leftrightarrow \forall c > 0, f(n) \leq c \cdot g(n)$

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

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`
- 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