Efficiency

Readings: None

The primary goal of this section is to be able to analyze the efficiency of an algorithm.

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. ***Power efficiency*** (power consumption) is also becoming an important measure.

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 revisit the issue of built-in functions later.

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

We only count operations in these notes for illustrative purposes.

We introduce some simplification shortcuts soon.

Data 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 data**.

Traditionally, the variable n is used to represent the **size** (or **length**) of the data. m and k are also popular when there is more than one parameter.

Often, n is obvious from the context, but if there is any ambiguity 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 data 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 data, or the length of the *string* necessary to describe the data.

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 and Bart 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);
}
```

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

// these are 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

Include the `time` (efficiency) of each function that is not $O(1)$ or is $O(1)$ and not *obviously* $O(1)$.

If there is any ambiguity as to how n is measured, specify it.

```
// int f(int a[], int a_len, int b[], int b_len) ...
// time: O(n^2 + m) where n is a_len and m is b_len
```

For a function with a single array, it is obvious:

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

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 depends on those functions.

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 data (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 data* and does not necessarily reflect the actual loop counter values.

```
k = n;                  // n is size of the data
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.

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

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

```
k = n;                  // n is size of the data
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: $\sum_{i=0}^{n-1} (O(1) + O(i)) = O(n^2)$

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 data 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

```
int sum_first(int n) {
  if (n == 0) return 0;
  return n + sum_first(n - 1);
}
```

1. All non-recursive operations: $O(1)$ `+`, `-`, `==`
2. size of the recursion: $n - 1$
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**.

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

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

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

Selection sort

Recall our C implementation of selection sort:

```
void selection_sort(int a[], int len) {
  int pos = 0;
  for (int i = 0; i < len - 1; ++i) {
    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
  }
}
```

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

Insertion sort

The analysis for the worst case of insertion sort is also $O(n^2)$.

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

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

However, in the *best case*, the array is already sorted, and the inner loop terminates immediately. This best case running time is $O(n)$.

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
selection sort	$O(n^2)$	$O(n^2)$
insertion sort	$O(n)$	$O(n^2)$
quick sort	$O(n \log n)$	$O(n^2)$

From this table, it might appear that insertion sort is the best choice.

However, as mentioned with quick sort, the “typical” or “average” case for quick sort is much better than insertion sort.

In Section 10, we will see merge sort, which is $O(n \log n)$ in the worst case.

Binary search

In Section 07, we implemented binary search on a sorted array.

```
int find_sorted(int item, const int a[], int len) {
  // ...
  while (low <= high) {
    mid = (low + high) / 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 are mostly straightforward.

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

We 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;
  int sum = 0;
  for (i = 0; i < len; ++i) {
    for (j = i; j < len; ++j) {
      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;
  int sum = 0;
  for (i = 0; i < len; ++i) {
    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 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.

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 an algorithm.

Many confuse these two topics but they are completely **separate concepts**. The best case and the worst case can both be described asymptotically.

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

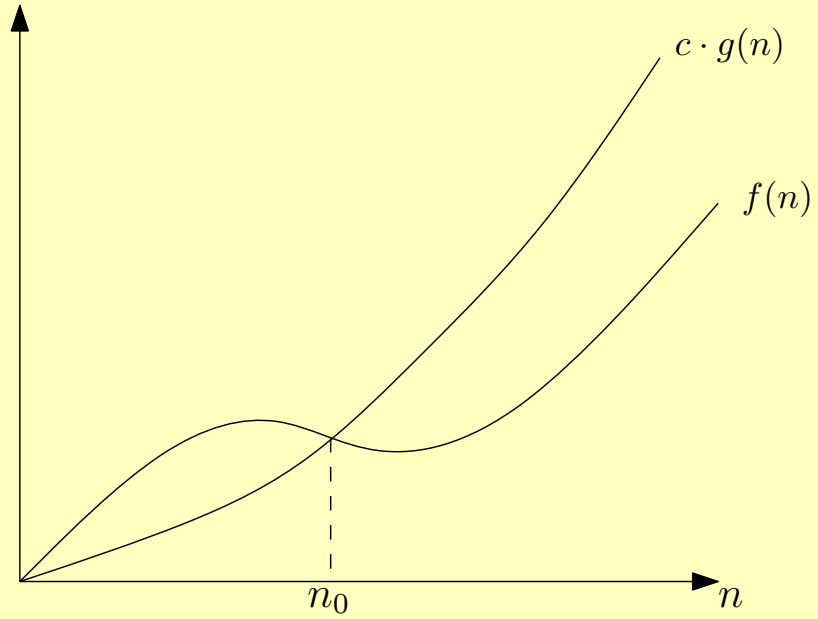
$0.01n^3 + 1000n^2 \in O(n^3)$

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 data
- determine the "worst case" running time for a given implementation
- deduce the running time for many built-in functions

- 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
- use running times in your contracts