# ECE 406 COURSE NOTES
# ALGORITHM DESIGN AND ANALYSIS

**Paolo Torres**

**University of Waterloo**

**Winter 2021**

# Table of Contents

# 1 INTRODUCTION, PROLOGUE, BASIC ARITHMETIC

## 1.1 Algorithms, Correctness, Termination, Efficiency

### 1.1.1 Algorithms

Given the specification for a function, an algorithm is the procedure to compute it.

Example:

$$F: Z_o^+ \rightarrow Z_o^+, \text{ where } F(n) = \begin{cases} 0, if\ n = 0 \\ 1, if\ n = 1 \\ F(n-1) + F(n-2), otherwise \end{cases}$$

Commonly used sets: $N, Z(all\ ints), Z^+(positive\ ints), Z_o^+(non-negative\ ints), R, \ldots$

Fibonacci Sequence:

$FIB_1(n)$

$1\ if\ n = 0\ then\ return\ 0$

$2\ if\ n = 1\ then\ return\ 1$

$3\ return\ FIB_1(n-1) + FIB_1(n-2)$

Important aspects:

- Function has been specified as a recurrence, so a recursive algorithm seems natural
- Imperative (procedural) specification of an algorithm has consequences:
    - Intuiting correctness can be a challenge
    - Intuiting time and space efficiency may be easier
- No mundane error checking, can focus on core logic
- Input value $n$ is unbounded but finite

### 1.1.2 Correctness

Correctness refers to an algorithm's ability to guarantee expected termination. In the case of $FIB_1$, it is a direct encoding of the recurrence.

### 1.1.3 Termination

The end of an algorithm. It can be proven that $FIB_1$ terminates on every input $n \in Z_o^+$ by induction on $n$.

### 1.1.4 Time Efficiency

Can be calculated by counting the number of: $(i)$ comparisons – these happen on Lines $(1)$ and $(2)$, and $(ii)$ number of additions – this happens on Line $(3)$.

Suppose $T(n)$ represents the time efficiency of $FIB_1$:

$$T(n) = \begin{cases} 1, if\ n = 0 \\ 2, if\ n = 1 \\ 3 + T(n-1) + T(n-2), otherwise \end{cases}$$

How bad is $T(n)$? Is it exponential in $n$?

For all $n$, $T(n) \geq F(n)$.

### 1.1.5 Claim 1

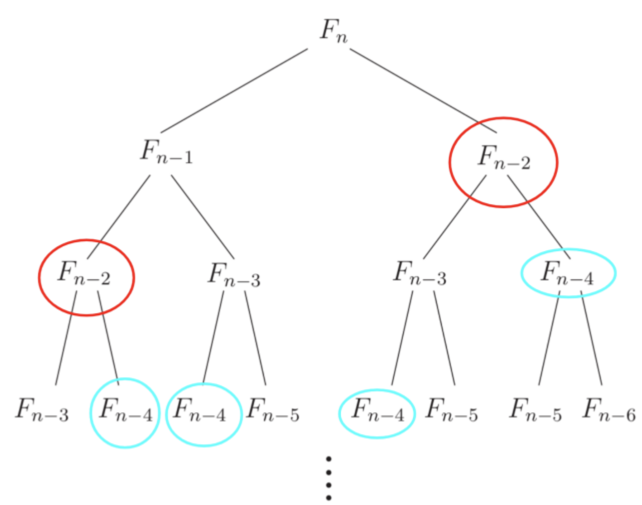
For all $n \in Z_o^+, F(n) \geq \left(\sqrt{2}\right)^n$.

If this claim is true, then $T(n) \geq \left(\sqrt{2}\right)^n$, and because $\sqrt{2} > 1$, $T(n)$ is exponential in $n$.

Proof for the claim: by induction on $n$.

Does a better algorithm exist from the standpoint of time efficiency?

**Figure 0.1** The proliferation of recursive calls in `fib1`.




```
a() {
  do something
  do something else
  invoke b()
  do a third thing
}

b() {
  do something
  c()
}

c() {
  ...
}
```

Recall how subroutine (recursive, in this case) invocation works:

- Every node in the tree corresponds to an invocation of the algorithm
- Sequence of invocations corresponds to a pre-order traversal
- Maximum depth of the call stack at any moment: $n$

Main point in this case: Redundancy, $F_i$, appears more than once.

### 1.1.6 More Efficient Algorithm

$$FIB_2(n)$$
$$if\ n = 0\ then\ return\ 0$$
$$create\ an\ array\ f[0, \dots, n]$$
$$f[0] \leftarrow 0, f[1] \leftarrow 1$$
$$foreach\ i\ from\ 2\ to\ n\ do$$
$$f[i] \leftarrow f[i-1] + f[i-2]$$
$$return\ f[n]$$

Let $U(n)$ be the # of comparisons plus additions on input $n$:

$$U(n) = \begin{cases} 1, if\ n = 0 \\ n, otherwise \end{cases}$$

Linear in $n$ for $n \geq 1$, more efficient than $FIB_1$.

### 1.1.7 Note on Measuring Time Efficiency

Need to pick the right level of abstraction, meaning picking some kind of “hot spot” or “hot operation,” then count. For example, number of additions, comparisons, recursive calls, etc.

## 1.2 Big-O Notation

### 1.2.1 Definition 1 (O)

Let $f: N \to R^+$, and $g: N \to R^+$ be functions. Define $f = O(g)$ if there exists a constant $c \epsilon R^+$ such that $f(n) \leq c \cdot g(n)$.

- $N = Z^+: \{1,2,3, \dots\}, R^+: set\ of\ positive\ real\ numbers$
- Typically consider non-decreasing functions only

### 1.2.2 Definition 2 (Ω)

Define $f = \Omega(g)$ if $g = O(f)$

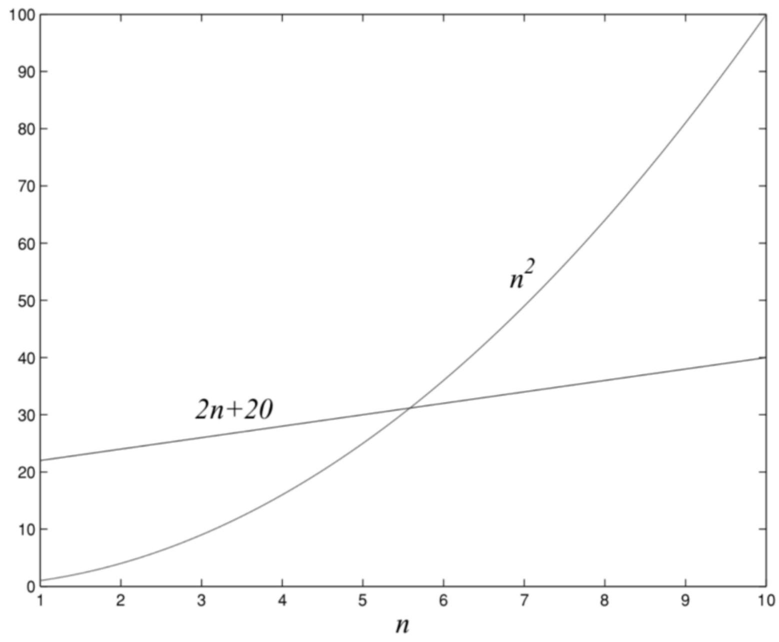
### 1.2.3 Definition 3 (Θ)

Define $f = \Theta(g)$ if $f = O(g)$ and $f = \Omega(g)$

- $f = O(g)$ analogous to $f \leq g$
- $f = \Omega(g)$ analogous to $f \geq g$
- $f = \Theta(g)$ analogous to $f = g$

### 1.2.4 Example

**Figure 0.2** Which running time is better?




Precise answer to this question: depends on $n$.

But in big-O notation:

- $2n + 20 = O(n^2)$. Proof: Adopt as the constant $c \epsilon R$ for any $c > 22$
- $2n + 2 \neq \Omega(n^2) \therefore 2n + 2 \neq \Theta(n^2)$

### 1.2.5 Big-O Explanation

Suppose algorithm A runs in $2n + 20$ time, B in $n^2$, and C in $2^n$. Now suppose the speed of the computer doubles, which algorithm gives the best payoff?

For a given time period $t$, what is the largest input $n$ each algorithm can handle? Set $runtime = t$ and $runtime = 2t$, solve for $n$:

| Algorithm | Old Computer | New Computer |
|---|---|---|
| A | $t/_2 - 10$ | $t - 10$ |
| B | $\sqrt{t}$ | $\sqrt{2} \cdot \sqrt{t}$ |
| C | $\log_2 t$ | $1 + \log_2 t$ |

So, payoff with algorithm A is approximately $2x$, B is $1.4x$, and C is $1 +$.

### 1.2.6 Big-O Simplifications

- Multiplicative and additive constants can be omitted
- $n^a$ dominates $n^b$ for $a > b \geq 0$
- Any exponential dominates any polynomial, any polynomial dominates any logarithm
- Big-O simplifications should be used prudently, not applicable in all settings

## 1.3 Arithmetic

### 1.3.1 Addition

Hypothesize access to a function $T : \{0,1\} \times \{0,\ldots,9\} \times \{0,\ldots,9\} \to \{0,1\} \times \{0,\ldots,9\}$:

| **Carry** | **One Digit** | **Other Digit** | **Result Carry** | **Result Sum** |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 1 |
| | | … | | |
| 1 | 8 | 9 | 1 | 8 |
| 0 | 9 | 9 | 1 | 8 |
| 1 | 9 | 9 | 1 | 9 |

To add 7,814 and 93,404:

$$C \to 1\ 1\ 1\ 0\ 0\ 0$$

$$1 \to 0\ 0\ 7\ 8\ 1\ 4$$

$$2 \to 0\ 9\ 3\ 4\ 0\ 4$$

$$T \to 1\ 0\ 1\ 2\ 1\ 8$$

\# digits needed to encode $x \in Z^+ = \lfloor \log_{10} x \rfloor + 1$

\# bits needed to encode $x \in Z^+ = \lfloor \log_2 x \rfloor + 1$

So, time efficiency of an algorithm to add $x, y \in Z^+$ as measured by number of lookups to T:

- $1 + \lfloor \log_{10}(\max\{x,y\}) \rfloor$ in the best case
- $2 + \lfloor \log_{10}(\max\{x,y\}) \rfloor$ in the worst case
- So, either way, $\Theta(n)$, or linear time, where $n$ is the size of the input

### 1.3.2 Multiplication

For $x, y \in Z_o^+$, encoded in binary:

$$x \times y = \begin{cases} 0, if\ y = 0 \\ 2(x \times {}^{y}/_{2}), if\ y\ even, y > 0 \\ x + 2(x \times \lfloor {}^{y}/_{2} \rfloor), otherwise \end{cases}$$

Straightforward encoding as recursive algorithm $MULTIPLY(x, y)$.

**Figure 1.1** Multiplication à la Français.

```
function multiply(x, y)
Input: Two n-bit integers x and y, where y ≥ 0
Output: Their product

if y = 0: return 0
z = multiply(x, ⌊y/2⌋)
if y is even:
   return 2z
else:
   return x + 2z
```

Worst case running time:

- $Let\ \#\ bits\ to\ encode\ each\ of\ x\ and\ y = n$
- $\#\ recursive\ calls = \Theta(\mathrm{n})$
- $In\ each\ call$:
  - One comparison to $0$, one division by $2$ (right bit shift), one assignment to $z$, one check for evenness (check LSB), one multiplication by $2$ (left bit shift), one addition of $O(n)$-bit numbers
- So, $O(n^2)$ in the worst case

### 1.3.3 Division

**Definition 1:** Given $x \epsilon Z_o^+, y \epsilon Z^+$, the pair $\langle q, r \rangle$ where $q \epsilon Z_o^+$, $r \epsilon \{0, 1, \ldots, y-1\}$ of $x$ divided by $y$ are those that satisfy:

$$x = q \cdot y + r$$

**Claim 1:** For every $x \epsilon Z_o^+$, $y \epsilon Z^+$, $\langle q, r \rangle$ as defined above $(i)$ exists, and $(ii)$ is unique.

To specify a recurrence for $\langle q, r \rangle$, denote as $\langle q', r' \rangle$, the result of $\lfloor {}^{x}/_{2} \rfloor$ divided by $y$. Now:

$$\langle q, r\rangle = \begin{cases} \langle 0,0\rangle, if\ x = 0 \\ \langle 2q', 2r'\rangle, if\ x\ even\ and\ 2r' < y \\ \langle 2q', 2r'+1\rangle, if\ x\ odd\ and\ 2r'+1 < y \\ \langle 2q'+1, 2r'-y\rangle, if\ x\ even\ and\ 2r' \geq y \\ \langle 2q'+1, 2r'+1-y\rangle, otherwise \end{cases}$$

**Claim 2:** The above recurrence is correct.

*Proof:* Cases are exhaustive. Proof by case-analysis and induction on # bits to encode $x$.

By induction assumption: $0 \leq r' \leq y-1 \therefore 0 \leq 2r' \leq 2y-2$.

**Figure 1.2** Division.

```
function divide(x, y)
Input: Two n-bit integers x and y, where y ≥ 1
Output: The quotient and remainder of x divided by y

if x = 0: return (q, r) = (0, 0)
(q, r) = divide(⌊x/2⌋, y)
q = 2 · q,  r = 2 · r
if x is odd: r = r + 1
if r ≥ y: r = r − y,  q = q + 1
return (q, r)
```

Running time: $O(n^2)$.