

# CS 201 Notes

## Introduction to Computer Science

Jaeden Bardati

*Last modified November 3, 2021*

## 0 Course Overview

September 10, 2021

### 0.1 What is Computer Science?

Computer science is the study of **algorithms**.

An **algorithm** is an effective method for solving a problem, expressed as a finite sequence of steps.

The development of algorithms works in this recurring order:

- **Design**
- **Analysis**
- **Implement**
- **Experiment**

In the **design** phase, we design an algorithm using pseudo-code. In the **analysis** phase, we need to analyze the correctness and the efficiency. That is to say, we make sure that our algorithm will work and completes in a reasonable amount of time. During the **implementation**, the algorithm is written in code on a computer. Finally, the algorithm is run and debugged in the **experiment** phase. This process repeats.

## 1 Introduction to Algorithms

### 1.1 Design

September 11, 2021

An algorithm is a step by step procedure to solve a problem. For example,

- Step 1: Do something
- Step 2: Do something
- ...
- Step n: Do something

There are three basic types of steps. Note that there is a fourth (recursion), but it is not covered in this course.

### 1.1.1 Sequential Steps

**Do a single task.**

*For example:* Let  $x$  be a variable. A sequential step could be to add 1 to  $x$ .

### 1.1.2 Conditional Steps

**Ask a question** that supports only **logic answers** (true or false answer).

*For example:* Let  $x$  be a variable. A conditional step could be to ask if  $x > 0$ . If so, add 1 to  $x$ ; otherwise, subtract 1 from  $x$ .

### 1.1.3 Iterative Steps (loops)

**Repeat a task** until a certain condition is satisfied. This step links the sequential step to the conditional steps.

*For example:* If you have a recipe that you need to add water until its dry. An iterative step would be one where you add  $\frac{1}{2}$  cup to mixture while mixture is dry.

## 1.2 Case Study: Addition Algorithm

September 11, 2021

Let's say we want to add 472 to 593. We would do it like so:

$$\begin{array}{r} \phantom{+} \overset{1}{4} \phantom{0} 7 \phantom{0} 2 \\ + \phantom{0} 5 \phantom{0} 9 \phantom{0} 3 \\ \hline 1 \phantom{0} 0 \phantom{0} 6 \phantom{0} 5 \end{array}$$

If you know how to add these numbers, you know how to do it for any numbers. Why? Because we used a sequence of steps to solve it: We used an algorithm. What is this algorithm?

We know that we can break down the work for each of the digits. It is an iterative statement for each digit. What is the work we need to do at each iteration?

First, we add the digits plus the carry in (starts at 0). This is a **sequential step**. Then, we ask if it is greater than 9. If so, we set the resulting digit as the addition subtracted by 10 and set the carry out (which is the carry in for the next step) to 1; otherwise, we simply set the resulting digit as the addition and set the carry out to be 0. This is a **conditional step**. Then we repeat this process until we have no more digits to add. This is an **iterative step**.

Now, we need to conceptualize this. Let's let  $m \geq 1$  be the number of digits. Let us define  $a_i$  (first number digits),  $b_i$  (second number digits) and  $c_i$  (resulting number digits) as follows:

$$\begin{array}{rcccc}
 & a_{m-1} & \dots & a_1 & a_0 \\
 + & b_{m-1} & \dots & b_1 & b_0 \\
 \hline
 c_m & c_{m-1} & \dots & c_1 & c_0
 \end{array}$$

Let's write the steps of our algorithm:

---

#### Algorithm 1

Addition Algorithm

---

```

1: get  $m$  (provided by user)
2: get  $a_{m-1}, \dots, a_1, a_0$  and  $b_{m-1}, \dots, b_1, b_0$  (provided by user)
3: set  $i = 0$  (digit index) and  $\text{carry} = 0$ 
4: while ( $i \leq m - 1$ ) do
5:   set  $c_i = a_i + b_i + \text{carry}$ 
6:   if ( $c_i \geq 10$ ) then
7:     set  $c_i = c_i - 10$ 
8:     set  $\text{carry} = 1$ 
9:   else
10:    set  $\text{carry} = 0$ 
11:   set  $i = i + 1$ 
12: set  $c_m = \text{carry}$ 
13: print  $c_{m-1}, \dots, c_1, c_0$ 
14: stop

```

---

This can be programmed now using a programming language.

If you want to test your algorithm, you can perform a **trace**. A trace is when you go through the algorithm yourself step by step for a test case.

## 1.3 Pseudocode

September 11, 2021

Pseudocode is:

- Simplified
- A tradeoff between natural and programming languages
- Not unique

We will now look into the types of statements and the syntax we will use.

### 1.3.1 Sequential statements

- **Input:** Get “variable”. E.g. Get  $m$ , Get radius
- **Computation:** Set variable = expression. E.g. Set area =  $\pi \times \text{radius}^2$
- **Output:** Print “variable”. E.g. Print area.

---

#### Algorithm 2

Calculates the average of three numbers.

---

```
1: get  $x, y, z$ 
2: set average =  $\frac{x+y+z}{3}$ 
3: print average
4: stop
```

---

### 1.3.2 Conditional statements

- If (condition) Then  
    operation  $T_1$   
    operation  $T_2$   
    ...  
    operation  $T_m$   
Else  
    operation  $F_1$   
    operation  $F_2$   
    ...  
    operation  $F_n$

When a conditional statements within a conditional statement it is called a **nested** conditional statements (or nested ifs). This also applies to nested iterative statements (or nested loops).

---

**Algorithm 3**

Print average of three number if the first number is larger than 0, otherwise print an error message.

---

```
1: get  $x, y, z$ 
2: if ( $x \geq 0$ ) then
3:   set average =  $\frac{x+y+z}{3}$ 
4:   print average
5: else
6:   print "Bad Data"
7: stop
```

---

### 1.3.3 Iterative statements

- While (condition) Do step  $i$  to step  $j$ 
  - Step  $i$ : operation
  - Step  $i + 1$ : operation
  - ...
  - Step  $j$ : operation
- Stop

There are some considerations that you have to be careful of when writing a while loop:

If the condition is initially false, the loop will not execute at all.

If the condition is initially true, the loop is iterated until the condition is false. This means that *at least one step should change the condition at some point*. If this is forgotten, the loop will run forever (called an **infinite loop**)! This is considered a **fatal error**.

---

**Algorithm 4**

Compute and print the square of the first 100 integers.

---

```
1: set index=1
2: while ( $\text{index} \leq 100$ ) do
3:   set square = index * index
4:   print square
5:   set index = index + 1
6: stop
```

---

### 1.3.4 The Do-While

September 23, 2021

The do-while is similar to the while-do, but you check the condition after the do section.

- Do
  - Step  $i$ : operation
  - Step  $i + 1$ : operation
  - ...
  - Step  $j$ : operation
- While (condition)

This will always execute at least once. It will only execute once if the condition is false, and multiple if it is true. For example,

---

#### Algorithm 5

Read a var  $x$ , print  $\sqrt{x}$  and repeat the process as long as requested by user.

---

```
1: get  $x$ 
2: do
3:   get  $x$ 
4:   if ( $x \geq 0$ ) then
5:     set root =  $\sqrt{x}$ 
6:     print root
7:   else
8:     print "Bad data"
9:   print "Do you want to continue? Y/N"
10:  get continue
11: while (continue == 'Y')
12: stop
```

---

## 1.4 The Sequential Search Algorithm

September 24, 2021

Given a dataset of a given size  $N$  and given a target, is the target in the dataset?

For example, if the dataset is the list [13, 4, 5, -20, 45, 112] with  $N = 6$ . Is the target 45 in the dataset? Yes. Is the target 130 in the dataset? No.

To solve for much larger values of  $N$ , we can search sequentially through the list until we reach the target (the target is in the list), or the end of the list (the target is not in the list).

Let us assume we have a list of size  $N$  whose elements are  $L_1, L_2, \dots, L_N$  and a target element called Target.

---

**Algorithm 6****Sequential Search Algorithm**

---

```
1: get  $L_1, L_2, \dots, L_N, N, \text{Target}$ 
2: set Found = No
3: set  $i = 1$ 
4: while Found = No AND  $i \leq N$  do
5:   if ( $L_i == \text{Target}$ ) then
6:     set Found = Yes
7:   else
8:     set  $i = i + 1$ 
9: if Found = Yes then
10:  print "Target in list."
11: else
12:  print "Target not in list."
13: stop
```

---

## 1.5 Find Largest

September 25, 2021

Given a list of  $N$  elements, what is the largest element in the list?

For example, if the list is [19, 41, 12, 63, 22] with  $N = 5$ . Each element has an index (1, 2, 3, 4, and 5, respectively). The largest element is 63.

To solve, we can store a value for the largest value starting with the first number and then iterate through each other element of the list and update the largest number to the current element if it is larger than the stored value for the largest element.

Let us assume we have a list of size  $N$  whose elements are  $L_1, L_2, \dots, L_N$ .

---

**Algorithm 7****Find Largest**

---

```
1: get  $N, L_1, L_2, \dots, L_N$ 
2: set  $i = 2$ 
3: set largest =  $L_1$ 
```

---

---

```
4: set index = 1
5: while  $i \leq N$  do
6:   if ( $L_i > \text{largest}$ ) then
7:     set largest =  $L_i$ 
8:     set index =  $i$ 
9:   set  $i = i + 1$ 
10: print largest
11: stop
```

---

## 1.6 The Swap Algorithm

September 25, 2021

Assume we have two variables  $x$  and  $y$  and want to swap them. Let us say,  $x = 5$  and  $y = 5$ . After the swap we want  $x = 3$  and  $y = 5$ .

To do this, we could try simply setting  $x = y$  and then  $y = x$ . Doing this would simply make  $x = y = 3$ . Thus, we need a temporary variable to store the value of  $x$  in during the swap.

---

### Algorithm 8

Swap( $x, y$ )

---

**Require:**  $x, y$

```
1: set temp =  $y$ 
2: set  $y = x$ 
3: set  $x = \text{temp}$ 
```

---

## 1.7 The Gauss Sum

September 25, 2021

Gauss was a mathematician who came up with a method to do the following. Let's assume  $n > 1$ . We want to find the result of the sum  $= 1 + 2 + 3 + \dots + n$ . For  $n = 5$ , then sum  $= 1 + 2 + 3 + 4 + 5 = 15$ .

Gauss, however, noticed that sum  $= \frac{n(n+1)}{2}$ . For  $n = 5$ , sum  $= \frac{5(5+1)}{2} = 15$ . This makes the algorithm much faster.

## 1.8 Algorithms Efficiency (Complexity)

An algorithm is **efficient** if it uses the smallest number of steps to solve the problem. What we are really interested in is how well the algorithm *scales* with large datasets.

For example, assume the size of the data is  $n$  and two correct algorithms.



---

**Algorithm 9****Gauss Sum**

---

```
1: get  $n$ 
2: set  $\text{sum} = 0$ 
3: set  $i = 1$ 
4: while ( $i \leq n$ ) do
5:   set  $\text{sum} = \text{sum} + i$ 
6:   set  $i = i + 1$ 
7: print  $\text{sum}$ 
8: stop
```

---

Algorithm 1 is 20 steps.

Algorithm 2 is 15 steps.

Which is the best algorithm? You might think that the best algorithm is the one that uses 15 steps. You are probably wrong.

Why? When we refer to “steps” in the algorithmic efficiency definition, we really mean the number of CPU operations not the number of steps written in the algorithm. We could have a loop that causes the algorithm to run much longer than one without a loop. So the number of steps in the written algorithm is not a good measure of efficiency.

Assume we have some arbitrary algorithm for a dataset of size  $n$  with a loop and some sequential steps.

- $n$  small —————> very large  $n$
- Sequential statement —————> Cost is the same
- Conditional statement —————> Cost is the same
- Iterative statement —————> Cost increases

Thus, **efficiency** is related to the loops in the algorithm. When we say **efficiency analysis**, we really mean *loop analysis*.

**1.8.1 Case Study: Sequential Search**

October 5, 2021

We will analyze the efficiency of the [Sequential Search Algorithm](#).

**Best case scenario:** The algorithm will stop immediately. The number of iterations is 1.

**Worst case scenario:** The algorithm will go through all elements before stopping. The number of iterations is  $n$ .

**Average case scenario:** The target has an equal likelihood to be in any location (1, 2, 3, 4, ...,  $n$ ). The average is therefore  $\frac{1+2+3+4+\dots+n}{n}$ . This is a Gauss sum which equals  $\frac{(n+1)n}{2n} = \frac{n+1}{2} = \frac{n}{2} + \frac{1}{2}$ . Thus, the average case scenario has a number of iterations  $\frac{n+1}{2}$ .

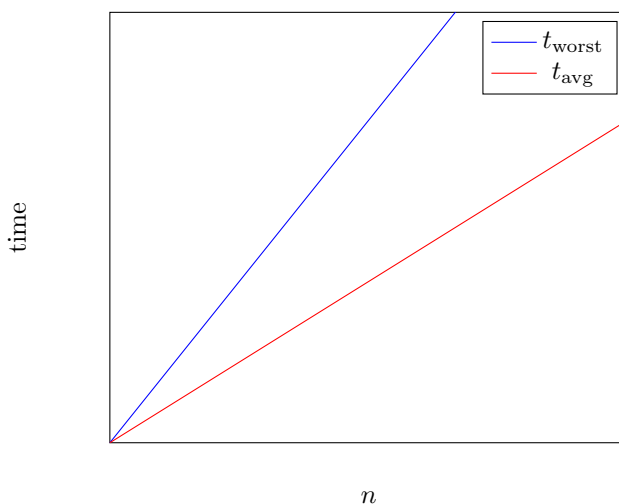
We will now do a **time analysis**.

Assume that the time required for 1 iteration is  $C$  (depending on the computer).

**Worst case scenario:**  $t_{\text{worst}} = Cn$

**Average case scenario:**  $t_{\text{avg}} = C(\frac{n}{2} + \frac{1}{2}) = \frac{1}{2}Cn + \frac{1}{2}C \approx \frac{1}{2}Cn$  for large  $n$ .

We can interpret these results in the following graph.



Note that the relationship between  $n$  and time is **linear**. That is to say, “if you scale by a certain factor in the dataset size, you scale by that same factor in time.” We say that it such an algorithm *scales linearly*. These are very efficient algorithms.

The **complexity** of this algorithm is  $\Theta(n)$ . That means linear scalability.

## 1.9 The Selection Sort Algorithm

October 5, 2021

Given a list of  $n$  elements, we need to sort the list from smallest to largest.

For example, if the list = 5, 7, 2, 8, 3, then the sorted list should = 2, 3, 5, 7, 8. Here  $n = 5$ .

We want to search the entire algorithm for the largest value (here, it is 8). We also start with marker = 5 (denoted by the red vertical line). Our list is:

5	7	2	8	3
---	---	---	---	---

Then, we swap the largest element with the last element (here, we swap 8 and 3). Now marker = 4.

5	7	2	3	8
---	---	---	---	---

These steps repeat until marker = 1. So the next few iterations would make the list look like the following.

5	3	2	7	8
---	---	---	---	---

2	3	5	7	8
---	---	---	---	---

2	3	5	7	8
---	---	---	---	---

Now, the list is sorted. We can write the algorithm.

---

### Algorithm 10

Selection Sort

---

```
1: get  $n, L_1, \dots, L_n$ 
2: set Marker =  $n$ 
3: while (Marker > 1) do
4:   set largest = FindLargest( $L_1, \dots, L_{\text{Marker}}$ )
5:   Swap(largest,  $L_{\text{Marker}}$ )
6:   set Marker = Marker - 1
7: stop
```

---

### 1.9.1 Complexity Analysis

Note that there is actually not only one loop in the algorithm, since FindLargest contains a hidden loop. This is called nested loops.

In FindLargest, we need to go through the whole list exactly once. There is no best or worst case scenario. *The number of iterations will always be equal to the size of the list.*

Similarly, in the selection sort algorithm loop, there will always be  $n$  iterations. The size of the list that is given to FindLargest decreases (not always  $n$ ).

So the total number of iterations of the selection sort algorithm is

$$n + (n - 1) + (n - 2) + \dots + 3 + 2 + 1$$

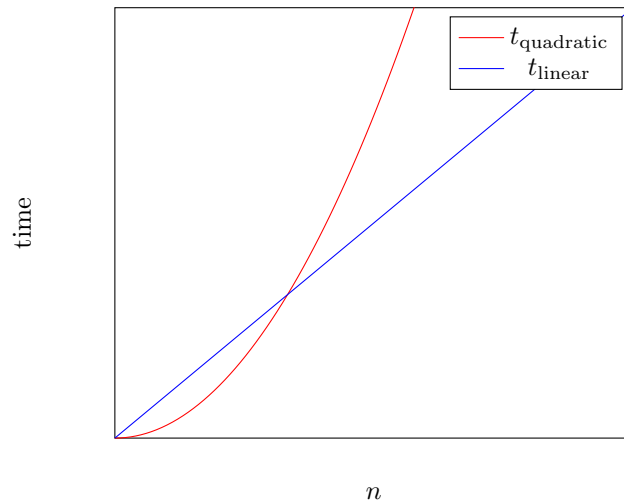
This is once again a Gauss sum. So the number of iterations required for selection sort is  $\frac{n(n+1)}{2}$ . For very large  $n$ ,

$$\text{num of iterations} = \frac{1}{2}n^2 + \frac{1}{2}n \approx \frac{1}{2}n^2$$

Therefore, the execution time is

$$\frac{1}{2}Cn^2$$

We say that this algorithm scales **quadratically**. Plotting this in a graph, we can interpret this result and comparing it to a linearly scaling algorithm.



We can see that for small values of  $n$  quadratic may be better than linear. However, with large values of  $n$ , a linear algorithm is much better than a quadratic one.

This has a complexity of  $\Theta(n^2)$ . This means that “if we double the size of  $n$ , we quadruple the time that the algorithm takes to complete.”

## 1.10 The Binary Search Algorithm

October 5, 2021

The objective of this algorithm is to search for an item in a list.

The *sequential search* is one way to do this. Recall that it has a complexity of  $\Theta(n)$ .

*Can we make the search faster?*

The answer is **yes, but the list must be sorted**. If the list is sorted, then we can use the **binary search** (also called the *half interval search*).

For example, we will use the following list with  $n = 7$ .

5	16	25	32	38	57	58
↑			↑			↑

And let our be target = 57.

We will define  $\text{Start} = 1$  and  $\text{End} = 7$  and we will also define  $\text{Mid} = \frac{\text{Start} + \text{End}}{2} = \frac{1+7}{2} = 4$ . The value at the index  $\text{Start}$  is denoted by the green colored arrow,  $\text{End}$  by the red colored arrow, and  $\text{Mid}$  by the blue colored arrow.

We then get the value at mid and check if it is less than target. It is, therefore, we set  $\text{start} = \text{mid} + 1 = 5$ . And we recalculate  $\text{mid} = \frac{\text{Start} + \text{End}}{2} = \frac{5+7}{2} = 6$ . Our new list is

5	16	25	32	38	57	58
				↑	↑	↑

Now, we get the value at mid which is 57. We check if it equals our target and it does, therefore, we say that our target is found (and at the index mid). We stop the algorithm there.

For this particular setup, it took 2 iterations. This is much faster than the 6 steps that it would have taken to do with sequential search.

Does this mean that the binary search is better than sequential search? Well, for finding a target it is. However, there is some time that is taken when initially sorting the list in the first place. If fast storage of data is required, perhaps the binary search is not the best option.

We will build now the algorithm.

---

#### Algorithm 11

Binary Search

---

```

1: get  $n, L_1, \dots, L_n$ 
2: get Target
3: set Found = No
4: set Start, End = 1, N
5: while (Found = No AND Start  $\neq$  End) do
6:   set Mid =  $\frac{\text{Start} + \text{End}}{2}$ 
7:   if ( $L_{\text{Mid}} = \text{Target}$ ) then
8:     set Found = Yes
9:   else
10:    if (Target <  $L_{\text{Mid}}$ ) then
11:      set End = Mid - 1
12:    else
13:      set Start = Mid + 1
14: if (Found = Yes) then
15:   print "Target found"
16: else
17:   print "Target not found"
18: stop

```

---

#### 1.10.1 Complexity Analysis

**Best case scenario:** The target is found in the middle. The number of iterations is 1.

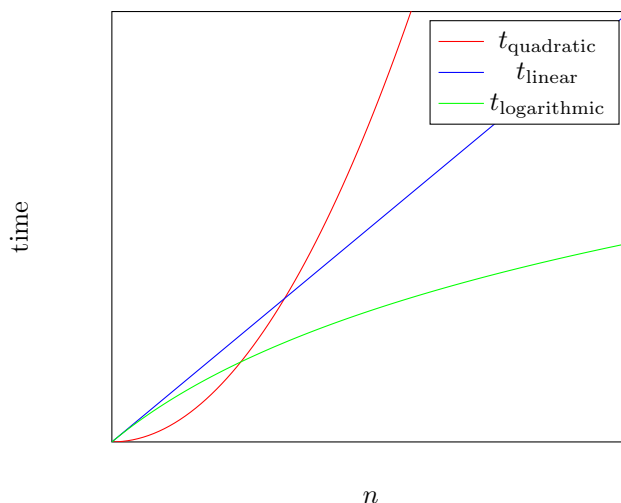
**Worst case scenario:** This happens when the target is not in the list. To do the analysis we will look at varying data sizes  $n$ :

- $n = 4 = 2^2 \longrightarrow 3$  iterations
- $n = 8 = 2^3 \longrightarrow 4$  iterations
- $n = 16 = 2^4 \longrightarrow 5$  iterations
- ...
- $n = 4 = 2^m \longrightarrow m + 1$  iterations

So,  $n = 2^m \implies m = \log_2 n$ . Thus, the number of iterations is  $\log_2 n + 1$ . This means that this algorithm scales **logarithmically** with  $n$ .

This means that the complexity of the algorithm is  $\Theta(\log_2 n)$ . This is much more efficient than linear scaling.

Plotting quadratic, linear, and logarithmic scaling,



We can see that logarithmic is the best scaling of the three, linear is worse, and quadratic is much worse.

## 1.11 Data Cleanup Algorithms

October 5, 2021

The objective of these algorithms is to clean a dataset by removing undesirable items.

For example, you are making a survey about age. You ask 10 people ( $n = 10$ ) and you get the following answers where 0 indicates missing data.

0	24	16	0	36	42	23	21	0	27
1	2	3	4	5	6	7	8	9	10

Let's say you want to find the average. If you take it directly, you would get that the average =  $\frac{0+24+16+\dots+27}{10} = 18.9$  years. This is clearly not correct.

If instead you clean the data or account for the bad values, then you would get that the average =  $\frac{24+16+\dots+27}{7} = 27$  years.

We will look at three data clean up algorithms:

- The Shuffle Left
- The Copy-Over
- The Converging Pointers

### 1.11.1 The Shuffle Left Algorithm

The idea behind this algorithm is that it will

- search the list from left to right
- If a zero is found, shuffle list to left
- It will also return the number of legitimate values in the list.

You can imagine a school bus with students spread across the bus and the bus driver wants to get all students to move to the front end. He could start at the front of the bus and move down until he finds an empty seat. Once he finds an empty seat, he tells all students to shuffle forward. He can then look for the next empty seat, and repeat this until all the empty seats are in the back end of the bus.

For example, if we have the following list ( $n = 6$ )

24	0	49	0	27	18
	↑	↑			

Let us define a pointer called Left (denoted in green) which indicates the index searching for zeros, and a pointer called Right (denoted in red) indicating position shuffle. We will also let Legit be the number of legit (here, non-zero) values. Initially, Left = 1, Right = 2, and Legit =  $n = 6$ .

Now, we check if the value at position Left (green arrow) is legit, if it is, we increment Left and Right, so the list becomes:

24	0	49	0	27	18
	↑	↑			



Now, we check if the value at position Left (green arrow) is legit, if it isn't we shuffle all values on the right of Left to the left and Legit is decremented. The list is now

24	49	0	27	18	18
	↑	↑			

with Legit = 5. Note that the last value is duplicated. Also note that the Right pointer is used to do the shuffling. This process repeats.

24	49	0	27	18	18
	↑	↑			

24	49	27	18	18	18
	↑	↑			

And now Legit = 4.

24	49	27	18	18	18
		↑	↑		

Since Left = Legit, the algorithm halts.

Let's build the algorithm.

---

**Algorithm 12** Shuffle Left

---

```

1: get  $n, L_1, \dots, L_n$ 
2: set Legit =  $n$ 
3: set Left, Right = 1, 2
4: while (Left  $\leq$  Legit) do step 5 to step 11
5:   if ( $L_{\text{Left}} \neq 0$ ) then
6:     set Left = Left + 1
7:     set Right = Right + 1
8:   else
9:     while (Right  $\leq$  Legit) do step 8 to 9
10:    set  $L_{\text{Right}-1} = L_{\text{Right}}$ 
11:    set Right = Right + 1
12: set Right = Left + 1
13: set Legit = Legit - 1
14: stop

```

---

We will now do the complexity analysis.

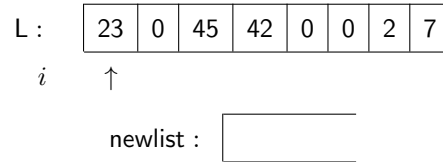
**Best case scenario:** This occurs when there are no 0s in the list, since there are no shuffles. The number of iterations in this case is  $n$ . The complexity is  $\Theta(n)$ .

**Worst case scenario:** This occurs when the list entirely consists of 0s, since there are the maximum amount shuffles. The number of iterations in this case is  $n - 1$  on the first shuffle,  $n - 2$  on the second, etc. This means that the number of iterations is  $1 + 2 + 3 + \dots + (n - 1) = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} \approx \frac{1}{2}n^2$ . The complexity is  $\Theta(n^2)$ .

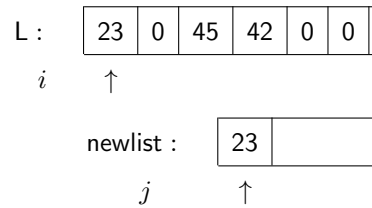
### 1.11.2 The Copy-Over Algorithm

The idea behind this algorithm is to scan the algorithm left to right and if a non-zero is found (legit), copy it into a new list.

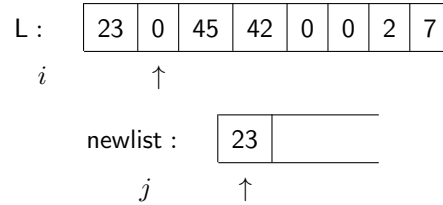
For example, we will have the list L (size  $n = 8$ ) and we will move the legit values into the list newlist (size not yet known). We will make  $i$  our pointer for iterating through L.



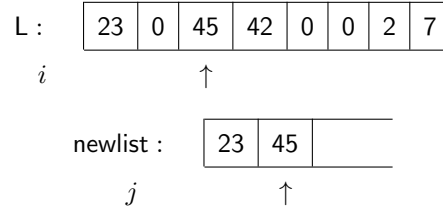
We can see that the first entry is non-zero, therefore, we put it into the newlist. We also increment  $j$  which is a pointer for newlist.



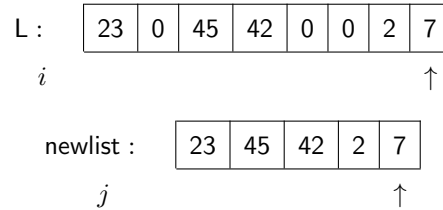
Then, we increment  $i$  and repeat.



Since  $L_i$  is 0 here, we do not put that element in newlist (and we do not increment  $j$ ). Instead, we simply increment  $i$  and repeat.



This whole process repeats until the lists and pointers look like this.



Building the algorithm, we have

---

**Algorithm 13** Copy-Over

---

```

1: get  $n, L_1, \dots, L_n$ 
2: set  $i = 1$ 
3: set  $j = 1$ 
4: while ( $i \leq n$ ) do step 5 to step 6
5:   if ( $L_i \neq 0$ ) then
       set  $newlist_j = L_i$ 
       set  $j = j + 1$ 
6:   set  $i = i + 1$ 
7: stop

```

---

Now, for the complexity analysis.

There is no best or worse case scenarios in this algorithm. There is always  $n$  iterations. That means that the **time complexity** is  $\Theta(n)$ .

Compared to the last algorithm, this is a much more efficient algorithm in time. However, the **space complexity** is worse. It requires double storage space.

### 1.11.3 The Converging Pointers Algorithm

The idea behind this algorithm is that there are two pointers, Left and Right.

- Left moves up only if it is pointing at a non-zero value.
- If Left points at zero, copy the value at Right into Left and decrement Right by 1.
- Stop when Left = Right

For example, consider the following list (with  $n = 8$ ). The blue arrow will represent the **Left** pointer and the red will represent the **Right** pointer.

0	24	16	0	23	21	0	27
---	----	----	---	----	----	---	----

↑↑

Notice that Left = 1 and Right =  $n = 8$ .

Since the value at Left is 0, we put the value at Right into Left and decrement Right by 1. The list becomes

27	24	16	0	23	21	0	27
----	----	----	---	----	----	---	----

↑↑

Now, Left moves forward until it is at a zero.

27	24	16	0	23	21	0	27
----	----	----	---	----	----	---	----

↑↑

The entry at the Right pointer gets copied into the Left pointer and Right is decremented, giving

27	24	16	0	23	21	0	27
----	----	----	---	----	----	---	----

↑↑

This process repeats until the pointers converge.

27	24	16	21	23	21	0	27
----	----	----	----	----	----	---	----

↑↑

27	24	16	21	23	21	0	27
----	----	----	----	----	----	---	----

↑↑

The number of legit values is equal to the converged pointers. All of the legit values are to the left of the converged pointers. Note that this algorithm does not preserve order.

We can now build the algorithm.

---

**Algorithm 14** Converging Pointers

---

```

1: get  $n, L_1, \dots, L_n$ 
2: set Left = 1, Right =  $n$ 
3: while (Left  $\leq$  Right) do step 4 to step 5
4:   if ( $L_{\text{Left}} \neq 0$ ) then
       set Left = Left + 1
5:   else
       set  $L_{\text{Left}} = L_{\text{Right}}$ 
       set Right = Right + 1
6: stop

```

---

We will now do the complexity analysis.

There is, again, no best or worse cases. The number of iterations is always  $n$ . Therefore, the complexity is  $\Theta(n)$ .

## 1.12 Sorting Algorithms

October 13, 2021

### 1.12.1 Selection Sort

Let's say we have the list

2	15	8	7
---	----	---	---

↑

We scan the list and find the largest (denoted by the red arrow). We also start with a marker (red line) equal to the length of the list. Then, we swap the element at that index with the last element and decrement the marker. Then, we repeat on all elements before the marker.

2	7	8	15
---	---	---	----

↑

2	7	8	15
---	---	---	----

↑

2	7	8	15
---	---	---	----

↑

2	7	8	15
---	---	---	----

The list is now sorted. The complexity of this algorithm is  $\Theta(n^2)$ . See [Section 1.9](#) for more details.

### 1.12.2 Bubble Sort

Let's say we have the following list of size  $n = 8$ .

34	7	19	35	2	8	10	22
----	---	----	----	---	---	----	----

↑   ↑

We start by comparing the first two elements with each other. If the element at the right (blue) arrow is larger than the left (red) one, then they swap. The red and blue counter advances.

7	34	19	35	2	8	10	22
	↑	↑					

This process repeats until the arrows reach the end of the list...

7	19	34	2	8	10	22	35
						↑	↑

Now, we have ensured that the largest element in the list is at its final location. We can repeat the same process on all elements that are not sorted yet (at the moment, the first 7 elements). This takes  $(n - 1)$  iterations to do. After repeated the process once, the list will become

7	19	2	8	10	22	34	35
					↑	↑	

This takes  $(n - 2)$  iterations to do. This repeats until we obtain the fully sorted list

2	7	8	10	19	22	34	35
	↑	↑					

The total number of iterations this algorithm will take is  $(n-1)+(n-2)+\dots+1 = \frac{n(n-1)}{2}$ . This is a Gauss sum (minus  $n$ ), so the complexity will be  $\Theta(n^2)$  in the **worst case**.

In the **best case**, if we include a Boolean variable to the algorithm called “sorted” which is initially true and switches to false if a swap occurs. In that case, if the list is initially fully sorted, then the number of iterations is  $n - 1$  and the complexity is  $\Theta(n)$ .

### 1.12.3 Quick Sort

This algorithm uses a divide and conquer strategy.

Such a strategy works by the following steps:

- Define a pivot
- Rearrange the list such that all elements that are less than the pivot are before it and all that are greater than the pivot come after it
- We now know that the pivot is in the right location. We then apply this strategy on the left and right sub-lists.

Using the following list,

34	2	15	7	33	9	20	4
----	---	----	---	----	---	----	---

Where do we put the pivot?

- Last element in the list
- First element in the list
- Median of sub-list

For our implementation, we will consider the first of these: Where the pivot (denoted by the red arrow) is initially at the last element in the list.

34	2	15	7	33	9	20	4
----	---	----	---	----	---	----	---

↑

Then, elements that are less than the pivot go to the left of the pivot, and elements that are greater than the pivot go to the right.

2	4	34	15	7	33	9	20
---	---	----	----	---	----	---	----

↑

We know now that the pivot is in the correct position. We end up with two sub-lists now. Since the left sub-list is of size 1, we leave it as it is. And now we pick a pivot for the right sub-list (green boxes). We choose the rightmost element (20) and proceed with the algorithm.

2	4	34	15	7	33	9	20
---	---	----	----	---	----	---	----

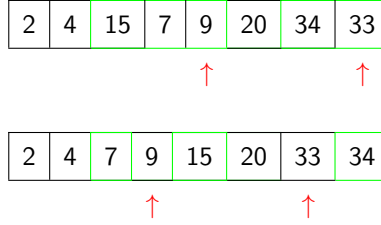
↑

2	4	15	7	9	20	34	33
---	---	----	---	---	----	----	----

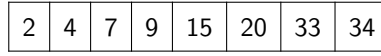
↑



Now we repeat this on the new sub-lists.



Since all the sub-lists are of size one, the algorithm is finished and the sorted list is



As for the complexity, the **worse case** will happen when each pivot that is selected fails to divide the list. In this case, the complexity is  $\Theta(n^2)$ .

In the **best case**, we divide each list in half. In this case, the complexity is  $\Theta(n \log_2 n)$ .

## 2 Hardware

### 2.1 Binary Numbering Systems

#### 2.1.1 Introduction and Definitions

October 13, 2021

A **positional numbering system** of base  $b$  and is given by

- A natural number for the base  $b$  (e.g. 2, 3, 4, 5, ...)
- A set of simple digits that contain  $b$  digits (e.g. digits 0, 1, 2, ...,  $(b - 1)$ )
- Position is used to determine the power of  $b$  by which the digit is multiplied

A number in that numbering system is written as  $(A)_b = A_m A_{m-1} \dots A_1 A_0 . A_{-1} A_{-2} \dots A_{-n}$ . The part in front of the fractional dot is the *integer part* and the part after is the *fractional part*.

This is equivalent to  $(A)_b = A_m \times b^m + A_{m-1} \times b^{m-1} + \dots + A_1 \times b^1 + A_0 \times b^0 + A_{-1} \times b^{-1} + A_{-2} \times b^{-2} + \dots + A_{-n} \times b^{-n}$ .

If no base is specified, we assume that the base is base 10.

For example,  $(4327)_{10} = 4 \times 10^3 + 3 \times 10^2 + 2 \times 10^1 + 7 \times 10^0 = 4327$ .  
 Similarly,  $(3.25)_{10} = 3 \times 10^0 + 2 \times 10^{-1} + 5 \times 10^{-2} = 3.25$ .

Let's say now we have  $(4021)_5$ . By the formula above,

$$\begin{aligned}(4021)_5 &= 4 \times 5^3 + 0 \times 5^2 + 2 \times 5^1 + 1 \times 5^0 \\ &= 4 \times 125 + 0 \times 25 + 2 \times 5 + 1 \times 1 \\ &= (511)_{10}\end{aligned}$$

Another example,

$$\begin{aligned}(127.4)_5 &= 1 \times 5^2 + 2 \times 5^1 + 7 \times 5^0 + 4 \times 5^{-1} \\ &= 1 \times 25 + 2 \times 5 + 7 \times 1 + \frac{4}{5} \\ &= (87.5)_{10}\end{aligned}$$

In we write  $(521)_4$ , this is incorrect. This is because there is no digit 5 in base 4 (the only allowed symbols are 0, 1, 2, and 3).

The most frequent bases are decimal (base 10), binary (base 2), octal (base 8), and hexadecimal (base 16). Note that both octal and hexadecimal have bases that are powers of two ( $8 = 2^3$  and  $16 = 2^4$ ), so it is convenient to use for computers (since they run on binary).

For a binary example,

$$\begin{aligned}(11011.01)_2 &= 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} \\ &= 1 \times 16 + 1 \times 8 + 0 \times 4 + 1 \times 2 + 1 \times 1 + \frac{0}{2} + \frac{1}{4} \\ &= (27.25)_{10}\end{aligned}$$

In binary, the allowed symbols are 0 and 1.

In octal, the allowed symbols are 0,1,2,3,4,5,6,7.

In hexadecimal, the allowed symbols are 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F.

For a hexadecimal example,

$$\begin{aligned}(\text{B657})_{16} &= 11 \times 16^3 + 6 \times 16^2 + 5 \times 16^1 + 7 \times 16^0 \\ &= (46687)_{10}\end{aligned}$$

Notice that hexadecimal is much more compact than in decimal or binary.

### 2.1.2 Base conversion

October 13, 2021

We have already converted from an arbitrary base to base 10. But how do we convert from base 10 to another base?

Starting with the *integer part*, we will apply successive *divisions* by the base and accumulate the remainders.

For example, if we want to convert  $(53)_{10}$  to binary ( $b = 2$ ), we divide by 2 to get 26 with a remainder of 1. Then, we divide again by 2 to get 13 with a remainder of 0, again to get 6 with remainder 1, again to get 3 with a remainder of 0, again to get 1 with a remainder of 1 and once more to get 0 with a remainder of 1. When we reach 0, we stop. Now, we accumulate the remainders (going **backwards**) to get that  $(53)_{10} = (110101)_2$ .

We can apply the same method to convert  $(53)_{10}$  to octal, giving that  $(53)_{10} = (65)_8$ .

For the *fractional part*, we successively *multiply* by the base and accumulate the integer parts.

For example, if we want to convert  $(0.6875)_{10}$  to binary ( $b = 2$ ), we multiply by 2 to get 1.375. We multiply the fractional part of the result by 2 again to get 0.75, again to get 1.5, and again to get 1.0. Since the fractional part is now 0, we stop. Now, we accumulate the integer parts (going **forwards**) to get that  $(0.6875)_{10} = (0.1011)_2$ .

### 2.1.3 Octal & Hexadecimal systems

October 13, 2021

The following table shows the conversions between octal and binary.

Octal	Binary
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Using this table, we can easily convert, for example,  $(562)_8$  to binary. It is simply  $(562)_8 = (101\ 110\ 010)_2$ . Similarly, we know that  $(110\ 011\ 010\ 001)_2 = (6321)_8$ .

We can use the same strategy for hexadecimal.

Hexadecimal	Binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

Using this table, we can easily convert, for example,  $(5AB1)_{16}$  to binary. It is simply  $(5AB1)_{16} = (0101\ 1010\ 1011\ 0001)_2$ . Similarly, we know that  $(1101\ 0111\ 1111)_2 = (D7F)_{16}$ .

We use hexadecimal and octal because they are compact, and because conversion to/from binary is easier.

Hexadecimal is used to represent memory addresses.

#### 2.1.4 Binary Representation using a fixed number of bits



October 23, 2021

In a computer, we use a **fixed** number of bits.

A **bit** stands for Binary Element. It is a binary digit. For example, 11 uses 2 bits, and 01101 uses 5 bits. A bit can be 0 or 1 and is defined by hardware.

Computer systems use a fixed number of bits to represent data. Most people use 64 bit computers nowadays.

For example, if we have an 8 bit computer system, then we can convert the decimals to binary as follows.

$(3)_{10}$	0	0	0	0	0	0	1	1
$(15)_{10}$	0	0	0	0	1	1	1	1
$(255)_{10}$	1	1	1	1	1	1	1	1
								

We call the leftmost bit (blue arrow) the Most Significant Bit (MSB), and the rightmost bit (red arrow) is called the Least Significant Bit (LSB).

For 8 bits, the smallest integer is  $(0)_{10} = (0000\ 0000)_2$  and the largest integer is  $(255)_{10} = (1111\ 1111)_2 = (256 - 1)_{10} = (2^8 - 1)_{10}$ .

In general for  $n$  bits, the smallest integer is  $(0)_{10}$  and the largest integer is  $(2^n - 1)_{10}$ . This gives  $2^n$  available integers.

## 2.2 Representation of Signed Integers

**Signed integers** are integers that have the possibility of being negative. For example, +6, -3, +51, and -23 are all signed integers. A negative integer is a signed integer, but a signed integer is not necessarily a negative number. There are a couple representations for signed numbers in a computer.

### 2.2.1 Signed magnitude representation

The MSB is used for the sign (+ or -) and the rest is the magnitude. By convention, the + is represented by a 0 and the - is represented by a 1.

$(+6)_{10}$	0	0	0	0	0	1	1	0
$(-6)_{10}$	1	0	0	0	0	1	1	0
$(-127)_{10}$	1	1	1	1	1	1	1	1

The smallest number for an 8 bit computer in this representation is  $(-127)_{10} = 1111\ 1111$  and the largest is  $(+127)_{10} = 0111\ 1111$ . There are 256 representations. Note, however, that there are two zeros  $((-0)_{10} = 1000\ 0000$  and  $(+0)_{10} = 0000\ 0000)$ ... This is an issue.

In general, for an  $n$  bit computer in the signed magnitude representation, there are 2 zeros,  $2^{n-1} - 1$  negative numbers and  $2^{n-1} - 1$  positive numbers. There are  $2^n$  representations.

To show that we have a problem with this representation system, we will assume a 3-bit computer system.

Binary	(Unsigned integers) <sub>10</sub>	(Signed integers) <sub>10</sub>
000	0	+0
001	1	+1
010	2	+2
011	3	+3
100	4	-0
101	5	-1
110	6	-2
111	7	-3

The *first issue* here is that there are two zeros, but zero should not have a sign.

The *second issue* has to do with integer addition. We can write that  $5 - 2 = (+5) + (-2)$ . Thus, we can use signed integers and addition to do subtraction of any numbers. Let's say we want to do  $1 - 3 = (+1) + (-3) = (-2)$  in binary. We can write this as follows

$$\begin{array}{r} 1 \phantom{000} 1 \phantom{00} 1 \\ \phantom{0} 0 \phantom{00} 0 \phantom{000} 1 \\ + \phantom{0} 1 \phantom{00} 1 \phantom{000} 1 \\ \hline 1 \phantom{00} 0 \phantom{00} 0 \phantom{000} 0 \end{array}$$

Typically, a computer will cut off the leftmost bit to make it 3 bits long to give an answer of  $(000)_2 = (+0)_{10}$ . In principle, we should get  $(-2)_{10} = (110)_2$ , which is not what we get. There is our second issue with this representation: signed magnitude does not allow arithmetic operations.

We need to use another representation to fix these problems.

### 2.2.2 The 2's complement

First we will define the **1's complement**: The 1's complement of a binary number is obtained by complementing (taking the opposite) of each binary digit in the number. For example, the 1's complement of 0 is 1, of 1 is 0, of 001 is 110, of 11011 is 00100.

Note that if you add a binary number to it's 1's complement, you will always get a number of the same size, consisting only of 1s.

The **2's complement** is found by first taking the 1's complement and then adding 1. For example, the 2's complement of 1011 is  $0100 + 1 = 0101$ , and of 110101 is  $001010 + 1 = 001011$ .

Note that if you add a binary number to its 2's complement, you will always get a number of the size, consisting of 1s, plus another bit to the left that is 1 (carry). If we cut off the carry, then we get 0. Therefore, the opposite sign can be given by the 2's complement (since the sum is 0).

Thus, we conclude that we can use the 2's complement represents signed numbers. If you take the 2's complement on a positive number, it will give you the equivalent negative number and vice versa.

### 2.2.3 2's complement representation

The following is a conversion table comparing this representation to the signed magnitude representation.

Binary	(Unsigned integers) <sub>10</sub>	(Signed magnitude) <sub>10</sub>	(2's complement) <sub>10</sub>
000	0	+0	+0
001	1	+1	+1
010	2	+2	+2
011	3	+3	+3
100	4	-0	-4
101	5	-1	-3
110	6	-2	-2
111	7	-3	-1

With the 2's complement representation, we solve the double zero issue. Now, we will check if arithmetic operations work in this representation. We will add  $(+1) + (-3)$ .

$$\begin{array}{r} \phantom{+} \phantom{0} \phantom{0} \phantom{1} \\ \phantom{+} \phantom{0} \phantom{0} \phantom{1} \\ + \phantom{0} \phantom{0} \phantom{0} \phantom{1} \\ \hline \phantom{0} \phantom{0} \phantom{0} \phantom{1} \end{array}$$

We indeed do get  $(110)_2 = (-2)_{10}$  as expected.

#### 2.2.4 Comprehensive example and other representations

Consider an 8 bit system. Assume that the following is stored in memory:  $A = 10100110$ .

What is  $A$ ?

We need a representation to answer this question. We will compute it for each of the representations we know (unsigned, signed magnitude, and 2's complement representations).

**Unsigned:**  $(2 + 4 + 32 + 128)_{10} = (166)_{10}$

**Signed magnitude:**  $-(2 + 4 + 32)_{10} = (-38)_{10}$

**2's complement:**  $-(01011001 + 1)_2 = -(01011010)_2 = -(2 + 8 + 16 + 64)_{10} = (-92)_{10}$

Your program decides which representation to use.

We can also use a number to represent a pixel on an image. If we use gray scale in an 8-bit computer system, we can represent the different shades by binary numbers. By convention, we let a black pixel be  $(0)_{10} = (0000\ 0000)_2$  and a white pixel be  $(255)_{10} = (1111\ 1111)_2$ . Therefore, for an 8-bit system, there are 256 possible shades. If we want to represent color, we use 3 binary numbers that represent the red, green and blue parts of the color (RGB).

For text, a common encoding (representation) is ASCII code. There is a table that converts an 8-bit binary number to an ASCII character.

### 2.3 Boolean Logic

**Boolean logic** is a branch of mathematics developed by George Boole. Boolean logic manipulates *Boolean data* using *Boolean operators*.

A **Boolean variable** takes only two possible values  $\in \{\text{True}, \text{False}\}$  ( $\{1, 0\}$  respectively).

A **Boolean (logic) operators** take a Boolean as input and result in a Boolean. The basic operators are AND, OR, and NOT.



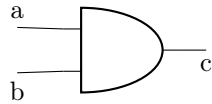
### 2.3.1 AND operator

Let  $a$ ,  $b$ ,  $c$  be Boolean variables. We define  $c = a \text{ AND } b = a \cdot b = ab$ . The following is the **truth table** for this operation.

a	b	$c = ab$
0	0	0
0	1	0
1	0	0
1	1	1

Sometimes the AND operation is called the *min operator* because it takes the minimum of the two values.

A **logic gate** is an electronic circuit that implements a Boolean operation. The AND gate is represented as follows:



In a logic gate, logic 1 corresponds to 5 Volts and logic 0 corresponds to 0 Volts.

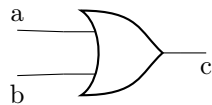
### 2.3.2 OR operator

Let  $a$ ,  $b$ ,  $c$  be Boolean variables. We define  $c = a \text{ OR } b = a + b$ . The following is the **truth table** for this operation.

a	b	$c = a + b$
0	0	0
0	1	1
1	0	1
1	1	1

Sometimes the OR operation is called the *max operator* because it takes the maximum of the two values.

The OR logic gate is represented as follows:

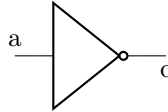


### 2.3.3 NOT operator

Let  $a, c$  be Boolean variables. We define  $c = \text{NOT } a = \bar{a} = a'$ . The following is the **truth table** for this operation.

$a$	$c = \bar{a}$
0	1
1	0

The OR logic gate is represented as follows:



### 2.3.4 Properties of Operators

The AND and OR operators are associative:  
 $(x \cdot y) \cdot z = x \cdot (y \cdot z)$  and  $(x + y) + z = x + (y + z)$ .

The AND and OR operators are commutative:  
 $x \cdot y = y \cdot x$  and  $x + y = y + x$ .

The AND and OR operators are distributive:  
 $x \cdot (y + z) = x \cdot y + x \cdot z$

The AND and OR operators have identity and zero values:  
 $x \cdot 1 = x$  and  $x \cdot 0 = 0$   
 $x + 0 = x$  and  $x + 1 = 1$   
 $x \cdot x' = 0$  and  $x + x' = 1$

### 2.3.5 De Morgan Theorem

De Morgan's Theorem states that

$$(x + y)' = x' \cdot y'$$
$$(x \cdot y)' = x' + y'$$

We will use truth tables to proof this theorem.

Starting with the  $(x + y)' = x' \cdot y'$  case:

x	y	$(x+y)$	$(x+y)'$	$x'$	$y'$	$x' \cdot y'$
0	0	0	1	1	1	1
0	1	1	0	1	0	0
1	0	1	0	0	1	0
1	1	1	0	0	0	0

Notice that the  $(x+y)'$  and  $x' \cdot y'$  columns are identical.

Now for the  $(x \cdot y)' = x' + y'$  case:

x	y	$(x \cdot y)$	$(x \cdot y)'$	$x'$	$y'$	$x' + y'$
0	0	0	1	1	1	1
0	1	0	1	1	0	1
1	0	0	1	0	1	1
1	1	1	0	0	0	0

Notice that the  $(x \cdot y)'$  and  $x' + y'$  columns are identical.

Thus, the de Morgan Theorem is proved.

### 2.3.6 Boolean functions

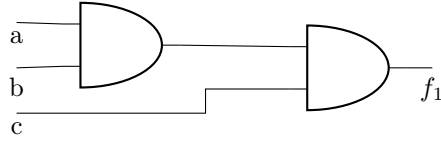
A **Boolean function** is a combination of multiple Boolean variables using multiple Boolean operators to obtain a Boolean output.

We express Boolean functions with: A Boolean expression, a truth table, a logic diagram (logic gates).

For example, let's express  $f_1 = a \cdot b \cdot c$  in three ways. It is already expressed as a Boolean expression. Let's construct a truth table:

a	b	c	$f_1$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

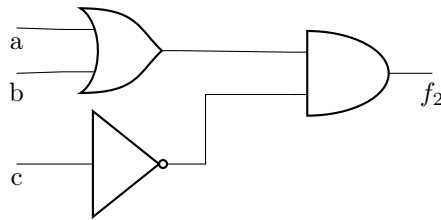
Representing this in a logic diagram, we get



A second example is with  $f_2 = (a + b) \cdot \bar{c}$ . The truth table of  $f_2$  is

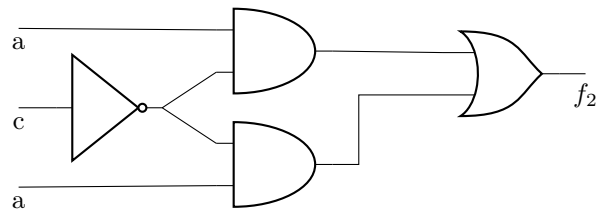
a	b	c	a+b	$f_2$
0	0	0	0	0
0	0	1	0	0
0	1	0	1	1
0	1	1	1	0
1	0	0	1	1
1	0	1	1	0
1	1	0	1	1
1	1	1	1	0

The logic diagram is



Question 1: Is the Boolean expression unique for a given function? No! Since  $f_2 = (a + b) \cdot \bar{c} = a \cdot \bar{c} + b \cdot \bar{c}$ .

Question 2: Is the diagram unique? No! As above, you can make the diagram be such that  $f_2 = a \cdot \bar{c} + b \cdot \bar{c}$ . Namely,

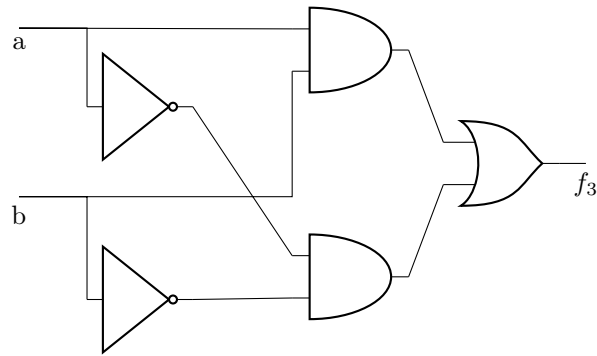


Question 3: Is the truth table unique? Yes! Always, since we use the truth table to define when two Boolean functions are equal.

A third example could be  $f_3 = ab + \bar{a}\bar{b}$ . The following is the truth table.

a	b	ab	$\bar{a}\bar{b}$	$f_2$
0	0	0	1	1
0	1	0	0	0
1	0	0	0	0
1	1	1	0	1

The logic diagram is



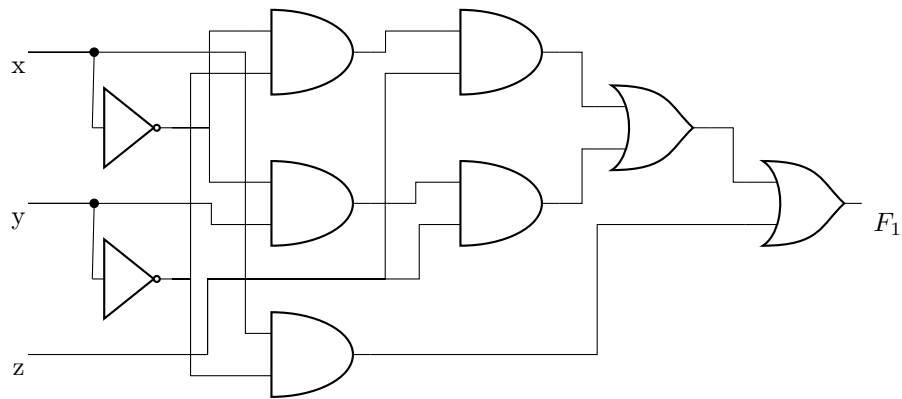
In a fourth example, we will consider  $F_1 = x'y'z + x'yz + xy'$  and  $F_2 = x'z + xy'$ . We will prove that they are identical and show each of their logic diagrams.

We can show that  $F_1$  and  $F_2$  are identical by checking their truth table results, however we will show it using Boolean properties.

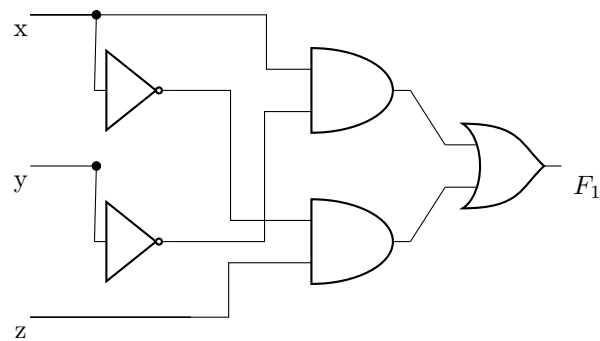
Since  $F_1$  has more terms than  $F_2$ , we will try to convert  $F_1$  to  $F_2$ .

$$\begin{aligned}
 F_1 &= x'y'z + x'yz + xy' \\
 &= x'z(y' + y) + xy' \\
 &= x'z(1) + xy' \\
 &= x'z + xy' \\
 &= F_2
 \end{aligned}$$

Now we will show the logic diagram of  $F_1$ .



And the logic diagram of  $F_2$  is



The following are a few examples of simplifying Boolean expressions:

$$\begin{aligned}
 x(x' + y) &= xx' + xy \\
 &= 0 + xy \\
 &= xy
 \end{aligned}$$

$$\begin{aligned}
 (x + y)(x + y') &= xx + xy' + xy + yy' \\
 &= x + x(y' + y) + 0 \\
 &= x + x(1) \\
 &= x + x \\
 &= x
 \end{aligned}$$

$$\begin{aligned}
((x + y)(x + y'))' &= (x + y)' + (x + y')' \\
&= x'y' + x'y \\
&= x'(y' + y) \\
&= x'(1) \\
&= x'
\end{aligned}$$

### 2.3.7 Boolean Expression from Truth Table

So far, we have been given a Boolean expression and have been told to generate a truth table and implement it in a logic diagram. If instead, we start with the logic diagram, it is not hard to find the Boolean expression and the truth table. However, if we are given a truth table, it is not as straightforward. If we can get the Boolean Expression from the truth table, getting the logic diagram is easy. So the question is: *How do we get a Boolean expression from a truth table?*

Let's assume a function  $f(x, y)$  that has the truth table

a	b	$f$
0	0	0
0	1	0
1	0	1
1	1	1

We know that  $f = 1$  when  $(x = 1 \text{ AND } y = 0)$  OR when  $(x = 1 \text{ AND } y = 1)$ . Simplifying,  $f = 1$  when  $x\bar{y} + xy$ . Checking the truth table, this works for all values. Thus, the function is  $f = x(\bar{y} + y) = x(1) = x$ .

Let's now assume a function  $f_2(x, y)$  that has the truth table

a	b	$f_2$
0	0	0
0	1	1
1	0	1
1	1	0

We will look at all values of  $f = 1$  and find the combination of  $x$  and  $y$  that make  $f = 1$ . In this case, we know that  $f = 1$  when  $(x = 0 \text{ AND } y = 1)$  OR when  $(x = 1 \text{ AND } y = 0)$ . Simplifying,  $f = 1$  when  $\bar{x} + x\bar{y}$ .

We say that  $f$  is written as a **sum of products** (OR of ANDs). This is also called the **canonical form**.

Let's do a larger example: Express the following using the sum of products.

x	y	z	$f_3$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

For the first time  $f = 1$ , we get  $\bar{x}y\bar{z}$ , then  $\bar{x}yz$ , and so on. In the end, we get that  $f = \bar{x}y\bar{z} + \bar{x}yz + xy\bar{z} + xyz$ . This simplifies to just  $f = y$ .