Section 2

Introduction to Computer Science

These notes are from CS201, the introduction course in computer science at Bishop's University. They also reference *Invitation to Computer Science* (8^{th} edition) by G. Michael Schneider and Judith L. Gersting.

Subsection 2.1

Course Overview

2.1.1 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.

Subsection 2.2

Introduction to Algorithms

2.2.1 Design

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.

Sequential Steps

Do a single task.

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

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.

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.

2.2.2 Case Study: Addition Algorithm

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

$$\begin{array}{cccc} & {\scriptstyle 1} & & \\ & 4 & 7 & 2 \\ + & 5 & 9 & 3 \\ \hline 1 & 0 & 6 & 5 \end{array}$$

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 te 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:

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

Let's write the steps of our algorithm:

Algorithm 1
Addition Algorithm

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

2.2.3 Pseudocode

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.

Sequential statements

- **Input**: Get "variable". E.g. Get m, Get radius
- **Computation**: Set variable = expression. E.g. Set area = $\pi \times$ 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**

Conditional statements

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

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**

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).

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 (index ≤ 100) do
3:     set square = index * index
4:     print square
5:     set index = index + 1
6: stop
```

The Do-While

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

- 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 ≥ 0) then
5:         set root = √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
```

2.2.4 The Sequential Search Algorithm

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, ..., L_N and a target element called Target.

Algorithm 6
Sequential Search Algorithm

```
1: get L_1, L_2, ..., L_N, N, Target
2: set Found = No
3: set i = 1
4: while Found = No AND i <= N do
5:     if (L_i == 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
```

2.2.5 Find Largest

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, ..., L_N.

Algorithm 7
Find Largest

```
1: get N, L_1, L_2, ..., L_N
2: set i = 2
3: set largest = L_1
```

2.2.6 The Swap Algorithm

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.

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

Algorithm 8
Swap(x, y)

```
Require: x, y
1: set temp = y
2: set y = x
3: set x = temp
```

2.2.7 The Gauss Sum

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 $= 1 + 2 + 3 + ... + n$. For $n = 5$, then sum $= 1 + 2 + 3 + 4 + 5 = 15$.

Algorithm 9
Gauss Sum

```
1: get n
2: set sum = 0
3: set i = 1
4: while (i ≤ n) do
5:     set sum = sum + i
6:     set i = i + 1
7: print sum
8: stop
```

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.

2.2.8 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 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*.

Case Study: Sequential Search

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 $\frac{1+2+3+4+...+n}{n}$. 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 $\frac{n+1}{2}$.

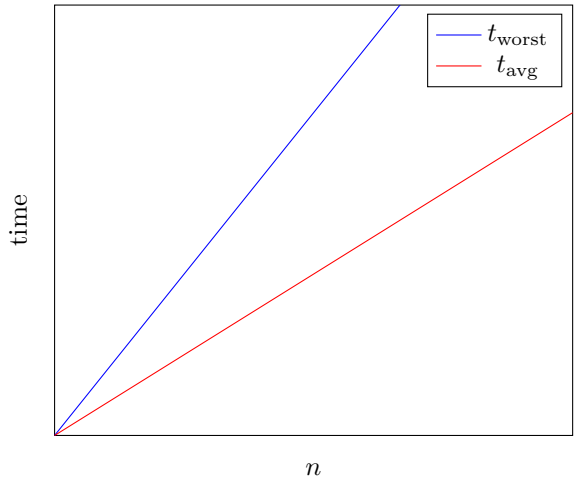
We will now do a **time analysis.**

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.

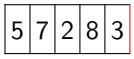
The **complexity** of this algorithm is $\Theta(n)$. That means linear scalability.

2.2.9 The Selection Sort Algorithm

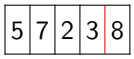
Given a list of n elements, we need to sort the list from smallest to largest.

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

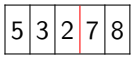
We want to search the entire algorithm for the largest value (here, it is 8). We also start with marker = 5 (denoted by the red vertical line). Our list is:

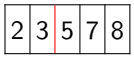


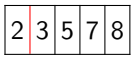
Then, we swap the largest element with the last element (here, we swap 8 and 3). Now marker = 4.



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







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

Algorithm 10
Selection Sort

```
1: get n, L_1, ..., L_n
2: set Marker = n
3: while (Marker > 1) do
4:     set largest = FindLargest(L_1, ..., L_Marker)
5:     Swap(largest, L_Marker)
6:     set Marker = Marker - 1
7: stop
```

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

$$n + (n-1) + (n-2) + ... + 3 + 2 + 1$$

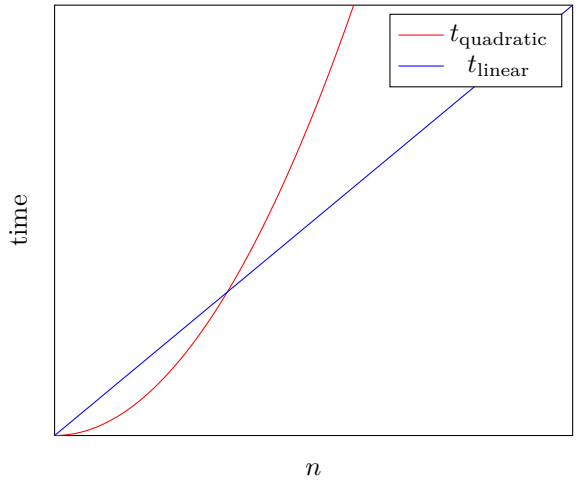
This is once again a Gauss sum. So the number of iterations required for selection sort is $\frac{n(n+1)}{2}$. For very large n,

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

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."

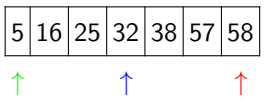
2.2.10 The Binary Search Algorithm

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?

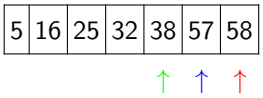
The answer is **yes, but the list must be sorted**. If the list is sorted, then we can use the **binary search** (also called the *half interval search*).
For example, we will use the following list with $n = 7$.



And let our be target = 57.

We will define Start = 1 and End = 7 and we will also define Mid = $\frac{\text{Start+End}}{2} = \frac{1+7}{2} = 4$. The value at the index Start is denoted by the green colored arrow, End by the red colored arrow, and 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 start = mid + 1 = 5. And we recalculate mid = $\frac{\text{Start+End}}{2} = \frac{5+7}{2} = 6$. Our new list is



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, ..., L_n
2:  get Target
3:  set Found = No
4:  set Start, End = 1, N
5:  while (Found = No AND Start ≠ End) do
6:      set Mid = (Start+End)/2
7:      if (L_Mid = Target) then
8:          set Found = Yes
9:      else
10:         if (Target < L_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
```

Complexity Analysis

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

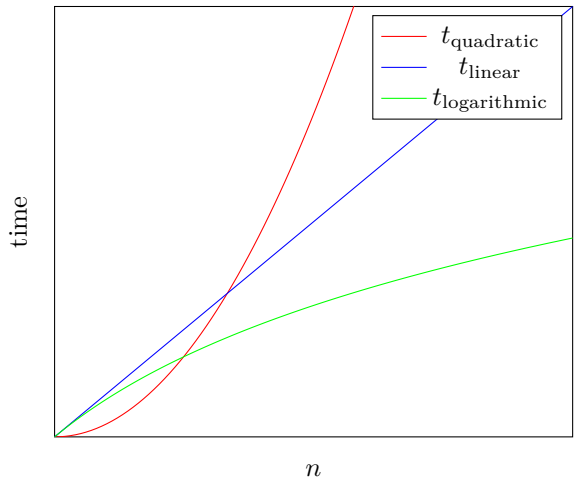
Worst case scenario: This happens when the target is not in the list. To do the analysis we will look at varying data sizes n:

- $n = 4 = 2^2$ ———> 3 iterations
- $n = 4 = 2^3$ ———> 4 iterations
- $n = 4 = 2^4$ ———> 5 iterations
- ...
- $n = 4 = 2^m$ ———> $m + 1$ 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.

2.2.11 Data Cleanup Algorithms

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 $= \frac{0+24+16+...+27}{10} = 18.9$ 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 $= \frac{24+16+...+27}{7} = 27$ years.

We will look at three data clean up algorithms:

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

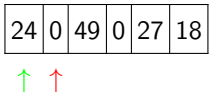
The Shuffle Left Algorithm

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.

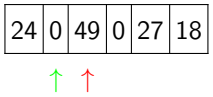
You can imagine a school bus with students spread across the bus and the bus driver wants to get all students to move to the front end. He could start at the front of the bus and move down until he finds an empty seat. Once he finds an empty seat, he tells all students to shuffle forward. He can then 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 ($n = 6$)

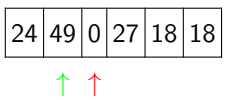


Let us define a pointer called Left (denoted in green) which indicates the index searching for zeros, and a pointer called Right (denoted in red) indicating position shuffle. We will also let Legit be the number of legit (here, non-zero) values. Initially, Left = 1, Right = 2, and Legit = $n = 6$.

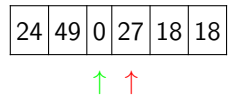
Now, we check if the value at position Left (green arrow) is legit, if it is, we increment Left and Right, so the list becomes:

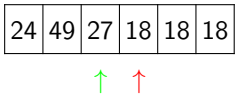


Now, we check if the value at position Left (green arrow) is legit, if it isn't we shuffle all values on the right of Left to the left and Legit is decremented. The list is now

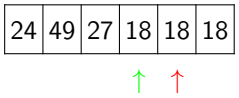


with Legit = 5. Note that the last value is duplicated. Also note that the Right pointer is used to do the shuffling. This process repeats.





And now Legit = 4.



Since Left = Legit, the algorithm halts.

Let's build the algorithm.

Algorithm 12 Shuffle Left

```
1:  get n, L_1, ..., L_n
2:  set Legit = n
3:  set Left, Right = 1, 2
4:  while (Left ≤ Legit) do step 5 to step 11
5:      if (L_Left ≠ 0) then
            set  Left = Left + 1
            set  Right = Right + 1
6:      else
7:          while (Right ≤ Legit) do step 8 to 9
8:              set L_(Right−1) = L_Right
9:              set Right = Right + 1
10:         set Right = Left + 1
11:         set Legit = Legit - 1
12: stop
```

We will now do the complexity analysis.

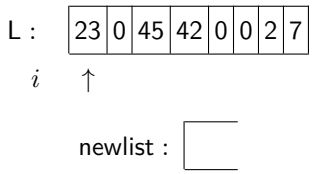
Best case scenario: This occurs when there are no 0s in the list, since there are no shuffles. The number of iterations in this case is n. The complexity is $\Theta(n)$.

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 $1+2+3+...+(n-1) = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} \approx \frac{1}{2}n^2$. The complexity is $\Theta(n^2)$.

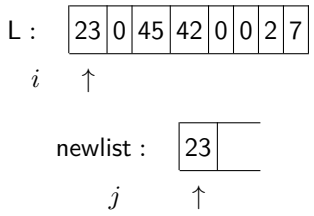
The Copy-Over Algorithm

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.

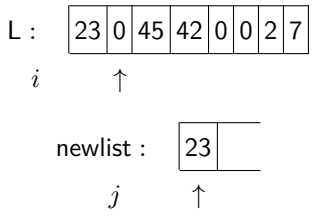
For example, we will have the list L (size $n = 8$) and we will move the legit values into the list newlist (size not yet known). We will make i our pointer for iterating through L.



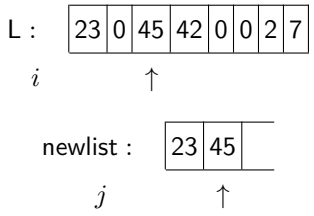
We can see that the first entry is non-zero, therefore, we put it into the newlist. We also increment j which is a pointer for newlist.



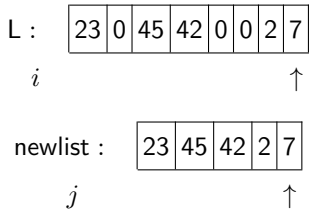
Then, we increment i and repeat.



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.



Algorithm 13 Copy-Over

1: **get** $n, L_1, ..., L_n$
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**

Building the algorithm, we have

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 $\Theta(n)$.

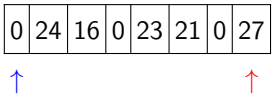
Compared to the last algorithm, this is a much more efficient algorithm in time. However, the **space complexity** is worse. It requires double storage space.

The Converging Pointers Algorithm

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

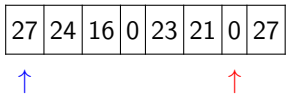
- Left moves up only if it is pointing at a non-zero value.
- If Left points at zero, copy the value at Right into Left and decrement Right by 1.
- Stop when Left = Right

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.

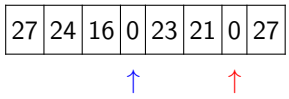


Notice that Left = 1 and Right = n = 8.

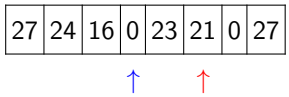
Since the value at Left is 0, we put the value at Right into Left and decrement Right by 1. The list becomes



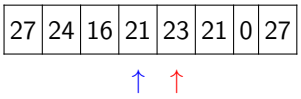
Now, Left moves forward until it is at a zero.

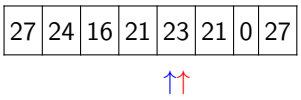


The entry at the Right pointer gets copied into the Left pointer and Right is decremented, giving



This process repeats until the pointers converge.





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.

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 $\Theta(n)$.

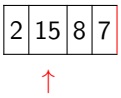
Algorithm 14 Converging Pointers

```
1: get n, L_1, ..., L_n
2: set Left = 1, Right = n
3: while (Left ≤ Right) do step 4 to step 5
4:     if (L_Left ≠ 0) then
           set Left = Left + 1
5:     else
           set L_Left = L_Right
           set Right = Right + 1
6: stop
```

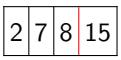
2.2.12 Sorting Algorithms

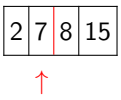
Selection Sort

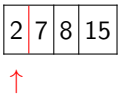
Let's say we have the list

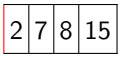


We scan the list and find the largest (denoted by the red arrow). We also start with a marker (red line) equal to the length of the list. Then, we swap the element at that index with the last element and decrement the marker. Then, we repeat on all elements before the marker.





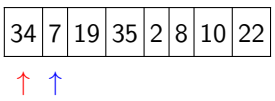




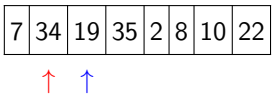
The list is now sorted. The complexity of this algorithm is $\Theta(n^2)$. See subsection 3.3.15 for more details.

Bubble Sort

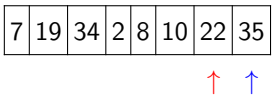
Let's say we have the following list of size $n = 8$.



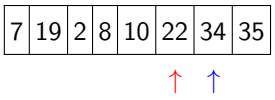
We start by comparing the first two elements with each other. If the element at the right (blue) arrow is larger than the left (red) one, then they swap. The red and blue counter advances.



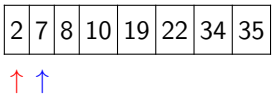
This process repeats until the arrows reach the end of the list...



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



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



The total number of iterations this algorithm will take is $(n-1)+(n-2)+...+1 = \frac{n(n-1)}{2}$. 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 if $\Theta(n)$.

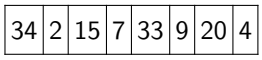
Quick Sort

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 picot is in the right location. We then apply this strategy on the left and right sub-lists.

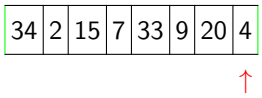
Using the following list,



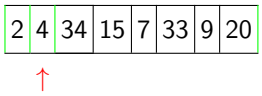
Where do we put the pivot?

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

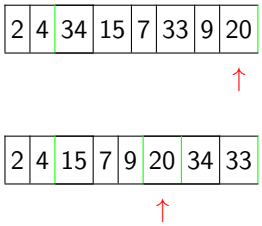
For our implementation, we will consider the first of these: Where the pivot (denoted by the red arrow) is initially at the last element in the list.



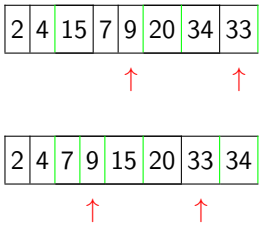
Then, elements that are less than the pivot go to the left of the pivot, and elements that are greater than the pivot goes to the right.



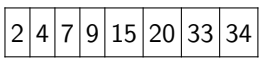
We know know that the pivot is in the correct position. We end up with two sub-lists now. Since the left sub-list is of size 1, we leave it as it is. And now we pick a pivot for the right sub-list (green boxes). We choose the rightmost element (20) and proceed with the algorithm.



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 $\Theta(n \log_2 n)$.

Subsection 2.3
Hardware

2.3.1 Binary Numbering Systems

Introduction and Definitions

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 $(A)_b = A_m A_{m-1} ... A_1 A_0 . A_{-1} A_{-2} ... A_{-n}$. The part in front of the fractional dot is the *integer part* and the part after is the *fractional part*.

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

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

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

$$\begin{aligned}(127.4)_5 &= 1 \times 5^2 + 2 \times 5^1 + 7 \times 5^0 + 4 \times 5^{-1} \\ &= 1 \times 25 + 2 \times 5 + 7 \times 1 + \frac{4}{8} \\ &= (87.5)_{10}\end{aligned}$$

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 hexdecimal 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.

Base conversion

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 $(0.6875)_{10} = (0.1011)_2$.

Octal & Hexadecimal systems

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.

Binary Representation using a fixed number of bits

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 difined 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.

2.3.2 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.

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)$_{10}$	(Signed integers)$_{10}$
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

$$\begin{array}{cccc} {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & \\ & 0 & 0 & 1 \\ + & 1 & 1 & 1 \\ \hline 1 & 0 & 0 & 0 \end{array}$$

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 arthritic operations.

We need to use another representation to fix these problems.

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 it's 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's complement representation

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

Binary	(Unsigned integers)$_{10}$	(Signed magnitude)$_{10}$	(2's complement)$_{10}$
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 arthritic operations work in this representation. We will add (+1) + (-3).

$$\begin{array}{cccc} & & {\scriptstyle 1} & \\ & 0 & 0 & 1 \\ + & 1 & 0 & 1 \\ \hline & 1 & 1 & 0 \end{array}$$

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

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.

2.3.3 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 $\in \{\text{True, False}\}$ ($\{1, 0\}$ respectively).

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

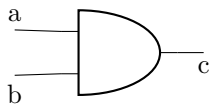
AND operator

Let a, b, c be Boolean variables. We define c = a AND b = a·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.

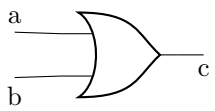
OR operator

Let a, b, c be Boolean variables. We define c = a 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:

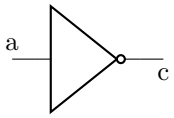


NOT operator

Let a, c be Boolean variables. We define c = 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:



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 $x + x‘ = 1$

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‘ · 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‘ · y‘ columns are identical.

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

x	y	(x·y)	(x·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·y)‘ and x‘ + y‘ columns are identical.

Thus, the de Morgan Theorem is proved.

Boolean functions

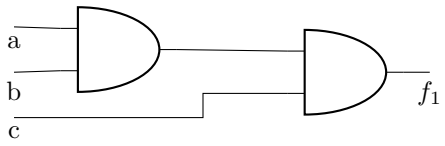
A **Boolean function** is a combination of multiple Boolean variables using multiple Boolean operators to obtain a Boolean output.

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

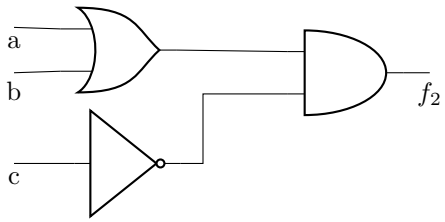
Representing this in a logic diagram, we get



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

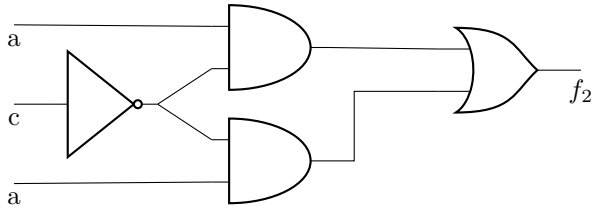
a	b	c	a+b	f_2
0	0	0	0	0
0	0	1	0	0
0	1	0	1	1
0	1	1	1	0
1	0	0	1	1
1	0	1	1	0
1	1	0	1	1
1	1	1	1	0

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,

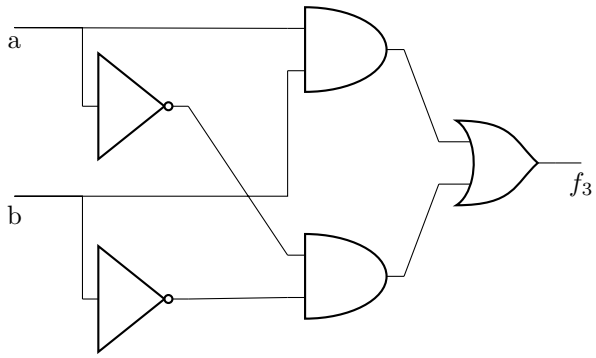


Question 3: Is the truth table unique? Yes! Always, since we use the truth table to define when two Boolean functions are equal.

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



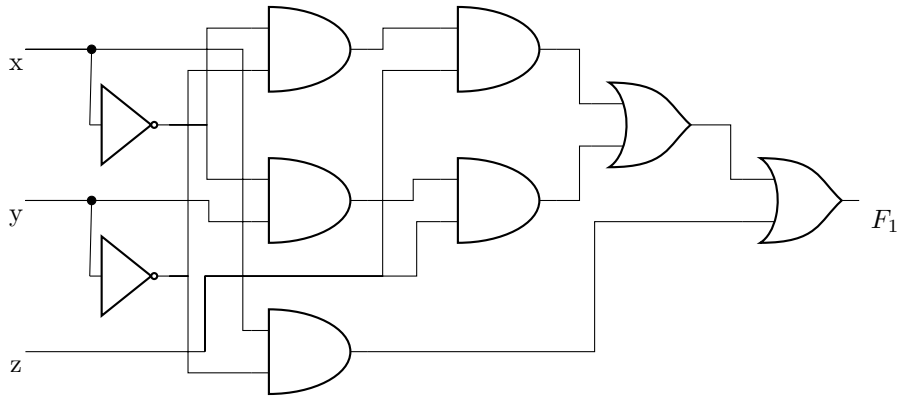
In a fourth example, we will consider $F_1 = x'y'z + x'yz + xy'$ and $F_2 = x'z + xy'$. We will prove that they are identical and show each of their logic diagrams.

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.

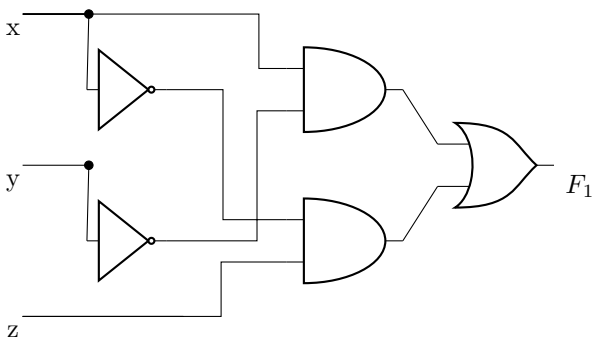
Since F_1 has more terms than F_2, we will try to convert F_1 to F_2.

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

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:

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

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

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

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 ($x = 1$ AND $y = 0$) OR when ($x = 1$ 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 ($x = 0$ AND $y = 1$) OR when ($x = 1$ 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 $f = y$.

2.3.4 Combinational Logic Circuits

A **logic circuit** is a circuit that

- Takes n binary inputs
- Transforms inputs using logic gates
- Produce m binary outputs

When we say that a logic circuit is **combinational**, we mean that the outputs at a given time t depend only on the inputs at time t. In other words, there is no notion of what the circuit has done in the past nor what it will do in the future.

The opposite of a combinational logic circuit is a **sequential** logic circuit that *does* depend on time. An example of a sequential logic circuit is an a digital clock.

If a logic circuit has n inputs, it has 2^n possible combinations of binary input.

The steps that one must follow in order to build a combinational logic circuit (CLC) is

1. From the description, build the truth table
2. From the truth table, build the Boolean expression (from the sum of products)
3. (optional) Simplify the Boolean expression
4. Draw logic diagram and implement on a circuit board

Examples

Example 1: Build a CLC that takes as input an integer between 0 and 7, detects if the input is greater or equal to 5. That is, if the input is greater than 5 then $f = 1$, otherwise $f = 0$.

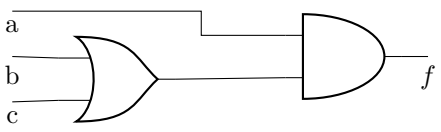
The input is an integer, so it must be converted to binary. We need 3 bits to represent 8 numbers. We will call these bits a, b and c. Now we can construct the truth table.

a	b	c	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Converting to a Boolean expression, we have that $f = a\bar{b}c + ab\bar{c} + abc$. Simplifying,

$$\begin{aligned} f &= a\bar{b}c + ab\bar{c} + abc \\ &= a\bar{b}c + ab(\bar{c} + c) \\ &= a\bar{b}c + ab \\ &= a(\bar{b}c + b) \\ &= a\overline{((b + \bar{c}) + \bar{b})} \\ &= a\overline{(b\bar{b} + \bar{c} + \bar{b})} \\ &= a\overline{(\bar{c} + \bar{b})} \\ &= a(b + c) \end{aligned}$$

Thus, $fa(b + c)$. We can now make the logic diagram:



Example 2: Build a CLC where $f = 1$ if the number of 1s in the input is *odd* nad f takes 3 binary inputs. For instance, with 101, $f = 0$; but with 010, $f = 1$. The truth table is therefore

a	b	c	f
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

We can find that $f = \bar{a}\bar{b}c + \bar{a}b\bar{c} + a\bar{b}\bar{c} + abc$.

Now we must also make the logic diagram. That's a large circuit... We will leave it as an exercise for the reader.

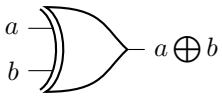
XOR and XNOR Gates

The **exclusive or** (XOR) is denoted with the $\bigoplus$ symbol. We say that $c = a \bigoplus b = a\bar{b} + \bar{a}b$.

The truth table of the XOR is

a	b	$a \bigoplus b$
0	0	0
0	1	1
1	0	1
1	1	0

The circuit diagram for it is

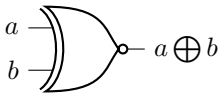


The complement to the exclusive or is the **exclusive nor** (XNOR). We say that $c = \overline{a \bigoplus b} = \overline{a\bar{b} + \bar{a}b} = ab + \bar{a}\bar{b}$.

The truth table of the XNOR is

a	b	$\overline{a \bigoplus b}$
0	0	1
0	1	0
1	0	0
1	1	1

The circuit diagram for it is



The properties of the XOR are

- $x \bigoplus 0 = x$
- $x \bigoplus 1 = \bar{x}$
- $x \bigoplus x = 0$
- $x \bigoplus \bar{x} = 1$

The Half Adder

The objective of the half adder is to build a logic circuit that adds two bits. For example,

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

In the last case, there is a carry.

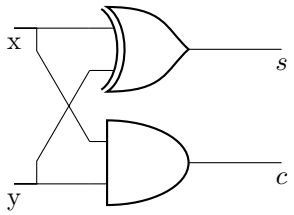
The number of *inputs* is 2. The number of *outputs* is 2 (sum (s) and the carry (c)).

Making the truth table,

x	y	s	c
0	0	0	0
1	0	1	0
0	0	1	0
1	1	0	1

The Boolean expression for the sum output in this table is $s = \bar{x}y + x\bar{y} = x \bigoplus y$. The Boolean expression for the carry in this table is $c = xy$.

Drawing the logic diagram,



The Full Adder

The full adder is a circuit that adds 2 binary inputs (x and y) and a carry in (c_{in}) and results in a sum (s) and carry out(c_{out}).

The truth table is

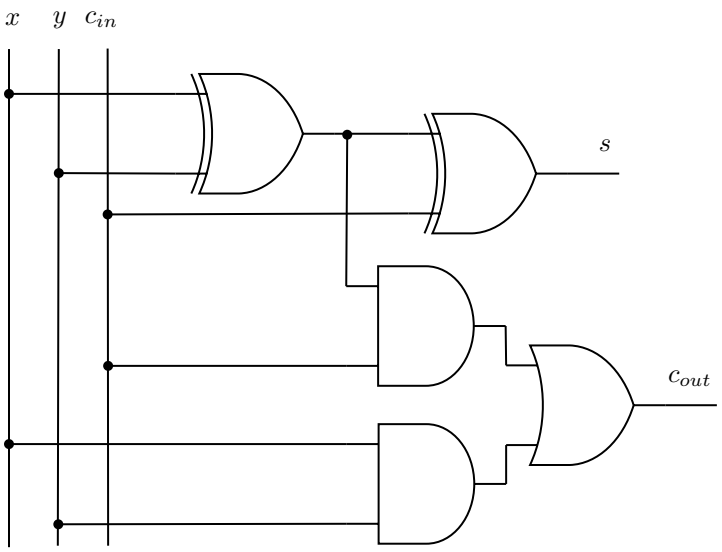
x	y	c_{in}	s	c_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

The Boolean expression of the sum is $s = \bar{x}\bar{y}c_{in} + \bar{x}y\overline{c_{in}} + x\bar{y}\overline{c_{in}} + xyc_{in}$. Also, the Boolean expression for the carry out is $c_{out} = \bar{x}yc_{in} + x\bar{y}c_{in} + xy\overline{c_{in}} + xyc_{in}$. We can simply these expressions.

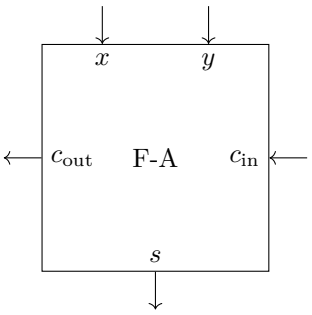
$$\begin{aligned} s &= \bar{x}\bar{y}c_{in} + \bar{x}y\overline{c_{in}} + x\bar{y}\overline{c_{in}} + xyc_{in} \\ &= \bar{x}(\bar{y}c_{in} + y\overline{c_{in}}) + x(\bar{y}\overline{c_{in}} + yc_{in}) \\ &= \bar{x}(y \oplus c_{in}) + x\overline{(y \oplus c_{in})} \\ &= x \oplus (y \oplus c_{in}) \end{aligned}$$

$$\begin{aligned} c_{out} &= \bar{x}yc_{in} + x\bar{y}c_{in} + xy\overline{c_{in}} + xyc_{in} \\ &= (\bar{x}y + x\bar{y})c_{in} + xy(\overline{c_{in}} + c_{in}) \\ &= c_{in}(x \oplus y) + xy \end{aligned}$$

The logic diagram of the full adder is therefore



From now on, we can write this as a black box as follows



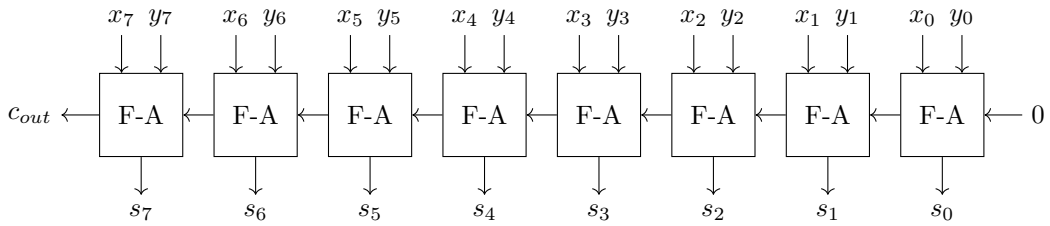
8 bit Binary Adder

We would like to add two numbers expressed each in 8 bits.

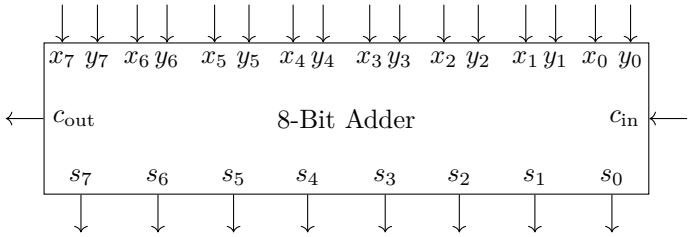
	x_7	...	x_1	x_0
+	y_7	...	y_1	y_0
c_{out}	s_7	...	s_1	s_0

The number of inputs is 16 and the number of outputs is 9. This program has $2^16 \approx$ 64000 combinations... Quite the truth table. We will skip that step.

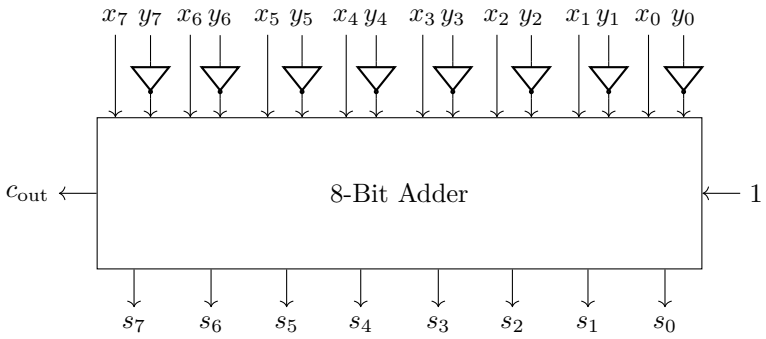
Instead we will construct the logic diagram by chaining full adders.



We can compact the **8 bit adder** in a black box as follows.

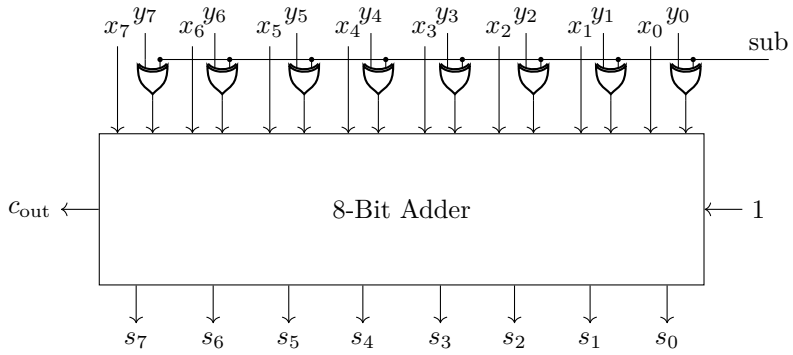


Let us now see if we can make an **8-bit subtractor** now. We want to find $x - y =$ $x + 2$'s complement of $y = x + \bar{y} + 1$. Therefore, all we have to do to make a subtractor is to invert y_i and set $c_{in} = 1$. The following is the diagram.



8-Bit Adder/Subtractor

We now want to make a circuit that take x and y (which are 8 bit integers), and computes $x + y$ or $x - y$ based on a selection made by the user. We can do this using XOR gates.



Moving Forward

So far we have used the sum of products method to find

- Arthmetic circuits (Adder & Subtractor)
- Logic circuits (Equality test)

Both of these types of circuits together are part of the **Arithmetic Logic Unit** (ALU).

The *CPU* is made of the *ALU* and the *Control Unit* (CU). The control unit is responsible for managing the ALU and memory. The control unit is composed of combinational circuits. These circuits perform control operations. What are these?

- Selection: Select data from memory
- Decoding: Decode the operation code
- Encoding
- ...

We will know build control circuits for Selection and Decoding.

The Multiplexer

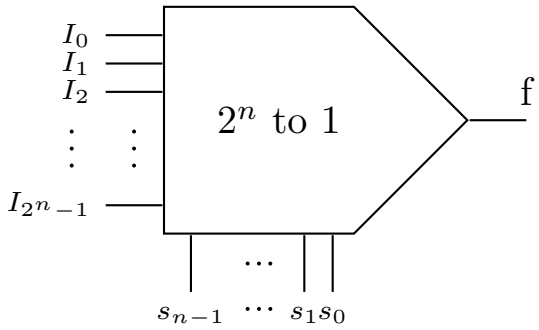
The **multiplexer** is a selection circuit which

- selects one of many inputs
- direct it to the output

A multiplexer is denoted as MUX. A MUX has

- 2^n inputs: I_0, I_1, I_2, ..., I_{2^n-1} where $I_i \in \{0, 1\}$
- 1 output
- n selection lines: s_0, s_1, ..., s_{n-1}

The decimal value of $s = s_{n-1}...s_1s_0$ is the *index of the input that will be directed to the output*. The multiplexer diagram is

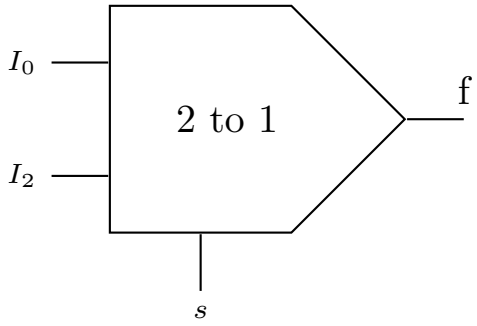


If $s_{n-1}...s_1s_0 = (00...0)_2 = (0)_{10} \implies f = I_0$.
If $s_{n-1}...s_1s_0 = (00...1)_2 = (1)_{10} \implies f = I_1$. And so on...
If $s_{n-1}...s_1s_0 = (11...1)_2 = (2^n - 1)_{10} \implies f = I_{2^n-1}$.

An everyday example of a multiplexer is the decimal multiplexer on a remote control for a TV. If you have $100 = 10^2$ channels (0 to 99), in order to select one, you choose two decimal numbers (say 8 and 5) to be your s_1 and s_0. The multiplexer then connects the input (in this case channel 85) to the output (TV).

2 to 1 Multiplexer

A 2 to 1 multiplexer will have $n = 1$ and therefore $2^n - 1 = 2^1 - 1 = 2$ inputs (I_0 and I_1) with $n = 1$ selection lines ($s_0 \equiv s$) and an output f. The corresponding diagram is



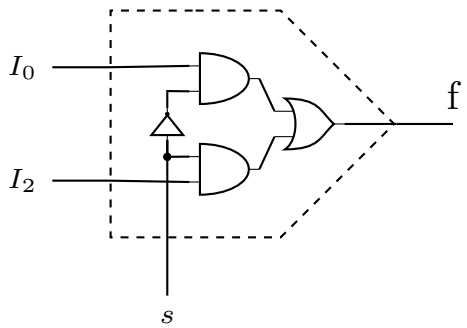
If $s = 0$ then $f = I_0$ and if $s = 1$ then $f = I_1$. We will now use the *sum of products* technique (as usual). The truth table of the 2 to 1 multiplexer is

I_1	I_0	s	f
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

The Boolean expression is therefore $f = \bar{I}_1 I_0 \bar{s} + I_1 \bar{I}_0 s + I_1 I_0 \bar{s} + I_1 I_0 s$. We can simply this expression.

$$\begin{aligned} f &= \bar{I}_1 I_0 \bar{s} + I_1 \bar{I}_0 s + I_1 I_0 \bar{s} + I_1 I_0 s \\ &= I_0 \bar{s}(\bar{I}_1 + I_1) + I_1 s(\bar{I}_0 + I_0) \\ &= I_0 \bar{s} + I_1 s \end{aligned}$$

Thus, $f = I_0 \bar{s} + I_1 s$. We can draw this diagram as



In order to scale this up to larger multiplexers, we need a different method because the standard sum of products method is too inefficient.

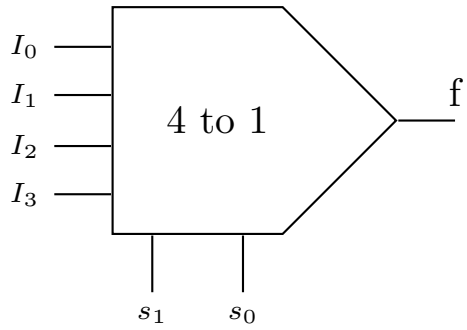
To do this, we will make a compactified truth table by only using s as an input and instead consider them as the values of the output.

s	f
0	I_0
1	I_1

Now we can get a Boolean expression from this truth table now. Using this method, we and each of the outputs of f (in this case I_0 and I_1) with the inputs (following the same rule as before, where a not in inserted if the value is 0). Namely, here we get that $f = I_0\bar{s} + I_1 s$. Note that this is the same as the expression we got using the previous method, but with less steps.

4 to 1 Multiplexer

In a 4 to 1 multiplexer, we will have 4 inputs and 2 selection lines. The multiplexer looks like

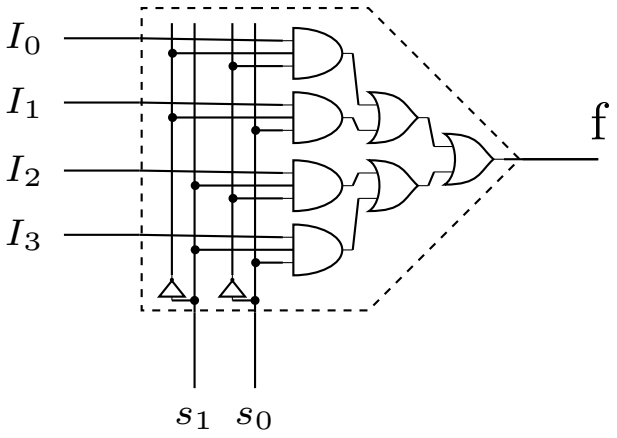


If $s_1 s_0 = (00)_2 \implies f = I_0$.
If $s_1 s_0 = (01)_2 \implies f = I_1$.
If $s_1 s_0 = (10)_2 \implies f = I_2$.
If $s_1 s_0 = (11)_2 \implies f = I_3$.

The truth table (compact form) is

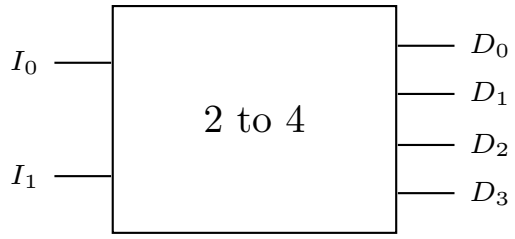
s_1	s_1	f
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

Thus, the Boolean expression is $f = I_0\bar{s_1}\bar{s_0} + I_1\bar{s_1}s_0 + I_2 s_1\bar{s_0} + I_3 s_1 s_0$. Now, we can draw the multiplexer circuit.


I_0
I_1
I_2
I_3
f
s_1
s_0

The Decoder (2 to 4)

The decoder is the opposite of a multiplexer. Instead of encoding a larger input into a smaller output, it decodes a smaller input into a larger output. It is denoted with the following diagram.



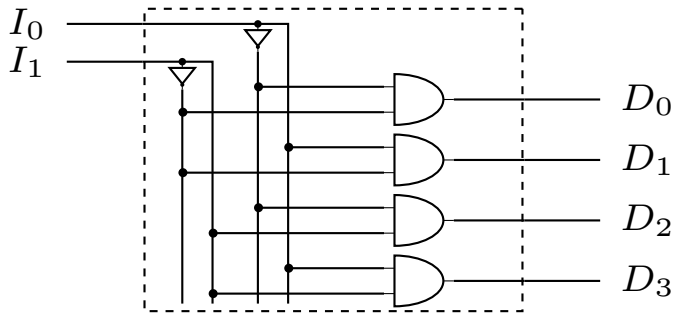
The truth table is

I_1	I_0	D_0	D_1	D_2	D_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

The Boolean expressions are therefore:

$$D_0 = \bar{I}_1\bar{I}_0$$
$$D_1 = \bar{I}_1 I_0$$
$$D_2 = I_1\bar{I}_0$$
$$D_3 = I_1 I_0$$

We can now draw the circuit diagram of the decoder:



Note that we can only make n to 2^n decoders: The number of outputs must always be a power of two.

For the 3 to 8 decoder, we would obtain the Boolean expressions: $D_0 = \bar{I}_2\bar{I}_1\bar{I}_0$, $D_1 = \bar{I}_2\bar{I}_1 I_0$, ... $D_7 = I_2 I_1 I_0$.

2.3.5 Memory Organization

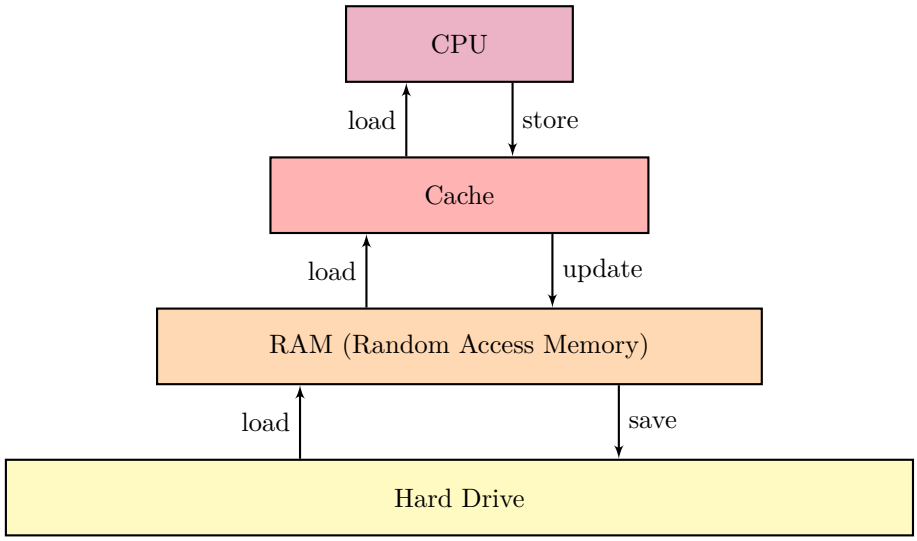
Computer Components

Modern computers are built using the **Von Neumann machine**. There are three aspects:

- **Architecture**: **I/O** (User interaction) + **Memory** (Storage) + **CPU** (*CPU*: Control Unit, *ALU*: Arithmetic and Logic Unit). These are all connected by a shared bus.
- **Stored Programs**: All programs and data are stored in memory (binary).
- **Sequential Execution**: Also called the **fetch-decode-execute** cycle. Instructions are **fetched** from memory, **decoded** by the CPU and then **executed** by the ALU. If there is a result, it is stored back in memory.

Memory

The memory is organized in a **hierarchy**. At the bottom of the hierarchy is the Hard Drive (in TB). At the top is the CPU. Since the hard drive is slow, when some data from the hard drive is needed, it is first loaded into **RAM (Random Access Memory)** (in GB). The RAM is still too slow for the RAM, so the data is stored in **cache** (in KB or MB). Yet still, this is not fast enough for the CPU, so **registers** (in Bytes) in the CPU itself are used to store variables.



As you **go up** the hierarchy, the **speed increases**, but the **size decreases** and the **cost increases**.

RAM

Random access memory is organized in an array of Bytes ("words").

Words in RAM are addressed with a byte themselves (e.g. 01101101 is an address). These are typically written in hexadecimal (e.g. 6D).

Words in RAM can be data or machine code instructions. Instructions contain a binary code for each operation (for example, addition). Instructions codes are dependent on the CPU.

For an n-bit computer system, there are 2^n bytes in memory (addressed in decimal with 0 to $2^n - 1$).

The size of the RAM is equal to the number of cells in memory times 1 byte. Thus, *the max size of the RAM in a n-bit computer is* 2^n *bytes.* For example, in a 32-bit computer, the (max) size of the RAM is 2^{32} bytes. This is a large number that is hard to get a grasp of, so we will use prefixes to denote larger numbers of bytes.

Namely, a **kilobyte** (KB) is $2^{10} = 1024 \approx 1000$. A **megabyte** (MB) is 2^{20}, a **gigabyte** (GB) is 2^{30}, a **terabyte** (TB) is 2^{40}, and so on.

We can write 2^{32} bytes $= 2^2 \times 2^{30}$ bytes $= 4$ GB. Thus, the max memory size of a 32-bit computer is 4 GB.

If instead we had a 64-bit computer, the max RAM size is 2^{64} bytes $= 2^{24} \times 2^{40}$ bytes $= 2^{24}$ TB. This is a very large number and totally impractical in real life. Therefore, in a 64-bit computer, we do not need to display memory addresses with the full 64-bits. The data it is necessary to, but not the memory addresses.

It is important to always distinguish between the address and content of a cell in memory. For example, the content could store the value 15, but the address is 34.

Memory Basic Operations

There are 2 basic operations.

Firstly, the **fetch** operation: Reads data from the memory. The idea is that we want to get a value from memory using a fetch operation, but we need to know where to get this value, so we use an *address* (value = fetch(address)). For example, value = fetch(34) returns the value 15 (which is stored in the memory address 34).

Secondly, the **store** operation: Stores data into memory. We need both a value and an address (store(value, address)) For example, stores(22, 34) would store the value of 22 in adress 34.

How can we do this in hardware?

To do this, we will discuss **registers**. Registers are storage cells (capacity in bits) and is located outside of the RAM (for example, in the CPU).

A system is equipped with 2 registers:
The **MDR:** Memory Data Register. It contains data read from (or to be stored in) memory.
The **MAR:** Memory Address Register. It contains the address for fetch *or* store.

For example, the MAR allows us to select a location in memory and the MDR allows us to transfer data from or to memory.

In a *fetch* operation, the steps are as follows:

- Put address in MAR

- Read
- Data is sent to MDR

In a *store* operation, the steps are as follows:

- Put address in MAR
- Put data in MDR
- Write

Address Decoding

The MAR will have N bits of memory. How to we select a location in RAM given a certain address in the MAR? We need a circuit that takes in N bits and gives 2^N outputs (one for each address in memory). Thus, the circuit that does this is a **N to 2^N bit decoder**. This is called *address decoding*.

Arithmetic and Logic Unit

The CPU is composed of the ALU and the CU.

The Arithmetic and Logic Unit (**ALU**) is responsible for implementing arithmetic and logic operations (A+B, A-B, A==B).

The ALU contains

- Registers to store operands and result
- Combinational circuits for operations (+, -, *, ==)
- Selection circuit (for operation)

Let us assume that we have 16 registers: R_0, R_1, ..., R_0.

We select a register for the left operand (eg. A), another register for the right operand (B), and a last for the operation (+).

In summary, there is a selection circuit to select the left and right operands (this is performed by the CU). The result of this selection is connected to all the operations (+/-, <, ==, etc.), but only one operation is of interest. The desired operation output is selected by a multiplexer which returns the value to a register. The selection lines come from the CU.

Control Unit

The control unit (**CU**) is responsible for the ALU operations and memory operations, but most importantly is responsible for executing the stored program.

The CU will fetch (memory operation), decode (select operation in ALU) and execute (select in ALU and save to memory).

The stored program is binary, but written in **machine language** instructions.

Machine language is not C++ or Java or Python. These are high level languages (designed for programmers). Machine language is a low level language (designed for the computer).

To go from a high level language to a low level language, we use a **compiler**. For example, the compiler translates C++ to machine language (binary).

Machine language varies across CPU architecture (or brand, such as Intel or AMD) uses its own set of machine instructions. For example, 64-bit AMD CPUs uses the AMD64 instruction set and Intel uses the X86 or X64 instruction set.

We can divide machine instructions into

- Data transfer
- Arithmetic
- Compare
- Branch

2.3.6 Machine Language Instructions

We will define a simple machine language for pedagogical reasons.

Data Transfer

Data transfer instruction moves data between the CPU and memory. We will define the following 3 instructions:

LOAD X: Loads a register R in the CPU with the content of memory at address X. We write: Con(R) = Con(X). Con stands for content.

STORE X: Stores the content of R in memory at the address X.

MOVE X, Y: Copies the content at memory address X into the memory address Y. We write: Con(Y) = Con(X).

Arithmetic

There are many possible arithmetic operations, but we will define the following instructions as an example:

ADD X, Y: Con(X) = Con(X) + Con(Y).
ADD X: Con(R) = Con(X) + Con(R). R is a predetermined register (which you must load a value into prior, and store a value from after).

There are many more.

Compare

To implement the compare operation, the CPU performs a subtraction and based on the sign (+ or -) and magnitude of the result (0 or not) returns a value.

The CPU sets the values of the **Condition Code Register** (CCR). There are three bits in the CCR. The less than (LT), the equal (EQ), and the greater than (GT):

$$\text{CCR}: \boxed{\text{LT}}\boxed{\text{EQ}}\boxed{\text{GT}}$$

We define the following instruction:

$$\textbf{COMPARE X, Y}: \begin{cases} \text{GT=1, EQ=0, LT=0} & \text{if Con(X)} > \text{Con(Y)} \\ \text{GT=0, EQ=1, LT=0} & \text{if Con(X)} = \text{Con(Y)} \\ \text{GT=0, EQ=0, LT=1} & \text{if Con(X)} < \text{Con(Y)} \end{cases}$$

Branch

In the memory, we have the program memory which are the set of instructions that the computer uses to run.

To go to a different part in the program (jump across instruction lines in memory), we need to use a branch instruction. This allows to jump to an instruction stored at a specified address in memory.

JUMP X: The next instruction to execute is stored at the memory address X.

JUMPGT X: Jump to X only if GT = 1 (otherwise, no effect).
JUMPEQ X: Jump to X only if EQ = 1.
JUMPLT X: Jump to X only if LT = 1.
These instruction must be proceeded by a compare operation.

HALT: Stops the program.

Example: Arithmetic Operation

If we want to perform the psudeocode:

Set a = b + c + d

then we would have the following in program memory:

50	LOAD 101
51	ADD 102
52	ADD 103
53	HALT
...	...
100	a
101	b
102	c
103	d
...	...

What occurs is that first a is loaded into R, then b is added to the content of R, then d is added to the content of R. Finally the program is halted.

Example: Conditional

If we want to perform the psudeocode:

```
If (a == b) then
    Set c = d
Else
    Set c = b
Stop
```

then we would have the following in program memory:

50	COMPARE 100, 101
51	JUMPEQ 54
52	MOVE 101, 102
53	JUMP 55
54	MOVE 102, 103
55	HALT
...	...
100	a
101	b
102	c
103	d
...	...

First, we compare the content at address 100 (a) to the content at address 101 (b). Then, we conditionally jump if EQ = 1, set c = d and then halt. Otherwise, we continue on and set c = b before jumping to the halt.

Example: Loop

If we want to perform the psudeocode:

While (a < b) then
    Set a = a+c
EndWhile
Stop

then we would have the following in program memory:

50	COMPARE 100, 101
51	JUMPEQ 57
52	JUMPGT 57
53	LOAD 100
54	ADD 102
55	STORE 100
56	JUMP 50
57	HALT
...	...
100	a
101	b
102	c
...	...

First, we compare a to b. Then, we conditionally jump to the halt if EQ = 1 or if GT = 1. If not, we load a in R, add c, store the content of R back into a, and then jump back up to the compare.

Machine Language Encoding

Instructions are written in binary. How do we encode them into a N-bit system?

op code	address 1	address 2

The first field is for the binary code of the operation, and the other two ar for the address of the 1st and/or 2nd operand(s).

For example, if we want to do the instruction

ADD 99, 100

Using an op code table we can find the op code of the ADD operation. If there are 8 bits for the code and $(0)_{10}$ is the op code for the ADD operation, then 0000 0000 is the binary op code of the instruction. Further, we can convert address 99 and 100 into binary to get that the instruction is the following in binary.

0000 0000	0110 0011	0110 0100

What occurs in the hardware is that the instruction is fetched from memory (written in binary as above), then the instruction is decoded, then executed, and finally it proceeds to the next instruction.

Instruction Register & Instruction Decoder

The **instruction register** (IR) is used to hold the instruction to execute. It holds the opcode and operand addresses.

The **instruction decoder** selects the circuit responsible for the execution based on the opcode value.

Example: Assume the following 3 bit opcode table.

op code	instruction
000	add
001	load
010	jump
011	store
...	...
111	halt

Physically, the memory is attached (via the fetching selection circuit discussed earlier) to the IR. The op code part of the instruction is connected to the instruction decoder. In this case, the instruction decoder has 3 inputs and 8 outputs (a 3 to 8 decoder). Each output pd the instruction decoder is connect to the enabling of its own operation as according to the op code table above.

Program Counter

The **program counter** (PC) is a register that stores the address of the *next* instruction to execute.

How do we update the program counter?

If the program is in **sequential** execution, we that the content of pc (an address) and increment it. We write Con(PC) = Con(PC) + 1. This depends on the size of the instructions in memory, of course, but for an 8bit instruction, the PC is incremented by 1 byte.

If program is in **not sequential** execution (Jump, halt, etc.), Con(PC) is extracted from the branching of the instruction.

For example, if we do the following program, loaded at address 0.

LOAD 10
ADD 11
STORE 12
JUMP 5

ADD 13
HALT

The program is written in memory as

Address	Instruction	
0	000	01010
1	001	01011
2	011	01100
3	010	00101
4	001	01101
5	000	00000
...	...	
10	a	
11	b	
12	c	
13	d	

The address of the PC and the instruction held by the IR in each step of the program is shown by going down the following table. Note that the PC register contains 5 bits because there are 5 bits in the instruction allocated to the memory address and the IR will contain the full 8 bits in an instruction, of course.

PC	IR
00001	000 01010
00010	001 01011
00011	011 01100
00101	010 00101
00110	111 00000

The Von Neumann Machine

A diagram of the Von Neuman Machine is found in the textbook on page 259.

The following operations are made using the indicated steps.

Fetch:

- PC –> MAR
- FETCH
- MDR –> IR
- PC + 1 –> PC

Decode:

- IR(opcode) –> Instruction Decoder

LOAD X:

- IR(addr) –> MAR
- FETCH
- MDR –> R

STORE X:

- IR(addr) –> MAR
- R –> MDR
- STORE

ADD X:

- IR(addr) –> MAR
- FETCH
- MDR –> ALU
- R –> ALU
- ADD
- ALU –> R

JUMP X:

- IR(addr) –> PC

COMPARE X:

- IR(addr) –> MAR
- FETCH
- MDR –> ALU
- R –> ALU
- SUBTRACT (this will set CCR)

2.3.7 The Assembly Programming Language

The assembly programming language is higher level than machine code, but lower than C++ or Python.

A **high-level programming language** (eg. C++, C, Java, C#, Python, etc.) is written in a text file. In order for the program to run by the program, the text must be converted to binary machine code. This is done with the **compiler**.

A **low-level programming language** (eg. Assembly) is higher than machine code, but lower than high-level languages. The assembly code must still be converted from as text file to a binary file executable by the computer. This is done with the **assembler**.

Structure of an Assembly Program

We start the assembly program with a .BEGIN. The basic structure of an Assembly program is as follows

```
.BEGIN

label: opcode reference -- comments
    ...
    ...
    HALT
```

The program ends when it reaches a HALT instruction.

The code segment is where we put the code. It is everything in between the .BEGIN and the HALT. THe data segment comes after.

A label is used if it is needed to jump to the instruction which has that label. This is where you loop to. It is optional.
The opcode is the operation desired (such as LOAD).
There are then reference variables (variable name instead of addresses).
Comments are explanations. This is optional.
A typical Assembly loop looks like

```
loopstart: load x -- loop starts here
    ...
    ...
    jump loopstart
```

We can declare variables in the data segment of the Assembly code.

```
.BEGIN
        ...
        ...                     -- This is the code segment
        ...
        HALT

x:                  .Data 0
five:   .Data +5    -- This is the data segment
...
.END
```

The end of the data segment is marked with a .END.

Converting an Algorithm to Assembly

Assume we have the following algorithm.
To convert to assembly, we must first identify all the variables and constants required.

Variables: SUM, N
Constants: 0

This goes in the data subsection. The following is the data subsection.

Algorithm 15 Algorithm to convert to Assembly

1: **set** SUM = 0
2: **get** N
3: **while** $(N \geq 0)$ **do** step 4 to step 5
4: **set** SUM = SUM + N
5: **get** N
6: **print** SUM
7: **stop**

```
SUM:        .Data 0
N:              .Data 0
ZERO:   .Data 0
```

Now we can translate the instructions. Refer to the Assembly op code table to do this. The following is the completed translation of the algorithm to Assembly language.

```
.BEGIN

            clear SUM                                                   -- Sets SUM to 0
            in N
         -- Gets N from user
start:              load ZERO                                                  --
     register R now contains a 0
            compare N                                                  -- compares N
      with 0 (in register R)
            jumplt printsum                          -- jump out of loop if N>=0
            load SUM                                                    -- R now
     contains SUM
            add N
     -- now R contains sum + N
            store SUM                                                  -- stores SUM
      in memory
            in N
         -- get N
            jump start
printsum:       out SUM
            HALT

SUM:                    .Data 0
N:                          .Data 0
ZERO:               .Data 0

.END
```

The Assembler

The assembler is a piece of software that is responsible for translating the Assembly code to the machine instruction code.

The assembler works in two full *passes*. It will do a first pass and a second pass. The objective of the first pass is to assign addresses to all labels (variables and/or loops) and create a **symbol table** which defines a correspondence between the labels and addresses. In the second pass, the assembler will use both the symbol table (which tells the assembler the address that it should use for each label) and the op code table (which

tells the assembler the binary code that it should use for each instruction). The second pass creates the machine instruction code.

We will now translate the following assembly code into machine instruction code.

```
.BEGIN

loop:       in x
        in y
        load x
        compare y
        jumpgt Done
        out x
        jump loop
Done:   out y
        halt

x:                  .Data 0
y:                  .Data 0

.END
```

In the **first pass**, we will find what address we will use to replace each label. To do so, the assembler assumes that the program will be loaded started at address 0. To keep track of each address, we associate an address to each instruction starting with the first, including the data segment. Then, the assembler goes through the program and whenever it sees a new label, it records its address in the symbol table. For this example, we would get the following symbol table:

Symbol	Address
loop	0
Done	7
x	9
y	10

For the next step, we will also use the following opcode table:

Op code	Binary
in	1101
load	0000
compare	0111

In the **second pass**, we go through each instruction line and directly translate this into machine instruction code. Note that the addresses are 4 bits long here. We will get the following result.

Address	Instruction	
0	1101	1001
1	1101	1010
2	0000	1001
3	0111	1010
...	...	
9	x	
10	y	