CS 201 Notes

Introduction to Computer Science

Jaeden Bardati

Last modified October 5, 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 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:

	a_{m-1}	...	a_1	a_0
+	b_{m-1}	...	b_1	b_0
c_m	c_{m-1}	...	c_1	c_0

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.

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 = $\pi \times$ 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** (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 ≥ 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
```

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, ..., 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
```

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, ..., L_N.

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 $y = 5$.

Algorithm 7
Find Largest

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

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** $x =$ 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 $= 1 + 2 + 3 + ... + n$. 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.

Algorithm 9
Gauss Sum

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

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*.

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.

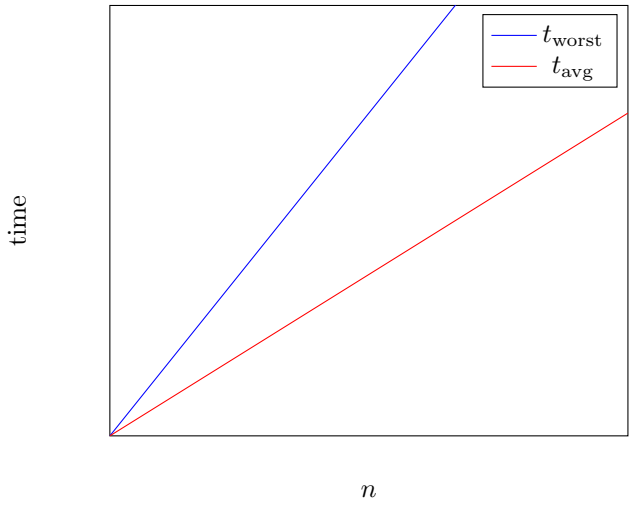
Average case scenario: The target has an equal likelihood to be in any location (1, 2, 3, 4, ..., n). The average is therefore $\frac{1+2+3+4+...+n}{n}$. This is a Gauss sum which equals $\frac{(n+1)n}{2n} = \frac{n+1}{2} = \frac{n}{2} + \frac{1}{2}$. Thus, the average case scenario has a number of iterations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

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.

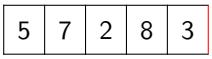
The **complexity** of this algorithm is $\Theta(n)$. That means linear scalability.

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.

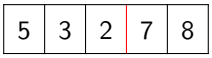
For example, if the list = 5, 7, 2, 8, 3, then the sorted list should = 2, 3, 5, 7, 8. Here $n = 5$.

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:



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($L_1, ..., L_{\text{Marker}}$)
5: Swap(largest, L_{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

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

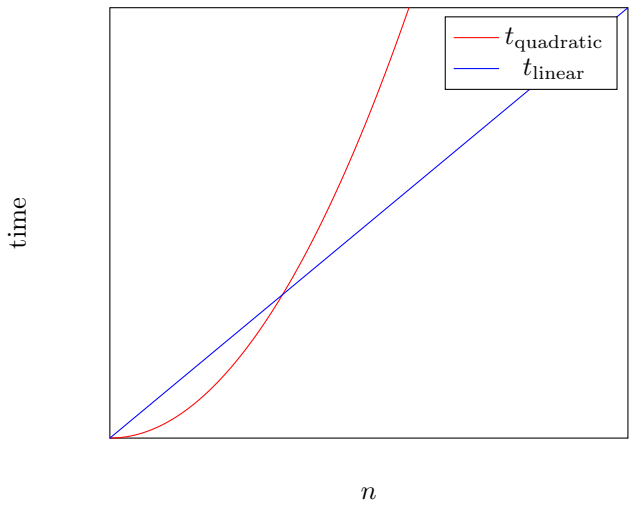
This is once again a Gauss sum. So the number of iterations required for selection sort is $\frac{n(n+1)}{2}$. For very large n,

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

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."

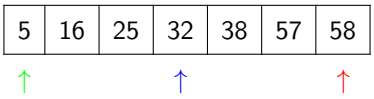
1.10 The Binary Search Algorithm
October 5, 2021

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?

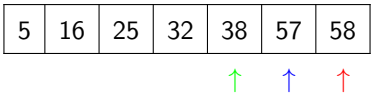
The answer is **yes, but the list must be sorted**. If the list is sorted, then we can use the **binary search** (also called the *half interval search*).
For example, we will use the following list with $n = 7$.



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
```

1.10.1 Complexity Analysis

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

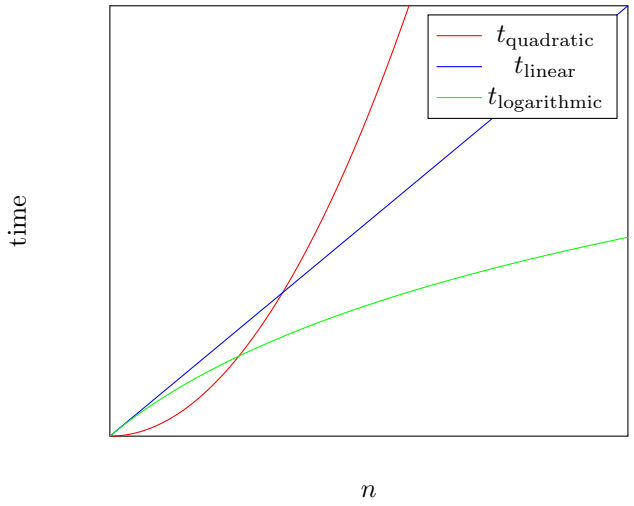
Worst case scenario: This happens when the target is not in the list. To do the analysis we will look at varying data sizes n:

- $n = 4 = 2^2$ ———> 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.