Chapter 0
Algorithms in Computing

An algorithm is a well-defined procedure that takes some input(s) and produces a correct output in finite amount of time. We will see a formal definition in CS 3780 later on....

In CS 2010 Fundamentals, we learned how to come up with an algorithm, then convert it to a Python program to run in a computer. In CS 2370 Intro. to Programming, we learned how to write a more sophisticated Python program. In CS 2381 Data Structures, we learned how to come up with a different program, using alternative data structures.

In this course, we will learn how to come up with a “best” algorithm to solve a problem.

We will see what problems cannot be solved in CS 3780.

Question: What do you mean by “best”? 😐
What is an algorithm?

We certainly have talked about a lot of algorithms in between, *functions* in *C, PHP* and *Python*, and *methods* in *Java*, each comes with 1) a name, 2) a list of parameters, 3) a type of an output, and 4) a procedure, i.e., an algorithm, that turns these inputs to the output. For example, below is a *C* function to get $\text{base}^n$:

```c
int power(int base, int n){

    int p;
    for(p=1; n>0; --n)
        p=p*base;
    return p;
}
```

Back in *CS 2010*, we once compared an algorithm to a recipe, in the sense that a recipe, often with a name, also specifies its ingredients, and a procedure that a chef follows to turn all the ingredients into an output (a dish).
A recipe I once used

Below is something I once took out of my bread recipe book for the traditional white bread:

The stuff (*Input*): 1 cup and 2 tablespoons water, 1 tablespoon butter or margarine, 1 $\frac{3}{4}$ teaspoons salt, 3 cups bread flour, 1 tablespoon dry milk, 2 tablespoons sugar, and 2 $\frac{3}{4}$ teaspoons active dry yeast.

The process (*Procedure*):

1. Measure and add liquid ingredients to the bread pan.

2. Measure and add dry ones (except yeast) to the bread pan.

3 Form a well in the flour to dump the yeast, which must NEVER come into contact with liquid when you are adding ingredients.
4. Snap the baking pan into the bread maker and close the lid.

5. Press “Select” button to choose the Basic setting.

6. Press the “Crust Color” button to choose light, medium or dark crust.

7. Press the “Start/Stop” button.

8. Wait for 75 minutes.

9. Your bread (Output) is ready.

With this recipe, we specify what we need, what we will get at the end, how to do it, and how long we have to wait.

This is exactly what we need in order to come up with an algorithm.

**Question:** Is this the “best” recipe? 😊

**Answer:** What do you mean by a “best” recipe?
A “real” example

An algorithm is often used to solve a computational problem. Sorting is one such problem, which is to put things into order.

Sorting finds a wide range of applications: You “see” a sorting whenever you Google. We certainly talked about sorting in CS 3600 Database, where an index file makes query much faster.

There are quite a few sorting algorithms, and we need to do it a lot. It thus makes sense to find the best sorting algorithm: fastest and/or cheapest, often in terms of the size of the data.

The sorting problem hence provides a nice example of algorithm analysis: 1) find out how much work a sorting algorithm has to do, data comparison and/or movement for this case, then 2) compare them to choose the best one.
Formally speaking...

To really understand the sorting problem, let’s give it a definition by specifying the input and output of this problem.

**Input:** a list of $n$ pieces of data: $\langle a_1, a_2, \ldots, a_n \rangle$, where duplicates are allowed

**Output:** a permutation $\langle a'_1, a'_2, \ldots, a'_n \rangle$, i.e., a different arrangement, of the input such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

**Process:** *How to turn an input to an output?*

In general, an instance of a problem consists of a specific input, together with whatever constraints such an input must satisfy, so that we can compute a corresponding output.

Thus, the sequence of integers $\langle 31, 41, 59, 26, 41, 58 \rangle$ is an instance of the sorting problem.
What should an algorithm be?

First and foremost, an algorithm has to be correct, namely, for every input instance of a problem, an algorithm has to halt with a correct output for this instance.

For example, if a sorting algorithm, besides producing $\langle 26, 31, 41, 41, 58, 59 \rangle$ for the input $\langle 31, 41, 59, 26, 41, 58 \rangle$, also produces a correct output for every instance, we will then call it a correct sorting algorithm.

We will show how to prove the correctness of some simpler algorithms.

Secondly, an algorithm should be efficient, i.e., fast, and inexpensive.

Although algorithms are language independent, we will use Java, and Chapel, to implement some of those algorithms in projects so that we can run them in a computer to test its correctness, and witness its efficiency.
Features of algorithms

An algorithm has to be *finite*, i.e., it can be described with a finite amount of text.

An algorithm has to be *effective*, i.e., each step of an algorithm is mechanically feasible.

An algorithm must be *terminated* within a finite amount of time, e.g., get the bread done in 75 minutes.

The sequence of all the steps in an algorithm has to be *uniquely determined*, so a computer always knows what to do with *it*.

Practically speaking, we also want to have an *efficient* algorithm to work with, which “does not take too much time and space to complete”.

In fact, the gist of this course is to find a(n) (most) efficient algorithm to correctly solve a realistic problem.
What can an algorithm do?

An algorithm *specifies* a mechanical solution to a problem, and a program *implements* an algorithm in a specific language.

To send out an email via the Internet, we have to design algorithms to solve such problems as finding a shortest path. To find out information over the Web, we have to design a fast search algorithm to quickly find pages where particular information resides. To help us to identify those most useful pages, we have to sort them out by certain criteria, often referred as its *rank*.

For e-commerce application, we need to design encryption algorithms to keep such confidential information as credit card numbers, passwords, and bank statements private while making them accessible to authorized users.

We will discuss the mathematical basis of hacking in CS3780.
A few specific problems

1. Given a sequence of $n$ matrices $\langle A_1, A_2, \cdots, A_n \rangle$, we might want to calculate their product. In particular, if all the involved are $m \times m$ square matrices, how could we solve this problem efficiently? It depends on the model of the computation.

If we only have a sequential machine, and, if each matrix multiplication takes a unit time (It actually takes $m^3$ times units. See the next slide for the reason.), we have to spend $(n - 1)$ units of time to wrap it up.

On the other hand, if we have a parallel machine with at least $\log n$ processors, we can cut the whole thing into two piles and multiply them out in parallel. This way, it only takes $\log n$ units of time, although we still have to do the same work by multiplying $n - 1$ matrices.

We will see many applications of such a divide’n conquer technique in this course.
Lesson 1: Let’s play it out

Recall that, when multiplying two matrices, we multiply each row and column respectively. For example, for the following,

\[
\begin{pmatrix}
1 & 2 \\
3 & 4
\end{pmatrix} \times \begin{pmatrix}
5 & 6 \\
7 & 8
\end{pmatrix} = \begin{pmatrix}
19 & 22 \\
43 & 50
\end{pmatrix}
\]

we have, in particular,

\[
19 = 1 \times 5 + 2 \times 7.
\]

In general, given two \(m \times m\) matrices, \(A\) and \(B\), to get \(C = A \times B\), for all \(i, j \in [1, m]\), we have

\[
c_{i,j} = \sum_{k=1}^{m} a_{i,k} \times b_{k,j}.
\]

In the above example,

\[
c_{1,1} = a_{1,1} \times b_{1,1} + a_{1,2} \times b_{2,1} = 1 \times 5 + 2 \times 7 = 19.
\]

Thus, it takes \(m\) multiplications to get \(c_{i,j}\), and it takes a total of \(m^3\) multiplications to obtain \(C\), the product of \(A\) and \(B\).
A sequential algorithm

If we have one processor available, we can only multiply two matrices at a time.

\[ P = (((A_1 \times A_2) \times A_3) \cdots A_n). \]

Below shows an algorithm in pseudo code, which we will always use in the lectures.

SeqMatMult(A)
1. \( P = A_1 \)
2. \( i = 2 \)
3. while \( i \leq n \) do
4. \( P = P \times A_i \)
5. \( i = i + 1 \)
6. Return P

If it takes one time unit to multiply a pair of matrices, Step 4 takes a total of \( n - 1 \) time units. The time spent on the other steps is much less, thus can be ignored. 😊

**Question:** How many time units will it take to execute Steps 1, 2 and 6? How about Steps 3 and 5?
A parallel approach

If we have as many processors as we want, we could follow a parallel approach.

\[ P = (((A_1 \times A_2) \times (A_3 \times A_4)) \cdots (A_{n-1} \times A_n)). \]

We first assume that \( n = 2^m \), i.e., \( m = \log_2 n \).
Let \( A = \langle A_1, A_2, \cdots, A_n \rangle \),

\[ \text{ParaMatMult}(A) \]
1. If \( m = 0 \)
2. return \( A_1 \)
3. Else
4. \( M_1 = \langle A_1, A_2, \cdots, A_{2^{m-1}} \rangle \),
5. \( M_2 = \langle A_{2^{m-1}+1}, A_{2^{m-1}+2}, \cdots, A_{2^m} \rangle \).
6. \( B_1 = \text{ParaMatMult}(M_1) \)
7. \( B_2 = \text{ParaMatMult}(M_2) \)
8. Return \( B_1 \times B_2 \).

Notice that, in this case, both \( M_1 \) and \( M_2 \) contain exactly \( 2^{m-1} \) matrices.
How long does it take?

Let $N(m), m \geq 0$, be the number of time units it takes to multiply those $2^m$ matrices.

If $m = 0$, i.e., $n = 1$, there is no matrix multiplication for us to do. Thus, $N(0) = 0$.

In general, since the size of both $M_1$ and $M_2$ is $2^{m-1}$, by definition, it takes $N(m - 1)$ time units to multiply all the matrices in both piles, in Steps 6 and 7, simultaneously. Finally, we need one more multiplication to get the final result by multiplying $B_1$ and $B_2$ in Step 8.

Hence, we have that

\[
N(0) = 0
\]

\[
N(m) = N(m - 1) + 1, m \geq 1.
\]
How to figure out $N(m)$?

You should have learned such a process in a previous math course, e.g., MA 2250 MA for CS, or MA 2450 Math Reasoning.

To evaluate $N(m), m \geq 1$, it is clear that

\[
N(m) = N(m - 1) + 1 \\
= [N(m - 2) + 1] + 1 \\
= N(m - 2) + 2 \\
= \ldots \\
= N(m - k) + k \\
= \ldots \\
= N(m - m) + m \\
= N(0) + m = m \\
= \log_2 n.
\]

Thus, it takes just $\log n$ time units to do the same work if we have an ample supply of (at least $\log_2 n$) processors.
Lesson 2: I want to see...

We now know that the sequential approach makes $n - 1$ (a linear function) matrix comparisons, while the parallel one makes just $\log_2 n$ (a logarithmic function).

**Question:** Which one is faster?

The following shows the comparative relationship among a few functions.

![Graph showing comparative complexity](image)

Since, for $n \geq 3$, $\log_2 n < n - 1$, the parallel approach is faster than the sequential one.

**Question:** Which algorithm do you want to use? 😊
How about the general case?

When \( n \neq 2^m \), there must exist \( m \) such that \( 2^m < n < 2^{m+1} \). For example,

\[
2^7 = 128 < 131 < 256 = 2^8.
\]

Let \( T(n) \) be the time units that it takes to multiple all these \( n \) matrices, we have

\[
m = N(m) = T(2^m) < T(n) < T(2^{m+1}) = N(m+1) = m+1.
\]

On the other hand, since \( 2^m < n < 2^{m+1} \), we have \( m < \log_2 n < m + 1 \). In other words,

\[
m = \lfloor \log_2 n \rfloor, \text{ where } "\lfloor \rfloor" \text{ is the floor function.}
\]

As a result, we have

\[
\lfloor \log_2 n \rfloor < T(n) < \lfloor \log_2 n \rfloor + 1.
\]

Thus, such a parallel approach is in the order of \( \log_2 n \), much faster than the sequential approach, which takes a linear time, at the cost of additional processors.

We will talk, and walk, a lot more about parallel programming (Chapter 27), with the help of the Chapel language. Stay tuned...
2. Given a road map which provides the distance between each pair of adjacent cities, we might want to determine the shortest distance from one city to another (Google Maps). We will spend a whole chapter, Chapter 24, on this problem later on to solve this *shortest path* problem.

An important observation is that, once we find a shortest path between two cities, any resulting path between any two cities along this shortest path must be a shortest one as well. Thus, to look for a shortest path in between two cities, we will start with the shortest ones for cities in between.

This technique is referred to as the *greedy* approach, Chapter 16, which always seeks the best solution from a local perspective.

*In this case,* it also leads to a globally optimal solution in this case.
3. We often have to solve scheduling problems.

For example, given a resource, a classroom, a video equipment, a hotel room, and a bunch of requests, each of which comes with a starting time, $s_i$, and a finishing time, $f_i > s_i$, meaning, this request wants to use that facility during the period of $[s_i, f_i]$.

We call two requests compatible if they don’t overlap, i.e., no two requests want to use the same facility during the same time period. For example, no two classes should be scheduled in Rounds 207 during the same class period.

The goal for this interval scheduling problem is to find a maximum subset of compatible requests so that we can do the most with the given resource, e.g., put in as many classes into our classrooms.

Similarly, we might want to use as few classrooms as possible for all the classes. (Further discussion on Page 22)
4. There could be an extension for the interval scheduling problem: for each request, we can attach a weight $w_i$, representing such things as priority, profit, size, etc..

Now, our goal is not just to maximize the size of compatible requests, but maximize the total weight of such a compatible request subset, i.e., the sum of the weights included in this subset.

For example, which classroom scheduling will take in most students?

To make this to happen, between two classes that we could put into the same room during the same period of time, we will put in the larger class, where we use size of a class as its weight.

We will scratch such problems in Chapter 16, as well.
A couple of points

1. We notice that, if we set $w_i$ to 1 for all the requests, we have the original interval scheduling problem.

We would love to solve such a more general problem since its solution immediately leads to a solution to a weaker one.

Abstraction or generalization is important. 🙂

2. This scheduling problem is an example of the Knapsack problem, where, e.g., we want to maximize the value out of objects with a fixed number of containers.

3. We will see later on how to solve this weighted version recursively and, if time allows, also show how to solve it rather quickly by applying the dynamic programming technique, where we will trade space for time.
Optimization problems

The scheduling problem is also an example of the ever important optimization problems: given a collection of requests, we might want to \textit{minimize} the number of rooms (\textbf{min} problems); or given a certain collection of rooms, we want to \textit{maximize} the number of requests that we can schedule there (\textbf{max} problems).

We also have to solve \textbf{maximin} problems or \textbf{minimax} problems. For example, in VLSI design, sometimes we want to arrange a set of circuit components on a circuit board such that the length of the longest connecting wire (and hence time delay) is minimized.

Many many such interesting, but challenging, problems have emerged in both theory and practice. Unfortunately, they are all tough to crack. 😞
What are common?

Those problems have at least the following two things in common:

1. There are many candidate solutions, e.g., there are 20+ solutions for the sorting problems (Check out the course page for a list), but we only want the best one, usually the one that runs the fastest and, perhaps, the one that takes the least amount of memory.

2. The solution of those problems often depend on how the data is organized. In general, a data structure is a way to organize data in order to facilitate access and modification.

That is why CS 2381 on data structure has become a prerequisite of this course.

**Homework:** Exercises 1.1-1 and 1.1-5.
Easy problems and hard ones

In this course, we discuss how to solve problems, and in particular, we talk about efficient algorithms, i.e., fast(?) algorithms, measured with *how long it takes for an algorithm to solve the problem in terms of the input size*.

Obviously, we first have to figure out the issue that what we mean by an algorithm being *fast*, or *efficient*.

We will also have a look at another class of problems, the infamous *NP-Complete* problems, Chapter 34, for which no “fast” algorithm has been found, but no one has proved such an algorithm does not exist, either. 😊

Moreover, these problems share a remarkable property that, *if any one of them can be efficiently solved, all of them will be*. 😊
Big deal?

It is good to know their existence so that we won’t waste our time when asked to solve one.

It is also true that some of those NP-Complete problems are similar to problems for which we have already found efficient solutions, which is interesting, since it means a small change of the problem will lead to a big one in terms of algorithm efficiency.

Finally, if we know something belongs to the NP-complete class, then what we could do is not to find out the best solution, but rather a good, approximate, one.

We will check out, in CS 3780, the other side of the coin, i.e., problems that cannot be solved, no matter what technology you will use. 😞
An NP-complete problem

Considering a truck company with a central warehouse such that each day a truck loads up in the warehouse and travels around to deliver goodies, and it has to come back at the end of the day, not following the Southwest model.

To cut down the cost on gas, we want to find out a traveling plan for the truck so that the total distance that it has to travel is minimized.

This is called the traveling salesman problem and is known to be NP-Complete. Only approximate algorithms exist.

Homework: Check out the G-rated demo, and the R-rated Traveling salesman web site on the course page, then complete Exercise 1.1-4.
Algorithm as a technology

Even with the assumption that computers are infinitely fast and its space is free, we still have reasons to study algorithms, such as how to make sure that they will terminate at the end, besides providing correct solutions.

In practice, computers may be faster and faster, doubling its speed every 18 months (Moore’s Law) for the last 30 years or so, but won’t be infinitely fast. (In fact, it is going to end shortly. 😞 Check out course page) Memory may be larger and larger, but won’t be infinitely large. 😊 Thus, both computing time and space are restricted resources.

As a result, we have to come up with algorithms that are efficient in terms of both space and time.

You might want to read the CACM article on the exponential nature of Moore’s law, which you can also find on the course page.
Why is Moore’s law ending?

Consider a 1” × 1” chip that contains $n$ transistors, and the dimension of each such transistor is $d_1$, we have $(1/d_1)^2 = n$.

If we double the number of transistors, and assume the new dimension of a smaller transistor is $d_2$, by the same token, we have $(1/d_2)^2 = 2n$.

As a result, $d_2 = \left(\frac{1}{2}\right)^{\frac{1}{2}} d_1$. (Why is this the case?)
Similarly, $d_3 = \left(\frac{1}{2}\right)^{\frac{1}{2}}d_2 = \left(\frac{1}{2}\right)^{\frac{2}{2}}d_1$

In general, after the $m^{th}$ doubling, $m \geq 1$, the dimension of such a transistor will be

$$d_{m+1} = \left(\frac{1}{2}\right)^{\frac{m}{2}}d_1 = 0.5^{\frac{m}{2}}d_1,$$

The function $0.5^x$ comes down very quickly.

This “doubling law” thus has to come to an end soon. 😃

Moreover, we cannot deal with the ever increasing collective heat that all these transistors will generate. (Watch the “really hot” Video on the course page)

Check out the other Nature article “on its way out”, also on the course page.
What should we do?

The alternative way is to put multiple processors (cores), instead of one, on a chip. This multi-core technology leads to quite different a ball game: concurrent and parallel programming.

For example, when two cars move in the same lane, they are running concurrently; but if they are moving in multiple lanes, they are running in parallel.
Parallel programming

We have to study and apply algorithms to let multiple processors work together to get things done (parallel processing and management), and let multiple processes get access to a shared memory (memory management) without interfering with each other.

We scratched the surface of parallel programming in *CS 3600 Database* with a few practical issues, such as last update.

As we will see later in this course, it is truly challenging to ensure that different pieces will run in *the intended order* so that we get the correct result at the end, and their analysis.

Related issues, especially those on management and scheduling, will be further studied in *CS 4310 Operating Systems*. The very reason of having this OS course is the *multi-processor*, and a *time sharing*, nature in all the modern computers.
What will we do here?

We will devote quite some time, *in this course*, to the *theory* of parallel programming in Unit 6.

It is pretty common to come up with multiple processors for a computer these days. For example, the A16 chip inside iPhone 14 Pro comes with six cores, while *Frontier*, the top computer on the *Top 500* list, dated November 2022, comes with 8,730,112 cores.

Nevertheless, parallel computing does have a bottleneck, *Amdahl’s law*, in the sense that, beyond which, nothing can be gained no matter how many processors you throw in. ☹️

We will also get our hands dirty by playing with *Chapel*, a parallel programming language, to see when, and how, parallel programming will dramatically speed up processing. 😊
Algorithm efficiency

Sequential algorithms designed to solve the same problem can differ dramatically in terms of their efficiency.

For example, we will soon discuss two solutions for the sorting problem: *insertion sort* and *merge sort*. Insertion sort takes about $c_1 n^2$ time to sort a list containing $n$ elements, while Merge sort only takes $c_2 n \log n$.

In the above, both $c_1$ and $c_2$ are constants. Although $c_1$ is often less than $c_2$, when $n$ gets reasonably large, such constants do not play a role in the algorithm efficiency.

In fact, beyond a certain point, the merge sort will always run much faster than the insertion sort. 😊

**Question:** How come?
Lesson 3: Don’t trust Dr. Shen

Assume that computer A and computer B execute at the speed of 1,000 MIPS and 10 MIPS, respectively, and assume that insertion sort and merge sort algorithms take $2n^2$ and $50n \log n$ comparisons, respectively.

To make our points even clearer, let’s run the slower insertion sort in a faster computer, and the faster merge sort on a slower one.

To sort 64 numbers, the faster computer A, running the slower insertion sort, takes

\[
(2 \times 64^2)/10^9 = 8.192 \times 10^{-6} \text{ seconds;}
\]

while the slower computer B, running the faster merge sort, takes

\[
50 \times 64 \times \log(64)/10^7 = 1.92 \times 10^{-3} \text{ seconds.}
\]

Thus, Insertion sort runs faster in this case.
A little conclusion

However, to sort one million numbers, computer A, running insertion sort, takes

$$2 \times (10^6)^2 / 10^9 = 2,000 \text{ seconds;}$$

while Computer B, running merge sort, takes

$$(50 \times 10^6 \times \log 10^6) / 10^7 \approx 100 \text{ seconds.}$$

Thus, the faster merge sort algorithm runs on the slower computer B 20 times faster than what takes the insertion sort to run on the faster computer A.

This fact will be clearer further when we compare the two on even bigger numbers.

That’s why we won’t compare algorithms in terms of absolute time.

Homework: Exercise 1.2-2
The system performance depends on algorithms as much as on hardware. Moreover, algorithms are also related to other technologies, since they are widely used in designing hardware, various GUI interfaces, coming up with various network routing strategies.

In fact, algorithms and their analysis run in every aspect of computer science. It is indeed often stated that computer science is the science about algorithms.

It addresses the fundamental issue that how computers work to get things done.

Hence, having a solid base of algorithmic knowledge and technique is critically important to be a computer science major, starting with CS 2010 Computing Fundamentals.

**Homework:** Problem 1-1.
A summary...

... goes like this:

![Diagram showing algorithm design process]

We have learned quite a bit about algorithm design in previous courses, and practiced their implementation with Project 1.

To get ready to analyze, and compare, time complexities of algorithms, let’s review a little about what you should have learned in MA 2450.