In informally, an *algorithm* is any well-defined procedure that takes some input(s) and produces an output. We will see a formal definition in CS3780 later on.

Way back in the Fundamental course, we once compared an algorithm to a recipe, in the sense that a recipe, often with a name, also specifies its input (ingredients), and a procedure that a chef follows to turn all the ingredients into an output (a dish), tasty or not.

We certainly have talked about a lot of algorithms in between, functions in Php, methods in Java, each comes with a *name*, a list of *parameters*, a type of an *output*, and a *procedure* that turns these inputs to the output.
A recipe I once used

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

The stuff: 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.

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 is ready.

With this recipe, we specify what to process, how to do it, and what we will get at the end.

This is exactly what an algorithm should do.
A more serious one

An algorithm is often used to solve a computational problem. For example, the problem of sorting, i.e., putting things into order, arises frequently in programming and our lives.

Sorting finds a wide range of applications. No phone book is useful if it is not sorted by, e.g., the last name. You also see a sorting whenever you Google. We certainly talked about sorting in database.

The sorting problem also provides a nice example of algorithm analysis, i.e., how much work we have to do to sort out a bunch of data, data comparison and/or movement for this case.

There are quite a few such sorting algorithms, it thus makes sense to find the best sorting algorithm, fastest and/or cheapest, often in terms of the size of the data.
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: \( \langle a_1, a_2, \cdots, a_n \rangle \), where duplicate is allowed

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

In general, an *instance of a problem* consists of an input, together with whatever constraints such an input must satisfy to compute a solution.

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

First and foremost, an algorithm has to be *correct*, namely, for every input instance, an algorithm has to halt with a correct output. We then say that a correct algorithm *solves* the associated problem.

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

Although algorithms are language independent, we will use Java to implement some of those algorithms so that we can run them in a computer to test its effectiveness.

We will also show how to prove the correctness for some simpler algorithms.
What can an algorithm do?

Every computer program is based on an algorithm.

To send out an email via the Internet, we have to design algorithms to solve such problems as finding a short(est) 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.

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.

It is also behind something sinister: We heard about computer hacking all the time, and will discuss its mathematical basis 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 square matrices, say $m \times m$, 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 $n^3$ times units.), we have to spend $(n - 1)$ units of time to wrap it up. On the other hand, if we have a parallel machine, 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.
A sequential approach

If we have one processor available, we can only multiply two matrices at a time. Below shows an algorithm in pseudo code.

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

If it takes one time unit to multiply a pair of matrices, Step 4 takes $n - 1$ 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 Step 3? How about Step 5?
A parallel approach

If we have as many processors as we want, we could follow a parallel approach. We first assume that $n = 2^m$, i.e., $m = \log_2 n$. Let $A = \langle A_1, A_2, \cdots, A_n \rangle$,

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 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, respectively.

Since the multiplication of both parts are independent of each other, $B_1$ and $B_2$ can be obtained simultaneously in Steps 6 and 7. We do 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 it in a previous math course, e.g., MA2250 or MA3320.

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

\[
\begin{align*}
N(m) & = N(m - 1) + 1 \\
& = [N(m - 2) + 1] + 1 \\
& = N(m - 2) + 2 \\
& = [N(m - 3) + 1] + 2 \\
& = N(m - 3) + 3 \\
& = \ldots \\
& = N(m - m) + m \\
& = N(0) + m = m \\
& = \log_2 n.
\end{align*}
\]

Thus, it takes just $\log n$ time units to do the same work if we have an ample supply of processors.
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 multiply all these \( n \) matrices, we have
\[ m = N(m) < T(n) < N(m + 1) = m + 1. \]

On the other hand, based on 
\[ 2^m < n < 2^{m+1}, \]
we have 
\[ m < \log_2 n < m + 1. \]
In other words,
\[ m = \lfloor \log_2 n \rfloor. \]

As a result, we have
\[ \lfloor \log_2 n \rfloor < T(n) < \lceil \log_2 n \rceil + 1. \]

Thus, such a parallel approach is \textit{in the order of} \( \log_2 n \), much faster than the sequential approach, which takes a linear time.

We will go through various issues of such \textit{parallel processes} in a later chapter.
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 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, 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.
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, 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 want to use the same room during the same period of time, we must put in the larger class.
A couple of points

1. We notice if we set $w_i$ to 1 for all the requests, we then 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 is important.

2. This scheduling problem is an example of the Packaging problem, where we put objects into containers to maximize a certain criterion.

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 such problems have emerged in both theory and practice.
What are common?

Many of those problems have at least the following two things in common:

1. There are many candidate solutions, e.g., there are over 30 solutions for the sorting problems, 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 CS2381 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 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.
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.

In CS3780, we will check out 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.

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, 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 and provide 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.) 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 Communication ACM article on the exponential nature of Moore’s law.
It has to stop....

1. There is a physical limit as how many transistors that we can put down on a one square inch chip.

Consider such a $1 \times 1$ chip that contains $n$ transistors, and the dimension of each such transistor is $d_1$, thus, $(1/d_1)^2 = n$.

If we double the number of transistors on this chip, and assume the new dimension of such a smaller transistor is $d_2$, by the same token, we have $(1/d_2)^2 = 2n$. As a result, $d_2 = (1/\sqrt{2})d_1$. After the $m$th doubling, the dimension of such a transistor will be $d_{m+1} = \left(\frac{1}{2}\right)^\frac{m}{2} d_1$, $m \geq 0$.

2. We cannot deal with the ever increasing collective heat that all these transistors will generate. (Watch the “really hot” Video)

Thus, Moore’s law will indeed come to an end in the near future. Check out the other Nature article “on its way out”.

24
What else?

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.

Before long, we have to study algorithms based on this new concept: how do we let multiple processors work together to get things done (parallel processing and management), and how we let multiple processes get access to a shared memory (memory management) without interfering with each other, etc..

We scratched the surface in CS3600 (Data base) and will further study these issues in CS4310 (Operating Systems).

In fact, the only reason that we teach this OS course is because of the multi-processor structures in all the modern computers.
Algorithm efficiency

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*. The insertion sort takes about $c_1 n^2$ time to sort a list containing $n$ elements, while the 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 $c_2$, we will see later that 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.

**Homework:** Exercise 1.1-2.
You don’t believe me?

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.

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

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

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 even clearer when we compare the two on even bigger numbers.

**Homework:** Exercises 1.2-2 and 1.2-3.
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 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.

**Homework:** Problem 1-1.