The purpose of CS 341 is to learn how to design and analyze efficient algorithms. This is one of the most important skills to master for computer scientists. It’s also something that can be done simply with pen and paper, and does not require any advanced knowledge of any programming language. To get an idea of how this course will proceed, let’s start with an example.

1. FIRST EXAMPLE: THE CONVEX HULL PROBLEM

Our first problem is one of the central problems in computational geometry: the convex hull problem.

**Problem 1.** Given a set of $n$ points in the plane, find their convex hull—the smallest convex set that contains all $n$ points.

For example, here is a set of 7 points and their convex hull:

Our goal will be to design an efficient algorithm for solving the convex hull problem. Before we do so, let’s talk about the problem itself a little bit more.

First, the convex hull problem statement is rather abstract—after all, which software developer in the real world would ever directly encounter the convex hull problem? This is by design: our goal in this class is to introduce techniques that are useful throughout many different areas of computer science. As such, throughout the course we will only consider problems that are as simple as possible; after this course, you will see that underneath many complex real-world computational problems lie exactly the problems that we see in this class or slight variations of them. (For example, the convex hull problem itself is at the heart of multiple fundamental problems in image processing, optimization problems, mapping applications, etc.)

Second, there is one convention that we used in the problem statement and will use throughout the course: the variable $n$ will be reserved exclusively for the most natural parameter of interest that represents the “size” of the input. Our goal will be to design

---

Note that this example is provided only to illustrate and motivate the topics that we will see in the rest of the class. You will not be responsible for understanding the details of this example in the assignments or the exam—though you are more than welcome to explore it in more detail if you find it interesting!
algorithms whose running time is as small as possible as a function of $n$. (We’ll talk about this convention and related topics in the next lecture when we formalize the model of computation a bit more precisely.)

Ok, so now we’re ready to tackle our challenge: Can you design an efficient algorithm that solves the convex hull problem?

1.1. A first algorithm. The first step in addressing all the algorithm challenges we will see throughout the course is to start by designing some algorithm that solves the problem, and not to worry too much about its efficiency. For the current challenge, we get a good hint about one possible algorithm by considering an alternative definition of the convex hull of a set of points.

**Definition 1.1.** The convex hull of a set $X$ of points is the polygon whose sides are the lines $\ell$ which

1. pass through at least 2 of the points in the set $X$, and
2. for which no points in $X$ lie on one of the sides of $\ell$.

This definition leads us to the following simple algorithm.

**Algorithm 1:** SimpleConvexHull($X$)

```
for every pair of points $r, s \in X$ do
    $\ell \leftarrow$ a line passing through $r$ and $s$;
    if every point in $X$ lies to one side of $\ell$ then
        Add $\ell$ to the convex hull;
```

Note that this algorithm is presented in rather high-level pseudocode. In the rest of the course, we will usually specify the algorithm in a bit more detail to make sure that we have a clear and precise algorithm, and to make its time complexity explicitly clear. (For instance, how do we define the lines in the algorithm? And, most importantly, how do we test whether all the points in $X$ lie on one side of the line?) But we will never aim for a completely formal definition/implementation of the algorithm: our goal is always to be able to present an algorithm in a way that we can

- verify the correctness of the algorithm,
- determine its running time, and
- understand clearly how it works.

We will discuss all these points again in the next lecture. But for now, can we determine the efficiency of the SimpleConvexHull algorithm? Assuming that all the operations on pairs of points take constant time and that we use a straightforward implementation of the test in the if condition, we have a total runtime of $O(n^3)$.

\textsuperscript{2}Hint: Cross products!
**Theorem 1.2.** The `SimpleConvexHull` algorithm solves the convex hull problem in time $O(n^3)$.

Can we do better? Yes!

1.2. **Jarvis march.** If we spend a bit of time with the `SimpleConvexHull` algorithm and some examples, we eventually notice a natural improvement: if we have found a line $\ell$ that goes through $r$ and $s$ in $X$ and is in the convex hull of $X$, then we can find the next line on the convex hull by “rotating” the line $\ell$ on the point $s$ until it hits another point $t$ in $X$.

![Diagram of Jarvis march](image)

The algorithm obtained by using this idea is known as *Jarvis’ march*, or the *gift wrapping algorithm*.

**Algorithm 2: JarvisMarch($X$)**

```
\begin{algorithm}
    \textit{r} \leftarrow \text{left-most point in } X; \\
    \textit{s} \leftarrow \text{next point in } X \text{ obtained by rotating vertical line from } r; \\
    \text{Add } (r, s) \text{ to the convex hull;}
    \textbf{while } s \text{ is not the left-most point of } X \text{ do}
    \quad \textit{t} \leftarrow \text{next point in } X \text{ obtained by rotating line } (r, s); \\
    \quad (r, s) \leftarrow (s, t); \\
    \quad \text{Add } (r, s) \text{ to the convex hull;}
\end{algorithm}
```

As before, more details are required to formalize this algorithm. As an exercise, you should verify that finding the left-most point in $X$ and finding the point $t$ at each iteration of the while loop can both be accomplished in $O(n)$ time since they are equivalent to the problem of finding the minimum element of a set. The Jarvis march algorithm therefore gives a significant improvement over our simple convex hull algorithm.

**Theorem 1.3.** The `JarvisMarch` algorithm solves the convex hull problem in time $O(n^2)$.

In fact, we can even say something stronger: if we let $h$ denote the number of vertices on the convex hull of $X$, then the time complexity of the `JarvisMarch` algorithm is $O(n \cdot h)$.

Can we do better? Once again, yes!

1.3. **Divide and conquer.** A very powerful algorithm design method that we will revisit in this course is *divide and conquer*: split the original problem up into smaller subproblems, solve the subproblems separately, and merge the solutions. This approach works very well for the convex hull problem: we split the set $X$ into two halves $X_0$ and $X_1$, find the convex hulls of these subsets, and find the upper and lower *bridges* that are needed to connect the two convex hulls.
Algorithm 3: ConvexHullDC(X)

\[
\begin{align*}
X_0 &\leftarrow \text{left half of } X; \\
X_1 &\leftarrow \text{right half of } X; \\
H_0 &\leftarrow \text{ConvexHullDC}(X_0); \\
H_1 &\leftarrow \text{ConvexHullDC}(X_1); \\
(ub, lb) &\leftarrow \text{upper and lower bridges of } H_0, H_1; \\
\text{return} &\text{ merged convex hull of } H_0 \text{ and } H_1 \text{ using } ub, lb;
\end{align*}
\]

As in our earlier example, more details are needed to complete the algorithm and are left as an exercise. The step of merging the convex hulls using the upper and lower bridges is straightforward, but finding the bridges is a little bit more subtle. This can be done by starting with the edge \(e\) connecting the right-most point in \(X_0\) to the left-most point in \(X_1\) and “walking” the edge up and down as far as possible.

Once we complete the algorithm, we see that finding the upper and lower bridges and merging the convex hull can be done in linear time. The time complexity of the ConvexHullDC algorithm therefore satisfies the recurrence relation
\[
T(n) = 2T\left(\frac{n}{2}\right) + O(n).
\]
This is the same recurrence relation as for merge sort, so we end up with a final time complexity of \(O(n \log n)\).

**Theorem 1.4.** The ConvexHullDC algorithm solves the convex hull problem in time \(O(n \log n)\).

Can we do better? Yes and no.

1.4. Other algorithms. Having gone from an initial algorithm that solves the convex hull problem in time \(O(n^3)\) to one that has time complexity only \(O(n \log n)\), it’s natural to ask if we can improve that result even a little bit more, ideally to obtain an algorithm that runs in time \(O(n)\). This is not possible in any setting where sorting takes time \(\Omega(n \log n)\):

**Theorem 1.5.** In any setting where sorting \(n\) numbers takes time \(\Omega(n \log n)\), then every algorithm for solving the convex hull problem also has time complexity \(\Omega(n \log n)\).

**Proof.** The theorem can be established using the idea of reduction: we show that an efficient convex hull algorithm can also be used to sort numbers efficiently. The idea of this argument is that if wish to sort a list of \(n\) numbers \(a_1, a_2, \ldots, a_n\), we can do this by
(1) Creating the set \( X = \{(a_1, a_1^2), (a_2, a_2^2), \ldots, (a_n, a_n^2)\} \),
(2) Finding the convex hull of \( X \),
(3) and extracting the sorted list of numbers by following the convex hull of \( X \).

Steps 1 and 3 can be shown to both take time \( O(n) \), so if we have a convex hull algorithm with complexity \( o(n \log n) \), we would also achieve the same runtime for sorting as well. □

This theorem shows that, in a sense, the ConvexHullDC algorithm is optimal. But in fact we can push our exploration of the problem even further. What if we assume that the points in \( X \) are not stored in an arbitrary order but instead are sorted according to their first coordinate? In that case, the convex hull problem can be solved more efficiently.

**Theorem 1.6.** The convex hull problem can be solved in time \( O(n) \) when the set \( X \) of input points is sorted according to the first coordinate.

The proof of this theorem is left as a challenge. A small hint is that you may want to consider an algorithm that finds the upper and lower parts of the convex hull separately.

Another natural question is raised by comparing the bounds of the JarvisMarch and ConvexHullDC algorithms: the refined analysis of the former shows that it runs in time \( O(nh) \) where \( h \) is the number of points on the convex hull, whereas the latter runs in time \( O(n \log n) \) regardless of the value of \( h \). Thus, the algorithms are in some sense incomparable: for some inputs—those corresponding to sets with convex hull that hit \( o(\log n) \) points—the JarvisMarch algorithm is better; for other sets, the ConvexHullDC algorithm is much faster. Is there a better algorithm that does at least as well as both of these algorithms for all inputs? Yes!

**Theorem 1.7.** The convex hull problem can be solved in time \( O(n \log h) \) when the set \( X \) of \( n \) input points has a convex hull of size \( h \).

The proof of this theorem is again left as a challenge. Interested readers can find more details about the solution to the challenge by searching for the Kirkpatrick–Seidel ultimate convex hull algorithm or Chan’s algorithm.

2. **Overview of the class**

2.1. **Goal.** By the end of this class, you should be able to tackle a wide variety of computational problems just like we did for the Fibonacci sequence above so that you can analyze the time complexity of algorithms, design better algorithms in many cases, and recognize when some problems are intractable.

2.2. **Topics.** The topics we will cover to help you achieve this goal are grouped as follows:

(1) Analysis of algorithms
(2) Greedy algorithms
(3) Dynamic programming
(4) Graph search algorithms
(5) NP-completeness
2.3. **Resources.** All the course information and relevant links are found on the course web page

www.student.cs.uwaterloo.ca/~cs341

The lecture notes and tutorial notes will be posted throughout the term on that page. We will also use Piazza for discussions.

The main textbook for the class is Cormen, Leiserson, Rivest, and Stein’s *Introduction to Algorithms* (3rd edition). It is not required but is highly recommended. Another great alternative textbook that is also highly recommended is *Algorithms* by Dasgupta, Papadimitriou, and Vazirani. There are also other alternatives that are on reserve at the library; see the website for details.

2.4. **Marking scheme.** The marks breakdown for the class is as follows:

- 35%: Weekly assignments
- 25%: Midterm
- 40%: Final exam

All sections of the class have the same assignments, midterm, and final exam. Two of the assignments are programming assignments; the rest are written assignments.

2.5. **How to succeed in CS 341.** There are a few tricks that will help you succeed in CS 341 and master all the tools and techniques we introduce in the class.

- Bring pen and paper to class, and nothing else. Lecture notes will be provided so you should not worry about taking complete notes in class. But there will be many pauses in the lectures where you will be asked to think about a problem before the answer is revealed. The best way to do this will be in pairs, with pen and paper. Doing so actively (whether you do find the answer or not) will help you learn the material much more effectively than if you’re just here to listen.
- Complete the homeworks *by yourself*. This will be the single most important factor determining your success in the course.
- Attend tutorial sessions. This is a new component for CS 341. The idea is to have extra practice at solving problems, on top of the assignments, in a setting where you can also get guidance and support from IAs. Take advantage of them!
- Ask questions! We are here to help. You are encouraged to discuss the material with other students, to ask questions on Piazza, to visit instructors or TAs during office hours, etc.