In this lecture, we examine two of the most powerful tools in an algorithm designer’s kit: reduction and recursion.

1. The 2SUM problem

Let’s examine a simple problem.

**Definition 3.1 (2SUM).** Given an array $A \in \mathbb{Z}^n$ of $n$ integers and an integer $m \in \mathbb{Z}$, return a pair of indices $i, j \in \{1, 2, \ldots, n\}$ such that $A[i] + A[j] = m$ if any such pair exists, and $\perp$ if no such pair exists.

The following algorithm solves the 2SUM problem.

**Algorithm 1: Simple2Sum($A, m$)**

```plaintext
for $i = 1, \ldots, n$ do
    for $j = i, \ldots, n$ do
        if $A[i] + A[j] = m$ return $(i, j);
    return $\perp$;
```

The claim that Simple2Sum solves the 2SUM problem requires a proof, but that proof is very simple: if there exist $k, \ell$ for which $A[k] + A[\ell] = m$, then we also have that $A[\ell] + A[k] = m$ so in the iteration of the inner loop with $i = \min\{k, \ell\}$ and $j = \max\{k, \ell\}$, the if test will be satisfied and the algorithm returns a valid solution. And if no such $k, \ell$ pair exists, then the if test is never satisfied, so the algorithm always returns $\perp$.

The time complexity of the Simple2Sum algorithm is also established with a simple argument, but we have to be a little bit careful if we want the argument to hold in the Word RAM model as specified in the last lecture. In this problem, what we really care about in terms of time complexity for this problem is the number of basic operations (addition, incrementing, and comparison) on integers. We do this in the Word RAM model by setting the width $w$ of words to be large enough that each integer in the input array $A$ fits in a single word.

**Proposition 3.2.** The Simple2Sum algorithm has time complexity $\Theta(n^2)$ in the Word RAM model when the word size $w$ is large enough that each integer in the input array $A$ fits in a single word.

**Proof.** Recall that we measure the time complexity of our algorithms in the worst-case setting. This worst-case is attained on inputs that return $\perp$, since for these inputs the two
for loops run to completion without interruption. In this case, the number of additions performed by the algorithm is
\[
n + (n-1) + \cdots + 2 + 1 = \frac{n(n-1)}{2}.
\]
For every \(n \geq 2\), we have \(\frac{n^2}{4} \leq \frac{n(n-1)}{2} \leq n^2\) so the algorithm performs \(\Theta(n^2)\) additions and the same bound also holds for the total number of increments and compare operations. □

Can we do better? Yes! Let’s break down what the algorithm is doing in detail. For each index \(i \in \{1, 2, \ldots, n\}\), the inner loop is trying to determine if there is an integer \(A[j]\) in \(A\) for which \(A[i] + A[j] = m\), or, to rephrase the same statement: if the integer \(m - A[i]\) is in the array.

As it turns out, we know a very efficient method for determining whether an integer (such as \(m - A[i]\)) is in an array: binary search! This, of course, only works when the array is sorted, but that’s something we can do in the algorithm as well. The resulting algorithm is as follows.

**Algorithm 2:** \(2\text{Sum}(A, m)\)

\[
B \leftarrow \text{Sort}(A);
\]
\[
\text{for } i = 1, \ldots, n - 1 \text{ do}
\]
\[
j \leftarrow \text{FindBinarySearch}(B, m - B[i]);
\]
\[
\text{if } j \neq \perp \text{ then return } (\text{Find}(A, B[i]), \text{Find}(A, B[j]));
\]
\[
\text{return } \perp;
\]

What we have just done looks simple, perhaps even obvious, yet it is a great example of an extremely powerful algorithmic technique: reduction. Specifically, we reduced the 2SUM problem to the problem of finding a given integer in an array. And since we already know how to solve the latter problem, we can use the algorithm for that problem to solve 2SUM as well.

**Reduction idea:** Use known algorithms to solve new problems.

We’ll see many other examples where the reduction technique proves very useful—and it will even return with a starring role in the last module of this class, when we tackle NP-completeness. But for now, let’s continue our work and establish the correctness and time complexity of the 2Sum algorithm.

**Proposition 3.3.** The 2Sum algorithm solves the 2SUM problem.

**Proof.** Consider first the case where there exist indices \(k, \ell\) for which \(A[k] + A[\ell] = m\). Then there exist indices \(k', \ell'\) for which \(B[k'] + B[\ell'] = m\). We can rewrite the identity as \(B[\ell'] = m - A[k']\), so for the iteration of the for loop where \(i = k'\), the integer \(m - B[i]\) is in \(B\) and the FindBinarySearch algorithm returns \(\ell'\) or another index \(\ell''\) for which \(B[\ell''] = B[\ell']\) and 2Sum returns indices \(i^*, j^*\) for which \(A[i^*] + A[j^*] = B[k'] + B[\ell'] = m\).

Consider now the case where for every indices \(i, j\) we have \(A[i] + A[j] \neq m\). Then it is also true that for every indices \(i, j\) we have \(B[i] + B[j] \neq m\) or, equivalently, \(B[j] \neq m - B[i]\). So for each value of \(i\), the integer \(m - B[i]\) is not in \(B\), FindBinarySearch returns \(\perp\), and so does 2SUM. □
Proposition 3.4. When Sort has time complexity $O(n \log n)$, FindBinarySearch has time complexity $O(\log n)$, and Find has time complexity $O(n)$, the 2Sum algorithm has time complexity $O(n \log n)$.

Proof. The initial call to Sort has time complexity $\Theta(n \log n)$. The FindBinarySearch algorithm is called at most $n$ times for total time complexity $n \cdot O(\log n) = O(n \log n)$, and the Find algorithm is called at most twice, for an additional time complexity $O(n)$. $\square$

Can we do even better? It takes $\Theta(n)$ time just to read the integers stored in the array $A$, so the best we can probably hope for is to remove the extra $\log n$ term in our runtime. But instead of trying to do this, let’s see if we can use the Reduction idea to solve even more complex problems.

2. The 3SUM problem

Definition 3.5 (3SUM). Given an array $A \in \mathbb{Z}^n$ of $n$ integers and an integer $m \in \mathbb{Z}$, return three indices $i, j, k \in \{1, 2, \ldots, n\}$ such that $A[i] + A[j] + A[k] = m$ if any such triple exists, and $\bot$ if no such triple exists.

We can again define a simple algorithm to solve the 3SUM problem.

$$
\text{Algorithm 3: Simple3Sum}(A, m) \\
\text{for } i = 1, \ldots, n \text{ do} \\
\quad \text{for } j = i, \ldots, n \text{ do} \\
\quad \quad \text{for } k = j, \ldots, n \text{ do} \\
\quad \quad \quad \text{if } A[i] + A[j] + A[k] = m \text{ return } (i, j, k); \\
\text{return } \bot;
$$

This algorithm is not so efficient. It has time complexity $\Theta(n^3)$, and we would like to do better. We can again use the Reduction technique to find a better algorithm. One option is to again try to reduce to the FIND problem. Or, since we just designed an efficient 2SUM algorithm, perhaps we can reduce 3SUM to 2SUM instead? Let’s find out by again fixing an index $i$ and seeing what the two inner loops are doing. They’re trying to find out if there are indices $j, k$ for which


This is an instance of the 2SUM problem!

$$
\text{Algorithm 4: 3Sum}(A, m) \\
\text{for } i = 1, \ldots, n \text{ do} \\
\quad (j, k) \leftarrow 2\text{Sum}(A, m - A[i]); \\
\quad \text{if } (j, k) \neq \bot \text{ then return } (i, j, k); \\
\text{return } \bot;
$$

Notice how simple the resulting algorithm is. As a bonus, the analysis of the time complexity of this algorithm is also very simple.

Theorem 3.6. The 3SUM problem can be solved by an algorithm with time complexity $\Theta(n^2 \log n)$. 
Proof. When the 3SUM algorithm calls the algorithm 2SUM with time complexity $\Theta(n \log n)$ that we designed in the previous section, its total time complexity is $n \cdot \Theta(n \log n) = \Theta(n^2 \log n)$. □

Can you do even better? With some work, you may be able to design an algorithm that solves the 3SUM problem in time $\Theta(n^2)$. Determining whether we can do even better is one of the major open problems in algorithms research today: it was only very recently that researchers were able to show that 3SUM can be solved by an algorithm with time complexity $o(n^2)$, and whether or not there is an algorithm that solves 3SUM with time complexity $O(n^\gamma)$ for some $\gamma < 2$ remains unknown.

3. Recurrence trees

Recursion is a special type of reduction, where we reduce the original problem to the same problem, but on a smaller input. This is a powerful technique, as you have already seen in previous classes when considering the MERGESORT algorithm:

**Algorithm 5: MERGESORT(A)**

```plaintext
if $n = 1$ return;
MERGESORT(A[1,...,[n/2]]);
MERGESORT(A[[n/2]+1,...,n]);
MERGE(A[1,...,[n/2]], A[[n/2]+1,...,n]);
```

When MERGE is an algorithm that merges the two input arrays in time $\Theta(n)$, what is the time complexity of MERGESORT? If we let $T(n)$ denote the time complexity of the algorithm on arrays with $n$ entries, we find that

$$T(n) = T([n/2]) + T([n/2]) + \Theta(n).$$

To simplify the analysis, let’s assume that $n$ is a power of 2. Then we have

$$T(n) = 2T(\frac{n}{2}) + \Theta(n).$$

To determine the solution to this recursion, the most natural approach is to draw the recursion tree and write down how much time is spent on each level of the tree:

```
T(n)  \hspace{1cm} cn = \Theta(n) \text{ work}
  \ / \ \ /
 T(\frac{n}{2}) T(\frac{n}{2})  \hspace{1cm} 2 \cdot c(\frac{n}{2}) = \Theta(n) \text{ total work}
 / \ / \ / \ / \ /
 T(\frac{n}{4}) T(\frac{n}{4}) T(\frac{n}{4}) T(\frac{n}{4})  \hspace{1cm} 4 \cdot c(\frac{n}{4}) = \Theta(n) \text{ total work}
 / \ / \ / \ / \ / \ / \ /
 T(\frac{n}{8}) T(\frac{n}{8}) T(\frac{n}{8}) T(\frac{n}{8}) T(\frac{n}{8}) T(\frac{n}{8}) T(\frac{n}{8}) T(\frac{n}{8})  \hspace{1cm} 8 \cdot c(\frac{n}{8}) = \Theta(n) \text{ total work}
```

The recursion tree has depth $\log n$ and $\Theta(n)$ time is spent at each level of the tree on completing the merges. The total time complexity of the MERGESORT algorithm is therefore

$$T(n) = \Theta(n \log n)$$

\[1\]Here and throughout this course, logarithms without subscripts are over base 2.
Many other recursions can also be solved using the same recursion tree approach. For example, what if we (magically) allowed the Merge algorithm to run in a single unit of time? Then the recursion defining the time complexity of the MergeSort algorithm would be

$$T(n) = 2T\left(\frac{n}{2}\right) + 1$$

and the recursion tree would now look like this:

```
T(n)  1 work
  / \     /
 T(n/2) T(n/2) 2 \ 2 total work
 / \ / \ / \
T(n/4) T(n/4) T(n/4) T(n/4)  4 \ 4 total work
 / \ / \ / \
T(n/8) T(n/8) T(n/8) T(n/8)  8 \ 8 total work
```

This tree again has depth $\log n$, so the total time complexity in this case is

$$T(n) = 1 + 2 + 4 + 8 + \cdots + \frac{n}{2} + n = 2n - 1 = \Theta(n).$$

One more variant: what if we proceed recursively but now divide the array in three instead of two pieces?

**Algorithm 6: TriMergeSort(A)**

```python
if n == 1:
    return;
TriMergeSort(A[1, ..., [n/3]]);
TriMergeSort(A[[n/3] + 1, ..., [2n/3]]);
TriMergeSort(A[[2n/3] + 1, ..., n]);
Merge(A[1, ..., [n/3]], A[[n/3] + 1, ..., [2n/3]], A[[2n/3] + 1, ..., n]);
```

Let Merge again be an algorithm with time complexity $\Theta(n)$. When $n$ is a power of 3, the time complexity of the TriMergeSort algorithm is

$$T(n) = 3T\left(\frac{n}{3}\right) + \Theta(n).$$

The recursion tree for this recurrence is:

```
T(n)  cn = \Theta(n) work
  / \     /
 T(n/3) T(n/3) T(n/3) 3 \ 3 total work
 / \ / \ / \
T(n/9) T(n/9) T(n/9) T(n/9) 9 \ 9 total work
```

The tree has depth $\log_3 n$, so the time complexity of TriMergeSort is

$$T(n) = \Theta(n \log_3 n) = \Theta(n \log n),$$

the same (asymptotically) as MergeSort!
4. **Master theorem**

We could continue exploring various different examples using the recursion tree method. But let’s be a bit more ambitious and try to consider a general scenario that captures many individual examples at once. Let $T$ be defined by the recurrence

$$T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^c)$$

for some constants $a > 0$, $b > 1$, and $c \geq 0$.

We can again use the recursion tree approach to solve for $T$ in this case.

![Recursion tree diagram]

We can therefore express $T$ as

$$T(n) = \left(1 + \frac{a}{b^c} + \left(\frac{a}{b^c}\right)^2 + \cdots + \left(\frac{a}{b^c}\right)^{\log_b n}\right) \cdot \Theta(n^c).$$

To determine $T$, we must simply understand the geometric sequence with ratio $\frac{a}{b^c}$. There are three cases to consider.

**Case 1.** When $\frac{a}{b^c} = 1$, then each term in the sequence is 1 and so

$$T(n) = \log_b(n) \cdot \Theta(n^c) = \Theta(n^c \log n).$$

**Case 2.** When $\frac{a}{b^c} < 1$, then the geometric series $1 + \frac{a}{b^c} + \left(\frac{a}{b^c}\right)^2 + \cdots + \left(\frac{a}{b^c}\right)^{\log_b n} = \Theta(1)$ is a constant and so

$$T(n) = \Theta(n^c).$$

**Case 3.** When $\frac{a}{b^c} > 1$, then we have an increasing geometric series that is dominated by the last term, so that

$$T(n) = \Theta\left(n^c \cdot \left(\frac{a}{b^c}\right)^{\log_b n}\right) = \Theta(a^{\log_b n}) = \Theta(n^{\log_a b}).$$

This result establishes what is known as the **Master Theorem**.

**Theorem 3.7 (Master theorem).** If $T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^c)$ for some constants $a > 0$, $b > 1$, and $c \geq 0$, then

$$T(n) = \begin{cases} 
\Theta(n^c) & \text{if } c > \log_b a \\
\Theta(n^c \log n) & \text{if } c = \log_b a \\
\Theta(n^{\log_a b}) & \text{if } c < \log_b a.
\end{cases}$$

What is most important to remember of this theorem is not the statement of the theorem itself but rather the method that we used to obtain it: with the recursion tree method, you can easily recover the Master Theorem itself and also solve recursions that are not of the form covered by the theorem.
5. Geometric series

The different cases of the Master Theorem were handled rather quickly in the section above. It’s worth slowing down a bit to see exactly how the asymptotic results about geometric series are obtained. The starting point for those results is the fundamental identity for geometric series.

Fact 3.8. For any \( r \neq 1 \) and any \( n \geq 1 \),
\[
1 + r + r^2 + r^3 + \cdots + r^n = \frac{1 - r^{n+1}}{1 - r}.
\]

We can now use this lemma to give the big-\( \Theta \) closed form expressions for geometric series when \( r \neq 1 \). Let’s start with the case where \( r < 1 \).

Theorem 3.9. For any \( 0 < r < 1 \), the function \( f : n \mapsto 1 + r + r^2 + \cdots + r^n \) satisfies \( f = \Theta(1) \).

Proof. From the geometric series identity,
\[
1 + r + r^2 + r^3 + \cdots + r^n = \frac{1 - r^{n+1}}{1 - r}
\]
we observe that for every \( n \geq 1 \), \( f(n) \leq \frac{1}{1 - r} \) so, setting \( n_0 = 1 \) and \( c = \frac{1}{1 - r} \) we have that for all \( n \geq n_0 \), \( f(n) \leq c \cdot 1 \) and, therefore, \( f = O(1) \).

Similarly, if we let \( n_0 \) be the smallest integer for which \( r^{n_0} \leq \frac{1}{2} \) (which is the value \( n_0 = \lceil \log_{1/r}(2) \rceil \)), then for every \( n \geq n_0 \) we have \( \frac{1 - r^{n+1}}{1 - r} \geq \frac{1}{2(1 - r)} \) and, taking the constant \( c = \frac{1}{2(1 - r)} \), \( f \geq c \cdot 1 \) so \( f = \Omega(1) \).

Since \( f = O(1) \) and \( f = \Omega(1) \), then also \( f = \Theta(1) \). \( \square \)

The analysis of geometric series when \( r > 1 \) is similar.

Theorem 3.10. For any \( r > 1 \), the function \( f : n \mapsto 1 + r + r^2 + \cdots + r^n \) satisfies \( f = \Theta(r^n) \).

Proof. The geometric series identity can also be expressed as
\[
1 + r + r^2 + r^3 + \cdots + r^n = \frac{r^{n+1} - 1}{r - 1}.
\]
With \( n_0 = 1 \) and \( c = \frac{r}{r - 1} \), this identity implies that for all \( n \geq n_0 \), \( f(n) \leq \frac{r^{n+1} - 1}{r - 1} = c \cdot r^n \) so \( f = O(r^n) \).

And when we choose \( n_0 \) to be the smallest integer that satisfies \( r^{n_0} \geq 2 \) (or, equivalently, \( n_0 = \lceil \log_r(2) \rceil \)) and \( c = \frac{r}{2(r - 1)} \) then we have that for all \( n \geq n_0 \), \( f(n) = \frac{r^{n+1} - 1}{r - 1} \geq \frac{r^{n+1} - \frac{1}{2}r^{n+1}}{r - 1} = c \cdot r^n \) so \( f = \Omega(r^n) \).

Since \( f = O(r^n) \) and \( f = \Omega(r^n) \), then also \( f = \Theta(r^n) \). \( \square \)

\footnote{Can you see how to prove this identity? \textit{Hint}: Multiply the left-hand side of the identity by \( 1 - r \) and remove the terms that cancel out.}