It's Time to Learn

- The running time of an algorithm \( A \) is a function \( T_A : \mathbb{N} \rightarrow \mathbb{N} \) defined by
  \[ T_A(n) = \max_{I \in S} \{ \text{number of steps} \ A \text{ takes on} \ I \} \]
  where \( S \) is the set of instances \( I \) of size \( n \).
- For \( f, g : \mathbb{N}^+ \rightarrow \mathbb{R}^+ \), we say \( f(n) = O(g(n)) \) if there exist constants \( c, n_0 > 0 \) such that \( \forall n \geq n_0 \), we have \( f(n) \leq cg(n) \).
- For \( f, g : \mathbb{N}^+ \rightarrow \mathbb{R}^+ \), we say \( f(n) = \Omega(g(n)) \) if there exist constants \( c, n_0 > 0 \) such that \( \forall n \geq n_0 \), we have \( f(n) \geq cg(n) \).
- For both of the above, your choice of \( c \) and \( n_0 \) cannot depend on \( n \).
- For \( f, g : \mathbb{N}^+ \rightarrow \mathbb{R}^+ \), we say \( f(n) = \Theta(g(n)) \) if \( f(n) = O(g(n)) \) and \( f(n) = \Omega(g(n)) \).

In cake cutting, each player \( i \in N = \{1, 2, \ldots, n\} \) has a non-negative valuation function \( V_i \) over pieces of cake. The following properties about \( V_i \) hold:
- **Additive.** For \( X \cap Y = \emptyset \), \( V_i(X) + V_i(Y) = V_i(X \cup Y) \)
- **Normalized.** \( V_i([0, 1]) = 1 \)
- **Divisible.** \( \forall \lambda \in [0, 1], \text{you can always cut some } I' \subseteq I \text{ such that } V_i(I') = \lambda V_i(I) \)
- We say an allocation \( A_1, A_2, \ldots, A_n \) is:
  - **Proportional** if \( \forall i \in N, V_i(A_i) \geq \frac{1}{n} \)
  - **Envy-free** if \( \forall i, j \in N, V_i(A_i) \geq V_i(A_j) \). In other words, every player values their piece of cake at least as much as they value everyone else’s.

Bits and Pieces

Determine which of the following problems can be computed in worst-case polynomial-time, i.e. \( O(n^k) \) time for some constant \( k \), where \( n \) denotes the number of bits in the binary representation of the input. If you think the problem can be solved in polynomial time, give an algorithm in pseudo-code, explain briefly why it gives the correct answer, and argue carefully why the running time is polynomial. If you think the problem cannot be solved in polynomial time, then provide a proof.

(a) Give an input positive integer \( N \), output \( N! \).

To represent the value of \( N! \), we will need \( \log_2(N!) \) bits. As shown in the course notes, \( \log_2(N!) = \Theta(N \log N) \). Just writing this information down will take exponential time in input-size, which is \( \log_2 N \) bits.

(b) Given as input a positive integer \( N \), output True if \( N = M! \) for some positive integer \( M \).
Algorithm 1 turing machine $m_{\text{fact}}$

1: input: $N$
2: $x = 1$; $m = 1$
3: while $x < N$ do
4: \hspace{1em} $x = x \times m$
5: \hspace{1em} $m = m + 1$
6: end while
7: if $x == N$ then
8: \hspace{1em} return True
9: else
10: \hspace{1em} return False
11: end if

The polynomial-time algorithm is as follows: starting $M$ as 1, we keep creating $M!$ by multiplying the current value by $M$ each time in the loop. Then, in every loop, we check whether $M! < N$. When $M! \geq N$, we exit the loop and check whether $M! = N$ or $M! > N$. If they are equal, then return true, and false otherwise.

The analysis of the algorithmic complexity is as follows: First, the number of loops will be bounded by $O(\log N)$ because after $M > 2$, in each loop, we will multiply the number by two every time, so the number of iterations of the loop will be at most $\log_2 N + 1$. Then, in every loop, we will multiply the numbers $x$ and $M$. The complexity of this multiplication is $O(\log^2 N)$ by naive multiplication. Thus, the whole while loop will take $O(\log^3 N)$. Lastly, the comparison between two numbers $x$ and $N$ will take $O(\log N)$ time. Thus, the whole algorithm takes $O(\log^3 N) = O(n^3)$ time, which is polynomial.

(c) Given as input a positive integer $N$, output True iff $N = M^2$ for some positive integer $M$.

Algorithm 2 turing machine $m_{\text{square}}$

1: input: $N$
2: $l = 1$; $h = N$
3: while $l \leq h$ do
4: \hspace{1em} $x = \lfloor \frac{(l + h)}{2} \rfloor$
5: \hspace{1em} if $x^2 < N$ then
6: \hspace{2em} $l = x + 1$
7: \hspace{1em} else if $x^2 > N$ then
8: \hspace{2em} $h = x - 1$
9: \hspace{1em} else
10: \hspace{2em} return True
11: \hspace{1em} end if
12: end while
13: return False
We will perform binary search for the value of $x$ such that $x^2 = N$. At the beginning of each iteration of the while loop, the invariant $l \leq \sqrt{N} \leq h$ will be maintained. Since $l$ and $h$ are integers, and $h - l$ is decreasing (see below), we will find $\sqrt{N}$ if it is also an integer.

To analyze the run-time, the while loop will run for $O(\log N)$ steps. Observe that $h-l$ decreases by at least a half at the end of each iteration of the while loop. Since both $l$ and $h$ are integers, this process can go on at most $\log_2(h-l)+1 = O(\log(N))$ steps. During each iteration (lines 4-10), we compute the product of two integers each of which is at most $N$. Hence, we can do this computation in $O(\log^2 N)$ time. We also perform some comparisons and assignments, each of which only takes $O(\log N)$ time. Thus the total time, is $O(\log^3 N)$ time which is polynomial in $n = \log N$.

**Odd-Paz**

State and prove a divide-and-conquer procedure for proportional cake cutting between any number of players. (The Even-Paz algorithm as described in lecture is an excellent starting point.)

If $n = 1$: the single player takes the whole cake.

If $n = 2k$ for some $k \in \mathbb{N}^+$: every player draws a vertical line on the cake that cuts it in half according to their valuation function. Cut the cake anywhere between the $k$th and $(k+1)$th lines inclusive. The players who drew the $k$ leftmost lines recurse on the left piece and the players who drew the $k$ rightmost lines recurse on the right piece.

If $n = 2k+1$ for some $k \in \mathbb{N}^+$: every player draws a vertical line such that, according to their valuation function, the left side of the line is worth $\frac{k}{2k+1}$ of the total cake and the right side of the line is worth $\frac{k+1}{2k+1}$ of the total cake. Cut the cake on the median line. The players who drew the $k$ leftmost lines recurse on the left piece and the players who drew the $k+1$ rightmost lines recurse on the right piece.

Claim: This algorithm is proportional: for any piece $C$ of cake and for any number $n$ of players, each player will walk away with a slice $S_i$ that they feel is worth at least $\frac{1}{n}$ of the value of $C$.

(Equivalently, $\forall i. \ n \cdot V_i(S_i) \geq V_i(C)$). The proof is by induction.

**Base case ($n = 1$):** The player walks away with all of the remaining cake. $1 \cdot V_i(C) \geq V_i(C)$.

**Induction hypothesis:** for some $k \in \mathbb{N}^+$, this algorithm always achieves a proportional allocation among $(k-1)$ players.

**Induction step ($n = 2k$):** Note that after the cake is cut, the $k$ players who recurse on the left piece (call it $L$) value it at least half as much as they value $C$. By the IH, each player $i$ will end up with a piece that they think is worth at least $\frac{V_i(L)}{k}$. But since we just showed that $V_i(L) \geq \frac{V_i(C)}{2}$ for these players, this value is at least $\frac{V_i(C)}{n}$. So proportionality is achieved for these $k$ players. An identical argument applies to the other $k$ players, who recurse on the right piece.

**Induction step ($n = 2k+1$):** Note that after the cake is cut, the $k$ players who recurse on the left piece (call it $L$) value it at least $\frac{k}{2k+1}$ as much as they value $C$. By the IH, each player $i$ will end up with a piece that they think is worth at least $\frac{V_i(L)}{k} \geq \frac{V_i(C)}{k+1} = \frac{V_i(C)}{n}$. A structurally identical argument applies to the $k+1$ players who recurse on the right piece: by the IH, they each end up with a piece that they think is worth at least $\frac{V_i(R)}{k+1} \geq \frac{V_i(C)}{2k+1} = \frac{V_i(C)}{n}$. So proportionality is achieved for everybody.
(Extra) \( O \), I Think I Understand Asymptotics Now

Let \( f, g, h \) be functions from \( \mathbb{N} \) to \( \mathbb{N} \). Prove or disprove the following:

(a) If \( f \in O(g) \) and \( g \in O(h) \), then \( f \in O(h) \)

We know by definition that there exist \( n_1 \) and \( c_1 \) such that for all \( n \geq n_1 \), \( f(n) \leq c_1 g(n) \).
Similarly, we have \( n_2 \) and \( c_2 \) for \( g \) and \( h \).
Choose \( n_0 = \max(n_1, n_2) \) and \( c = c_1 c_2 \) and let \( n \geq n_0 \).
Then, \( f(n) \leq c_1 g(n) \leq c_1 c_2 h(n) = ch(n) \), as desired, so \( f \in O(h) \)

(b) If \( f \in O(g) \), then \( g \in O(f) \)

Let \( f(n) = 1 \) and \( g(n) = n \).
While \( f \in O(g) \), there is no \( n_0, c \) you can pick that makes \( n \leq c \) for all \( n \geq n_0 \), as we just let \( n > \max(c, n_0) \) as a counterexample.

(c) \( f \in O(g) \) or \( f \in \Omega(g) \)

Let \( f(n) = 1 \) if \( n \) is even and \( n \) if \( n \) is odd.
Let \( g(n) = n \) if \( n \) is even and \( 1 \) if \( n \) is odd.
By the same argument as the previous problem, if we attempt to choose \( c, n_0 \) for either \( O \) or \( \Omega \), we will be able to find a sufficiently large \( n \) (even or odd as necessary) to disprove the inequality.
(Extra) Where is the Median?

You are given two sorted arrays of integers with equal length. You want to determine the median of all the elements (if there are two elements in the middle, take the smallest one to be the median). Obtain a running time bound under the comparison model in terms of $n$, which we define to be the total number of elements in the arrays. In the comparison model, comparing two elements takes 1 step, and all other operations are free.

**Algorithm 3 findMedian**

1: input: $a$ an $1 \times N/2$ array, $b$ an $1 \times N/2$ array
2: if len($a$) == 0 then
3: \hspace{1em} return $b[N/2]$
4: else if len($b$) == 0 then
5: \hspace{1em} return $a[N/2]$
6: end if
7: $m_a = a[N/2]$; $m_b = b[N/2]$
8: if $m_a < m_b$ then
9: \hspace{1em} (a[x: y] is a subarray of a from x to y, y exclusive)
10: \hspace{2em} return findMedian($a[N/2, N]$, $b[0 : N/2]$)
11: else if $m_a > m_b$ then
12: \hspace{2em} return findMedian($a[0 : N/2]$, $b[N/2, N]$)
13: else
14: \hspace{2em} return $m_a$
15: end if

The algorithm is as follows: If either of the arrays has length of 0, get the median of the other non-zero length array. Otherwise, pick the medians of two arrays. If $med_1 < med_2$, you can eliminate the elements smaller than $med_1$ in the first array, and the elements bigger than $med_2$ in the second array. Then recurse. This works out fine when the two arrays have the same size. If they don’t, then in each step, you make sure you eliminate the same number of elements from each array. So you eliminate half of the smaller array many elements from each array. The algorithm will follow a recurrence relation $f(n) = f(n/2) + O(1)$, which is $O(\log n)$. 


(Bonus) Your Guesses are Two High!

Suppose I am thinking of a number between 1 and $n$, and will tell you if your guess is too high, too low, or correct. However, I only allow you to guess too high once, or you lose. How quickly can you guess my number? One possible solution is to just guess incrementally from 1 to my number, which takes $O(n)$ time. Can you do better?

We claim that we can solve this problem in $O(\sqrt{n})$ time. Our algorithm is as follows:

For each $i$ from 1 to $\sqrt{n}$ (which we assume is an integer, just for simplicity), query if the number I am thinking of is $i\sqrt{n}$. As long as the answer is “too low”, continue iterating. Once we hit “too high”, at some value $k\sqrt{n}$, stop iterating. We now begin at $(k - 1)\sqrt{n}$ and iterate up one at a time until we find the number. It is clear that this is correct as once we get our only “too high” guess, we go back down to the last value we know was low and then iterate one by one until we find the answer.

For runtime, first note we have to compute $\sqrt{n}$, which just takes $O(\log^3(n))$ time by the previous problem. We only need to do this once. From here, we make at most $2\sqrt{n} \in O(\sqrt{n})$ queries, as to isolate the $\sqrt{n}$ size chunk that the number is in, we make at most $\sqrt{n}$ guesses, and then when we guess “too high” the first time, we have to make at most $\sqrt{n}$ more guesses to search within the chunk. This gives a total runtime of $O(\sqrt{n})$, as desired.