Module 1 - Algorithm Structures

1.3 - Introduction to algorithms

3 Basic building blocks of algorithms:

1. **Assignments** - Gives values to variables.
2. **Conditional Statements** - Direct flow of steps.
3. **Loops** - Allow steps to be repeated.

1.4 - If-then algorithms

IF-statement - Tests a condition, and executes one or more instructions if the condition evaluates to true.

1.5 - For-loop algorithm

Iteration - A process where a set of instructions or structures are repeated in a sequence a specified number of times or until a condition is met.

1.6 - While-loop algorithm

While-loop - A process that iterates an unknown number of times, ending when a certain condition becomes false.

1.7 - Nested loop algorithm

Nested loop - When a loop is placed inside another loop.

Module 2 - Analyzing Algorithms

1.9 - Computational complexity

Algorithm Complexity - The study of the efficacy of algorithms that gives us a framework to compare algorithms.
Time Complexity - The ammount of time needed to perform the algorithm.
Space Complexity - The ammount of space needed to perform the algorithm.

Input size affects algorithm efficiency.

1.10 - Evaluating algorithm complexity

Atomic Operations - Operators that cannot be further split (+, -, *, >, !=, etc.,)

Counting atomic operations example:

```
Max: = 0
For i: =1 to n
    If (a[i] > Max), Max: = a[i]
End-for
```

Number of atomic operations:

1. Assigning a value to a variable
2. Looking up the value of a particular element in a sequence
3. Comparing two values
4. Incrementing a value (and/or arithmetic operations such as addition and multiplication)

1.11 - Worst-case analysis

(compute the worst case scenario of a particular algorithm to make informed decisions)

Module 3 - Big-O Estimates

1.13 - Asymptotic growth

Asymptotic complexity - Describes the behavior of the complexity function as n grows.

To find asymptotic complexity, remove all constants and focus in on the factor that grows the fastest.

Examples:

1. $f(n) = 32n + 50$ has an asymtotic complexity of n, since we ignore 50 and 32.
2. $f(n) = 64n^2 + 32n + 50$ has an asymtotic complexity of n^2, since we ignore constants and the slower growing n.

# of loops	Asymtotic Complexity
0	1
1	n
2	n^2
x	n^x

1.14 Lesson: Algorithms and big-O

Big-O notation($O(n)$) - The upper bound for the asymptotic complexity.
Big-Omega notation($\Omega(n)$) - The lower bound for the asymptotic complexity.
Big-Theta notation($\Theta(n)$) - [Somewhere in the middle of asymptotic complexity?].

$\Omega(f(n)) \leq \Theta(f(n)) \leq O(f(n))$

Module 4 - Divisibility and Modular Arithmetic

2.3 - The division algorithm

Number Theory - Branch of mathematics concerned with integers.

Integer Division - Division with remainder.

```
Input: Integers n and d > 0.
Output: q = n div d, and r = n mod d.

Case 1: n ≥ 0.'

q := 0
r := n
While ( r ≥ d )
      q := q + 1
      r := r - d
End-While
```

2.4 - Modular arithmetic and multiplication

(this section is stupid and can be summed up as "do math before you take mod anything)

example: $6 + 8 \mod 4 \equiv 14 \mod 4$

2.5 - Congruence modulo n

congruence - meaning "same measure". Examples are $400^\circ \equiv 40^\circ$ on unit circle, and 13:00 $\equiv$ 1:00 PM for time.

In modular arithmetic, the results of two mod operations are considered congruent if the result is the same.
So $13 \mod 4 \equiv 29 \mod 4$

For very large numbers (example: $(43^{17} + 32 * 139) \mod 7$) you can break them down thanks to congruency.
This means the above can be simplified as $43 \mod 7 + 32 \mod 7 * 139 \mod 7$.

Module 5 - Prime Factorization, GCD, and Euclid's Algorithm

2.7 - Prime Factorizations

Prime Number - A number n is prime if it's only factors are 1 and n.

Prime Factorization - Every integer greater than 1 can be expressed as a product of prime numbers, which is it's prime factorization.

Non-Decreasing Sequence - each number is equal to or greater than the one that came before.

Good Example: 1, 1, 2, 3, 17
Bad Example: 1, 1, 3, 2, 17

Multiplicity of a Prime Factor - The number of times a number p appears in a product of primes. This is represented as an exponent.
For 120, the product of primes in a non-decreasing sequence is $2 \cdot 2 \cdot 2 \cdot 3 \cdot 5$. This can be written as $2^3 \cdot 3 \cdot 5$.

ONE IS NOT A PRIME NUMBER

2.8 - Greatest common division and least common multiple

Greatest Common Divisor (GCD) - For non-zero integers x & y, the largest integer that factors into both x & y.

Least Common Multiple (LCM) - For non-zero integers x & y, the smallest integer that is an integer multiple of both x & y.

To find the greatest common divisor between two numbers, first find the prime factorization of both. If for example you were trying to find the GCD of 147 and 315, it's prime factorizations are:
$147 = 3 * 7^2$
$315 = 3^2 * 5 * 7$

To compare both, you can add 5^0 to the list of 147's prime factorization.

From here, take the minimum value of the exponents for each prime number, then find the product of that. So...

GCD(147, 315) = $3^1 * 5^0 * 7^1 = 3 * 7 = 21$

To find the least common multiple, take the same approach above but use the maximum value for each exponent.

LCD(147, 315) = $3^2 * 5^1 * 7^2 = 9 * 5 * 49 = 2205$

2.9 - Factoring and primality testing

Chances of a number x is a prime number is: $\frac{1}{\ln x}$

A "better" brute force methond for finding prime number:

```
Input: Integer N greater than 1.
Output: "Composite" if composite, else "Prime".

For x = 2 to
      If x evenly divides N,
            Return( "Composite" )
End-for

Return( "Prime" )
```

2.10 - Euclidean algorithm

Euclid's algorithm for finding the greatest common divisor

```
Input: Two positive integers, x and y.
Output: gcd(x, y).

If ( y < x )
      Swap x and y.
r = y mod x.

While ( r ≠ 0 )
      y := x
      x := r.
      r := y mod x.
End-while

Return( x )
```

Module 6 - Number Representation in Other Bases

2.14 - Number representation- Decimal and binary numbers

For a number with base b, represented like $(1234)_b$ it can be rewritten ($1*b^3 + 2*b^2 + 3*b^1 + 4*b^0$), assuming the base has enough digits to accommodate that 4.

2.15 - Number representation- Hexadecimal numbers

Hexadecimal is a base 16 numbering system with letters up to F to represent 15.

Hexadecimal goes into binary very well, each hex number being represented by 4 binary numbers (0000 - 1111).

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101

Hex	Decimal	Binary
E	14	1110
F	15	1111

2.16 - Base b expansion

Algorithm to find the base b expansion for a positive integer

```
Input: Integers n and b. b > 1 and n ≥ 1.
Output: Base b expansion of n. Base b digits are output in reverse order.

x := n
while ( x > 0 )
      Output( x mod b )
      x := x div b
End-while
```

The number of digits required to represent a number - Can be found with the formula: $\log_b(n+1)$
Example: How many digits are required to express the base 7 expansion of 2401? $\log_7(2402)$

Module 7 - Fast Exponentiation Algorithms

2.18 - Fast Exponentiation

(Fast exponentiation uses successive squaring of values rather than repeated multiplication)

For a number b being raised to exponent x (so b^x), instead of multiplying b x times, instead follow these steps:

1. Convert x into binary. (example: 5^7 becomes $5^{(111)_2}$)
2. For each "1" in that binary number, convert it into form 2^x, where x is the placement in the number. (example, 7= 2^2, 2^1, 2^0)
3. Raise b to each of those exponents found in step 2. (Example: $5^{2^2} * 5^{2^1} * 5^{2^0}$)

That makes for a faster method to find exponents for large numbers.

An iterative algorithm for fast exponentiation

```
Input: Positive integers x and y.
Output: x^y

p := 1    // p holds the partial result.
s := x    // s holds the current x^(2^j).
r := y    // r is used to compute the binary expansion of y

While ( r > 0 )
    If ( r mod 2 = 1 )
        p := p·s
    s := s·s
    r := r div 2
End-while

Return(p)
```

2.19 - Modular Exponentiation

A faster version of the method in section 2.18 exists using modular arithmetic.

An iterative algorithm for fast modular exponentiation

Input: Positive integers x and y.
Output: $x^y \bmod n$

```
p := 1    //p holds the partial result.
s := x    //s holds the current x^(2^j).
r := y    //r is used to compute the binary expansion of y

While ( r > 0 )
    If ( r mod 2 = 1 )
        p := p·s mod n
    s := s·s mod n
    r := r div 2
End-while

Return(p)
```

Module 8 - Mathematical Foundations of Encryption

2.21 - Introduction to Cryptography

Important terms:

Encrypt: the process of transforming information or data into a code to prevent unauthorized access.
Decrypt: the process of transforming data that has been rendered unreadable through encryption back to its unencrypted form.
Cyphertext: an encrypted message.
Plaintext: an unencrypted message.
Key: an unpredictable (typically large and random) string of numbers to decrypt an encrypted message.

2.22 - Encryption and Decryption

Symmetric Key Cryptosystem - Uses the same key to decrypt and encrypt the message.

Encrypt	Decrypt
$c = (m + k) \mod N$	$m = (c - k) \mod N$
m = message k = key 1 N = key 2 c = cyphertext	

2.23 - RSA Encryption

RSA encryption uses a public key for encryption and a private key for decryption. It's based on very large numbers (hundreds of digits)

Preparation of public and private keys in RSA

1. Bob selects two large prime numbers, p and q
2. Bob computes N = pq and φ = (p-1)(q-1)
3. Bob finds an integer e such that gcd(e, φ) = 1

4. Bob computes the multiplicative inverse of e mod φ: an integer d such that (ed mod φ) = 1
5. Public (encryption) key: N and e
6. Private (decryption) key: d

$$c = m^e \bmod N \text{ (encryption)}$$

2.24 - RSA Decryption

$$m = c^d \bmod N \text{ (decryption)}$$

Module 9 - Recurrence Relations

3.3 - Introduction to Recursive Relations

Recursion - A method that finds new values using previous values.

When discussing the n^{th} term of a sequence, it's common to write it as a_n.
So the sequence {1,2,3,4,5}, $a_3 = 3$.
$a_{3+1} = a_4 = 4$

3.4 - Evaluating a Recursive Relation

Recursive Relation - A rule that defines a term a_n as a function of the previous terms in a sequence.

Recurrence relation defining an arithmetic sequence

$a_0 = a$ *(initial value)*
$a_n = d + a_{n-1}$ *for* $n \geq 1$ *(recurrence relation)*

Initial value = a. Common difference = d.

Recurrence relation defining an geometric sequence

$a_0 = a$ *(initial value)*
$a_n = r \cdot a_{n-1}$ *for* $n \geq 1$ *(recurrence relation)*

Initial value = a. Common ratio = r.

Some recursive relations can depend on more than 1 previous terms. A famous example is the *Fibonacci Sequence*.

Fibonacci Sequence
$f_n = f_{n-1} + f_{n-2}$
when $f_0 = 0$
& $f_1 = 1$

Module 10 - Induction Methods

3.6 - Summations

Summation notation

$\sum_{k=s}^{n} a_k$
k = index
n = upper limit
s = lower limit

3.7 - Summation: limits and variables

Change of variables in summations

You can substitute the index variable to make summation equations simplified.
For example: if you have the summation $\sum_{j=1}^{17} 2^{j-1}$, that $j-1$ is pretty ugly.

You can make a new index variable k, where $k = j - 1$
From here, substitute the lower limit into k so $k = 1 - 1 = 0$
You now should have $\sum_{k=0}$

From here, substitute the upper limit into k so $k = 17 - 1 = 16$
You now should have $\sum_{k=0}^{16}$

The equality $k = j - 1$ means that $j = k + 1$. When we substitute that into 2^{j-1} we get $2^{(k+1)-1}$ or 2^k
With all this substitution, we now have $\sum_{k=0}^{16} 2^k \equiv \sum_{j=1}^{17} 2^{j-1}$

Closed forms for sums

You can rewrite a lot (but not all) of summation equations into normal equations.

Closed form for the sum of terms in an arithmetic sequence

For any integer n ≥ 1:

$$\sum_{k=0}^{n-1}(a+kd) = an + \frac{d(n-1)n}{2}$$

Closed form for the sum of terms in a geometric sequence

For any real number r ≠ 1 and any integer n ≥ 1:

$$\sum_{k=0}^{n-1} a \cdot r^k = \frac{a(r^n-1)}{r-1}$$

3.8 - Inductive proof

For a proof by induction, if $Q(n)$ is true, then $Q(n+1)$ is true. Think dominoes: if the base case (first domino knocking over) is true, then the n^{th} domino falling must be true.

3.9 - Divisibility proof by induction

(Example with division)

3.10 - Induction proof of a recurrence relation

(Example with recurrence releation)

Explicit formula for a sequence defined by a recurrence relation.

Define the sequence $\{g_n\}$ as:

- $g_0 = 1$.
- $g_n = 3 \cdot g_{n-1} + 2n$, *for any* $n \geq 1$.

Then for any $n \geq 0$,

$$g_n = \frac{5}{2} \cdot 3^n - n - \frac{3}{2}$$

3.11 - Induction proof of closed summation

(Example with closed summation)

Closed form for the sum of terms in an arithmetic sequence

For any integer $n \geq 1$:

$$\sum_{j=0}^{n-1} (a + jd) = an + \frac{d(n-1)n}{2}$$

3.12 - Strong Induction

3.13 - Well-ordering principle

well-ordering principle - any non-empty subset of non-negative integers has a smallest element.

Mathematical induction $\equiv$ Strong induction $\equiv$ well-ordering principle.
Prove one, you prove the other two.

Module 11 - Recursive Structures

3.15 - Overview of recursive definitions

Recursion - The process of computing the value of a function using the result of the function on smaller input values.

In other words, it's a funciton that calls itself using a smaller value. It will keep calling itself until it hits a base case.

A famous example is factorials. For a factorial N:

```
function factorial (n) {
  return ( n === 0 ? 1 : n * factorial(n-1) );
}
```

3.16 - Recursively defined sets

Components of a recursively defined set:
Basis - explicitly states that one or more specific elements are in the set.
recursive rule - shows how to construct larger elements in the set from elements already known to be in the set.
exclusion statement - states that an element is in the set only if it is given in the basis or can be constructed by applying the recursive rules repeatedly to elements given in the basis.

An example of this is found in figuring out if nested parenthesis are correct. The **basis** in this example is the sequence () is properly nested. This has two **recursive rules**:

1. (u) is properly nested
2. uv is properly nested

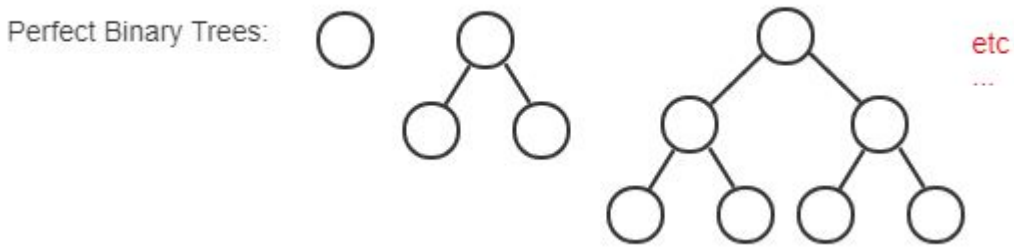
Our **exclusion statement** is: a string is properly nested only if it is given in the basis or can be constructed by applying the recursive rules to strings in the basis.

So using our recursive rules: () is correct, (()) is correct [(u)], and ()() is correct [uv]

3.17 - Recursive definition for perfect binary trees

1. Start with a single node, this will be our "tree".

2. Copy your existing tree.
3. Place your copied tree along side of your existing tree.
4. Add another node above, making it a parent of both trees.
5. Repeat steps 2-4 as much as you like.



3.18 - Recursive/inductive algorithms

Recursive Algorithm - an algorithm that calls itself.

Module 13 - Counting by Bijections and Products of Sets

4.3 - Sum and product rules

Product Rule - For a series of finite sets, the product (or the cardinality of) the set is |set| * |set|...
So if you have a meal that is made of 1 drink, 1 dish, 1 desert, and you have 3 options for each, there will be this many possible chocies:
$|drink| * |dish| * |desert| = 3 * 3 * 3 = 27$ possible choices.

To **count strings** with this rule, you first need to know how many possible choices there are for each "letter'.
For example, to count how many possible choices there are for a 5 letter "word" of binary (so $\{0, 1\}$) you would do = $\{0, 1\}^5$ or $2^5 = 32$.
For the possible amount of words using the english alphabet for length n you would have 26^n

Sum Rule - For when there are multiple choices but only one selection is made.
So if you have a meal with the choice of a hot drink (coffee or tea) or a cold drink (coke, milkshake or water) and the customer only wants one drink, you would do:
|cold drinks| + |hot drinks| $= 2 + 3 = 5$
To use this, all sets must be disjoint of each other (or in other words, an element is not in multiple sets. A drink can't be both hot and cold in this example)

An example that threw me off is:

Q: How many binary strings of length 5 or 6 start with a 1?

To answer this, find out how many strings exist that start with a 1 that are length of 5 ($2^4 = 16$)
Next, find out how many strings exist that start with a 1 that are length of 6 ($2^5 = 32$)
Add them together. The answer is 48.

4.4 - Bijection rule

If a set has a well defined inverse (so $f(s) = t$, if and only if $g(t) = s$), then the two sets have the same cardinality. In other words, if the sets are a bijection of each other, then they have the same cardinality.

The **k-to-1 rule** states that if there is a k-to-1 difference between sets A and B, then $|B| = |A|/k$.
In simple terms, a pile of shoes typically has a 2-to-1 ratio between humans (2 feet, 2 shoes) so |humans| = |shoes|/2.

4.5 - The generalized product rule

The set can "change size" when using the product/sum rules.
If you have a set of 10 racers and want to know the different possibilities of podium winners, then you would do:

$|10| * |9| * |8|$ since someone cannot win two different medals.

Module 14 - Counting with Permutations and Combinations

4.7 - Counting permutations

Instead of previous counts where the order doesn't matter, permutations means that the order matters.
r-permutations - permutations of length r taken from the same set without repetitions.

The *generalized product rule* is used here. If you are finding 4-permutations of a set of 10 items, then you have $10 * 9 * 8 * 7$ different possibilities.

permutation (without the leading "r-") is a sequence that contains every element of a finite set exactly once.

Formulas:

1. **r-permutation** = $P(n, r) = \frac{n!}{(n-r)!}$
   A sequence of r items chosen from n total items in which the order of the items matters.
2. **Permutation** = $P(n, n) = n!$
   A sequence of n items in which the order of the items matters and every item in a set is included exactly once.
3. **r-Subset / r-combination** = $C(n, r) = \binom{n}{r} = \frac{n!}{r!(n-1)!}$
   A sequence of r items chosen from n total items in which the order of the items does *not* matter.

4.8 - Counting subsets

Combinations (or subsets) - counting from a group when the order doesn't matters.
In order to count subsets/combinations, use the 3rd formula from the lesson above.

In the binomial $\binom{n}{r}$ it's read: "n choose r", or "from the set n, choose r choices for the subset"

Identity for combinations $\binom{n}{n-r} \equiv \binom{n}{r}$
In other words, "12 choose 8" $\equiv$ "12 choose 4"

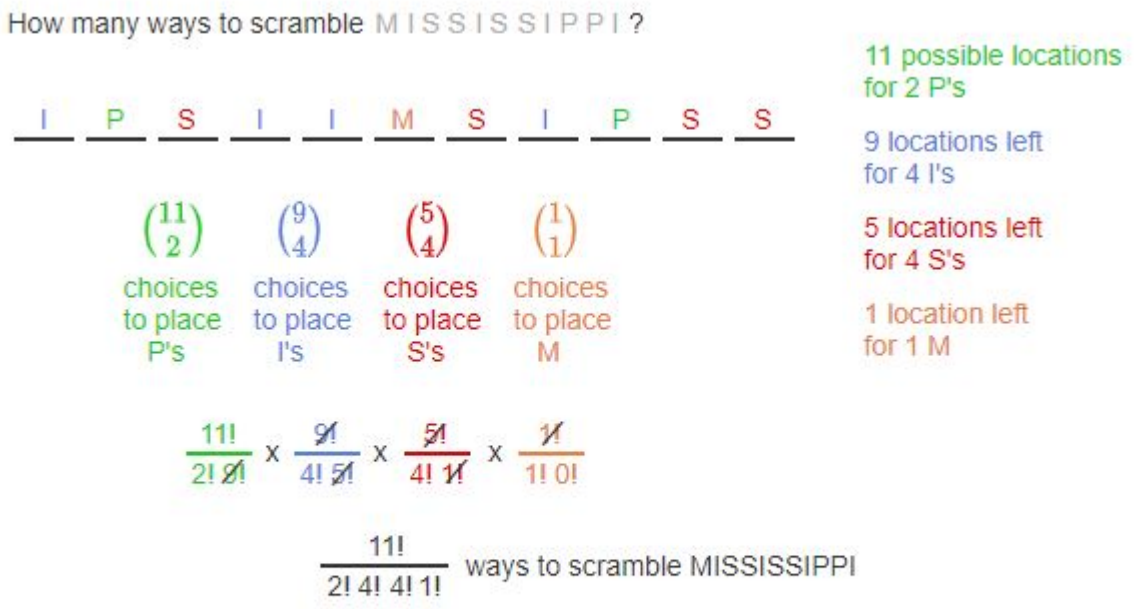
4.9 - Subset and permutation examples

(Just exercises for the above two sections)

4.10 - Permutations with repetitions

Sometimes in permutations, a value will be repeated and the order will not matter for any repeated outcome.
For example, the string "Good", and "Good" are different (the two "o"s are swapped) but it doesn't matter since the outcome is the same.
This is called **permutation with repetition.**

Useful illustation:



$$\binom{11}{2}\binom{9}{4}\binom{5}{4}\binom{1}{1} = \frac{11!}{2!9!} \cdot \frac{9!}{4!5!} \cdot \frac{5!}{4!1!} \cdot \frac{1!}{1!0!} = \frac{11!}{2!4!4!1!}$$ ways exist to scramble MISSISSIPPI.

Module 15 - Counting with Multisets

4.12 - Counting by complement

You have 3 total cookies, one of them being oatmeal raisin. How many of them are actually good? The answer is 2 (3-1=2).
This is called **Counting by complement**. $|P| = |S| - |\overline{P}|$

4.13 - Counting multisets

Set - A collection of *distinct* items.
Multiset - a collection that can have multiple instances of the same kind of item.

Formula for counting a multiset:
$\binom{n+m-1}{m-1} = \frac{(n+m-1)!}{(m-1)!n!}$
Where n objects and m varieties (12 cookies, 4 possible flavors)

Module 16 - Generating Permutations and Combinations

4.15 - Generating permutations and combinations

Lexicographic order - Compares two tuples and how they differ, which one will be larger than the other.
This follows the same rules as alphabetical order in grade school. For (1,2,3,5) and (1,2,6,1), the second one is larger because where they differ (index 3, or 3 & 6) it's larger.
This only works for *sequences* and *tuples* where the order matters.

Algorithm to generate permutations in lexicographic order

```
GenLexPermutations(n)

Initialize P = (1, 2, …, n - 1, n)
Output P
While P ≠ (n, n - 1, …, 2, 1),
    P = NextPerm(P)
    Output P
```

(4.15.4: Finding the next permutation in lexicographic order. <- is very helpful)

4.16 - Generating r-subsets of a set

Where the order doesn't matter, you first sort the set in increasing order to compare them.
So {3,2,4} > {5,1,3} because sorted they become {2,3,4} > {1,3,5}

Algorithm to generate r-subsets in lexicographic order

```
GenLexSubsets(r, n)

Initialize S = {1, 2, …, r-1, r}
Output P
While S ≠ {n-r+1, …, n-1, n},
    S = NextSubset(n, S)
    Output S
```

Module 17 - Advanced Counting Techniques

4.18 - Inclusion-exclusion principle

inclusion-exclusion principle - When counting two sets that overlap (red playing cards & face cards) you will need to use the following formula to not count the intersection twice:

$|A \cup B| = |A| + |B| - |A \cap B|$

inclusion-exclusion principle with 3 sets -

$|A \cup B \cup C| = |A| + |B| + |C| - |A \cap B| - |B \cap C| - |A \cap C| + |A \cap B \cap C|$

4.19 - Binomial coefficients and combinatorial identities

combinatorial identity:
$\binom{n}{k} \equiv \binom{n}{n-k}$

The Binomial Theorem:

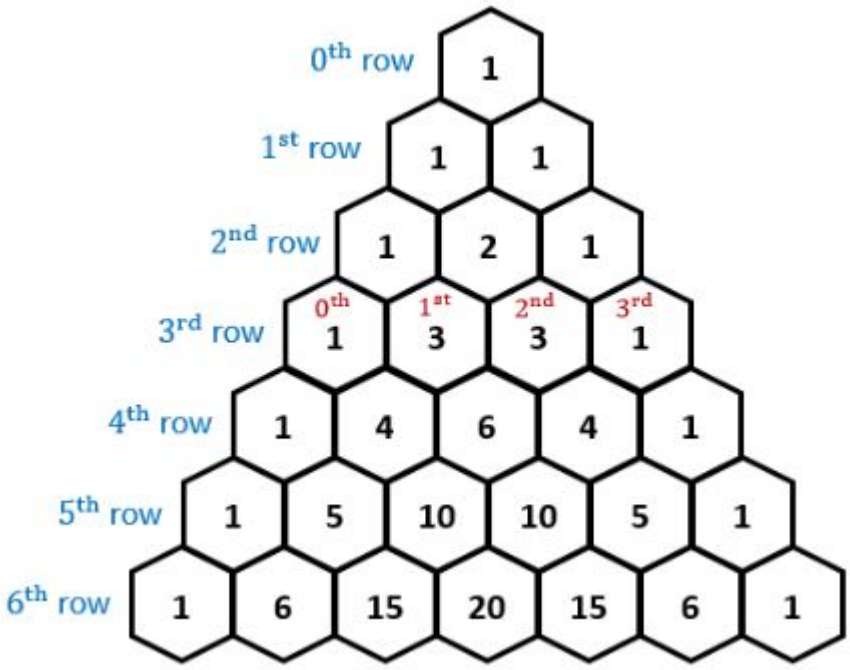
$$(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^{n-k} b^k$$

For the case n = 5, the Binomial Theorem says that

$$(a+b)^5 = \binom{5}{0}a^5 + \binom{5}{1}a^4 b + \binom{5}{2}a^3 b^2 + \binom{5}{3}a^2 b^3 + \binom{5}{4}ab^4 + \binom{5}{5}b^5$$

$$= a^5 + 5a^4 b + 10a^3 b^2 + 10a^2 b^3 + 5ab^4 + b^5$$

Pascal's Triangle
Construct a triangle with binomials $\binom{n}{k}$ where n is the number of rows from the top (0 being the tip) and k being the index (or column) in row n.
To calculate each rows value, the outside values are 1, and the inside values are the sum of the two numbers above it.



4.20 - The pigeonhole principle

Pigeonhole Principle - if n+1 pigeons are placed in n boxes, then there must be at least one box with more than one pigeon.

In other words:

- If you have 3 colors of socks. After 3 socks are selected, there might not be a pairAfter the next sock is selected, it will make a pair.
- Among a group of 400 people, there are at least two who have the same birthday.

> The generalized pigeonhole principle
>
> *Consider a function whose domain has n elements and whose target has k elements, for n and k positive integers. Then there is an element y in the target such that f maps at least $\lceil n/k \rceil$ elements in the domain to y.*

In the case of the socks, n = 10 is the number of socks and k = 3 is the number of colors. After selecting 10 socks, the twins are guaranteed that they have at least $\lceil 10/3 \rceil = 4$ socks which are all of the same color.

Generalized pigeonhole principle

```
Suppose that a function maps a set of n elements to a target set with k elements, where n and k are positive int
In order to guarantee that there is an element y in the target to which f maps at least b items,
then n must be at least k(b - 1) + 1.
```

This works for situations where you are filling in a list evenly. Say you are creating two basketball teams and want to know how many people you have to pick until at least one team has someone on the bench. That would be from a set N (students in gym class) going to set K (basketball teams), where b is a benched player (or 6 people on a team).

$k(b-1)+1$

or $2(6-1)+1=11$

So the 11th player pick is guaranteed to be placed on the bench.

Module 18 - Introduction to Discrete Probability

5.3 - Introduction to discrete probability

Outcome - A single possible output.
Sample Space - Set of all possbile outcomes.
Event - Subset of the Sample Space.
Discrete probability - Experiments in which the sample space is a finite or countably infinite set.
countably infinite - one-to-one correspondence between the elements of the set and the integers.

5.4 - Probability of an event

$P(E) = \frac{|E|}{|S|}$ where E is how many times the event can come up, and S is the entire sample size.
Example, getting a 1 on a d6 is 1/6.
Getting a 6 on a loaded d6, where 6 is twice as likely to come up is 2/7.

5.5 - Inclusion-exclusion rule

Mutually exclusiive events - Two or more events that cannot coexist.
Example: in a sample space of 5 coin flips, the first three flips are heads and the last three flips are tails. Both cannot exist at the same time.

If two events are not mutually exclusive, the probability of the union of events can be determined by a version of the **Inclusion-Exclusion principle**:

$$p(E_1 \cup E_2) = p(E_1) + p(E_2) - p(E_1 \cap E_2)$$

5.6 - Complement rule

$p(\overline{E}) = 1 - p(E)$

Because it's sometimes easier to find out the opposite and subtract that from the total than it is to find out the event itself.

Module 19 - Conditional Probability and Bayes' Theorem

5.8 - Conditional probability

The probability of an event E happening in sample space S is known. Then an event F happens. This is how you find out what the likelyhood of E is AFTER F has happened.

For example: Say you are rolling dice and want to know the likely hood that the total is at least 10 (event E). That set is {(4,6), (6,4) (5,5), (5,6), (6,5), (6,6)}.
You roll the first die and it comes up 5 (event F). Now what is the likelyhood that the total is at least 10?
Event F is the likleyhood that the roll came up 5, so 1/6.

To find the **Conditional Probability** of this event, use the following formula:

$$p(E|F) = \frac{p(E \cap F)}{p(F)}$$

So we know $|E| = 6$ and $|F| = 6$. $|E \cap F| = 2$
$2/6 = 1/3$

5.9 - Independent vs. dependent events

Two events are **independent** if conditioning on one event does not change the probability of the other event.

Independent events

Let E and F be two events in the same sample space. The following three conditions are equivalent:

1. $p(E \mid F) = \frac{p(E \cap F)}{p(F)} = p(E)$
2. $p(E \cap F) = p(E) \cdot p(F)$
3. $p(F \mid E) = \frac{p(E \cap F)}{p(E)} = p(F)$

If the three conditions hold, then events E and F are independent.

5.10 - Bayes' Theorem

Bayes' Theorem

Suppose that F and X are events from a common sample space and p(F) ≠ 0 and p(X) ≠ 0. Then

$$p(F \mid X) = \frac{p(X \mid F)p(F)}{p(X \mid F)p(F) + p(X|\overline{F})p(\overline{F})}$$

Module 21 - Deterministic Finite State Machine

6.3 - Theory of computation

Computation - Any type of calculation that follows steps that provide a solution to a problem.

4 Major Families of Automata

1. **Finite State Machine** - Every bit exists as either a 0 or 1, and a finite possibility of inputs exist.
2. **Pushdown Automaton** - Uses a stack/collection that collects elements.
3. **Linear-bounded Automation** - Two end markers with input in between.
4. **Turing Machine** - A finite state machine with infinite storage.

Grammar - A set of rules for generating strings.

6.4 - Deterministic finite state automata (DFA)

A deterministic finite state machine is a machine who's outcomes are predetermined and dependent on the input.
For example, a subway turnstile. It starts locked, and accepts either a push or a coin. If it gets a coin, it unlocks (and therefore can be pushed).
If it's pushed while unlocked, it turns then locks.

DFA components: **Alphabet** - The finite amount of input strings.
Transistion Function - The set of rules that determine the output. Based on current state and next input string.
Start State - The state of which processing strings begins.

Formally, a DFA is defined using the following **5-tuple (quintuple)**, or ordered list of 5 elements.

$$(Q, \Sigma, \delta, q_0, F)$$

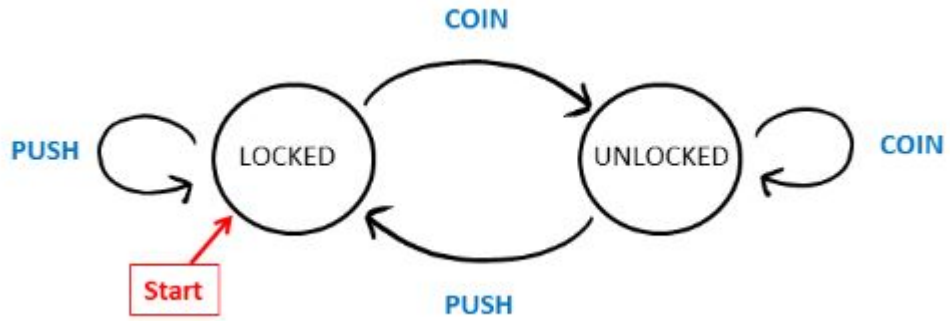
The meaning of each symbol is as follows.

Components of a DFA

Symbol	Description
Q	The finite set of states.
Σ	The alphabet.
$\delta : Q \times \Sigma \to Q$	The transition function.
$q_0 \in Q$	The state in Q that is the start state.
$F \subseteq Q$	A set of accepting states that is a subset of Q.

6.5 - DFA state diagrams

(Below is a visualization of the DFA state)



6.6 - DFA state transition tables

A table can be used to represent the transition between states.
The columns will be the input, the rows being the current state.
The cells will be the output (given the current state and input)

	COIN	PUSH
LOCKED	UNLOCKED	LOCKED

	COIN	PUSH
UNLOCKED	UNLOCKED	LOCKED

6.7 - DFA analyzing transition rules

(Examples & exercises of the three sections above)

6.8 - DFA evaluating outcomes

Some finite state machines can have "accepting" values, meaning that the output is "true" if it ends in a certain state.
On a diagram, this is represented by a double circle around the state.

Module 22 - Nondeterministic Finite State Machine

6.10 - Nondeterministic finite automata (NFA)

Nondeterministic Finite Automata (NFA) - A finite state automaton whose next state is not uniquely determined by the current state and the next input symbol.

Epsilon Transition (ε) - A transition from one state to another without reading an input.

Formally, an NFA is also defined using a 5-tuple.

$$(Q, \Sigma, \delta, q_0, F)$$

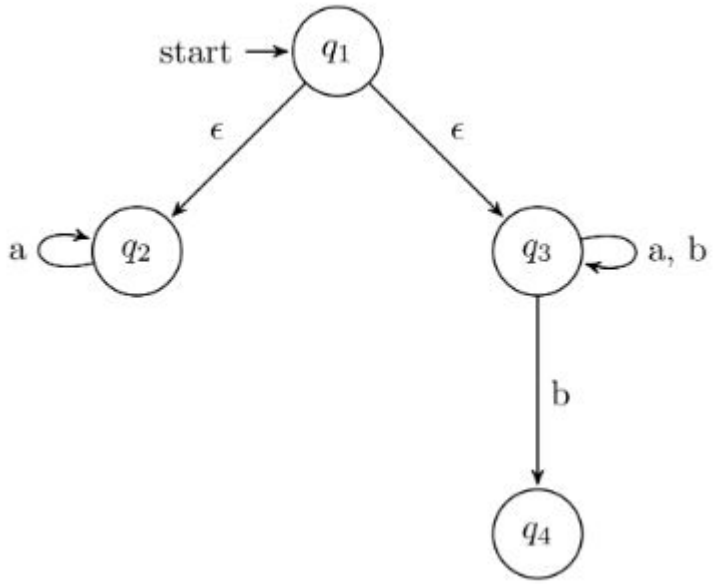
The meaning of each symbol is as follows.

Components of an NFA

Symbol	*Description*
Q	*The finite set of states.*
Σ	*The alphabet.*
$\delta : Q \times \Sigma_\epsilon \to P(Q)$	*The transition function. Σ_ϵ represents $\Sigma \cup \{\epsilon\}$, that is, the set of possible input symbols plus the possibility of no input symbol at all. Since the next state can be no state, one state, or one of many possible states, the output of the transition function is an element of $P(Q)$, the power set of Q.*
$q_0 \in Q$	*The state in Q that is the start state.*
$F \subseteq Q$	*A set of accepting states that is a subset of Q.*

6.11 - NFA state diagrams

The NFA diagram is similar to the DFA, only it resembles more of a tree structure.



6.12 - NFA state transition tables

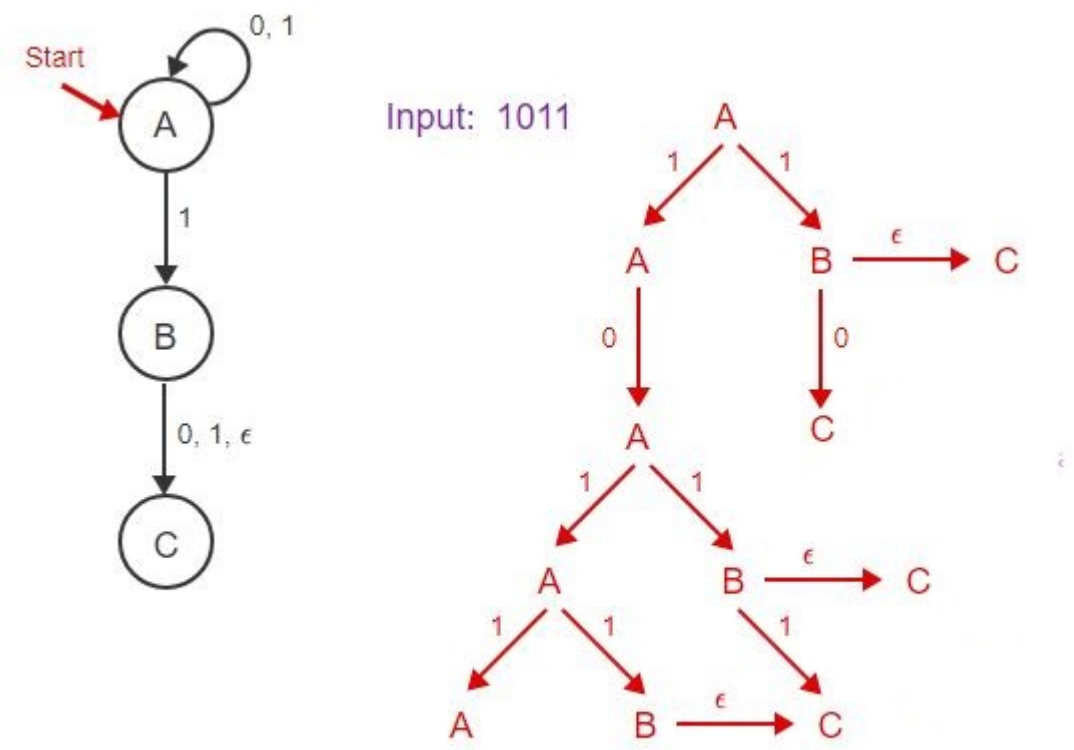
The diagram found in 6.11 can be represented as a table:

State transition table for an NFA

	a	b	ϵ
q_1	$\emptyset$	$\emptyset$	$\{q_2, q_3\}$
q_2	$\{q_2\}$	$\emptyset$	$\emptyset$
q_3	$\{q_3\}$	$\{q_3, q_4\}$	$\emptyset$
q_4	$\emptyset$	$\emptyset$	$\emptyset$

6.13 - NFA analyzing transition rules

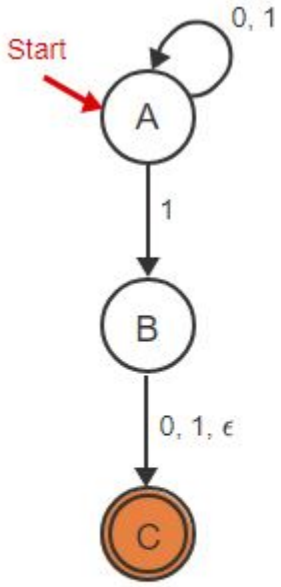
When analysing inputs, you create a tree of *ALL* possible outcomes.
Even if there is an input, you must also consider the epsilon transition of each step.
Also add an epsilon transition at the end of the input (so an input of {1,0} becomes {1,0,ε})



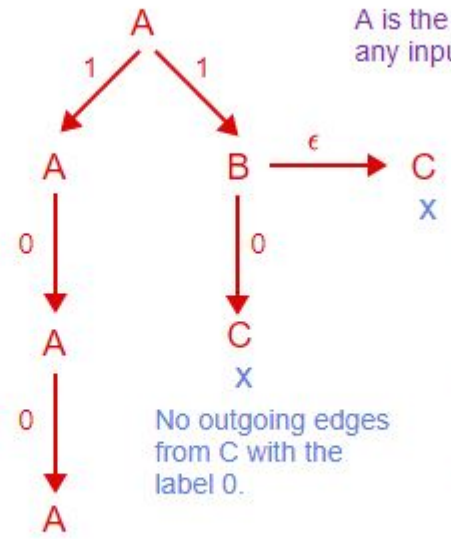
6.14 - NFA evaluating outcomes

If the machine has an accepting state outcome (indicated by double circles), then it is true (accepting) if the outcome set includes the accepted state.
Example: If the accepting outcome is {c} and the outcome set is {a,b,c}, then the outcome is acceptable.

HOWEVER, the last part of the input is what determines the output. See example below:



Input: 100



A is the only possible state before any input characters have been read

The set of possible states after input "1" is {A,B,C}

No outgoing edges from C with the label 0.

No outgoing edges from C with the label 0.

The set of possible states after input "10" is {A,C}

The set of possible states after input "100" is {A}

The state A is not an accepting state. Therefore the NFA rejects string "100".