In [1]:

```python
from typing import Tuple
import math
```

In [2]:

```python
def normal_approximation_to_binomial(n: int, p: float) -> Tuple[float, float]:
    """Returns mu and sigma corresponding to a Binomial(n, p)"""
    mu = p * n
    sigma = math.sqrt(p * (1 - p) * n)
    return mu, sigma
```

In [5]:

```python
from scratch.probability import normal_cdf
```

In [6]:

```python
# The normal_cdf Is the Probability That the Variable Is Below a Certain Threshold
normal_probability_below = normal_cdf
```

In [7]:

```python
# It's Above the Threshold If It's Not Below the Threshold
def normal_probability_above(lo: float,
                             mu: float = 0,
                             sigma: float = 1) -> float:
    """The probability that a N(mu, sigma) is greater than lo."""
    return 1 - normal_cdf(lo, mu, sigma)
```

In [8]:

```python
# It's Between the Threshold If It's Less Than "hi", but Not Less Than "lo"
def normal_probability_between(lo: float,
                               hi: float,
                               mu: float = 0,
                               sigma: float = 1) -> float:
    """The probability that a N(mu, sigma) is between lo and hi."""
    return normal_cdf(hi, mu, sigma) - normal_cdf(lo, mu, sigma)
```

In [9]:

```python
# It's Outside the Threshold If It's Not Between
def normal_probability_outside(lo: float,
                               hi: float,
                               mu: float = 0,
                               sigma: float = 1) -> float:
    """The probability that a N(mu, sigma) is not between lo and hi."""
    return 1 - normal_probability_between(lo, hi, mu, sigma)
```

In [10]:

```python
from scratch.probability import inverse_normal_cdf
```

In [11]:

```python
def normal_upper_bound(probability: float,
                       mu: float = 0,
                       sigma: float = 1) -> float:
    """Returns the z for which P(Z <= z) = probability"""
    return inverse_normal_cdf(probability, mu, sigma)
```

In [12]:

```
def normal_lower_bound(probability: float,
                       mu: float = 0,
                       sigma: float = 1) -> float:
    """Returns the z for which P(Z >= z) = probability"""
    return inverse_normal_cdf(1 - probability, mu, sigma)
```

In [13]:

```
def normal_two_sided_bounds(probability: float,
                            mu: float = 0,
                            sigma: float = 1) -> Tuple[float, float]:
    """
    Returns the symmetric (about the mean) bounds that contain the specified probabilit
    """
    tail_probability = (1 - probability) / 2

    # The Upper Bound Should Have a tail_probability Above It
    upper_bound = normal_lower_bound(tail_probability, mu, sigma)

    # The Lower Bound Should Have a tail_probability Below It
    lower_bound = normal_upper_bound(tail_probability, mu, sigma)

    return lower_bound, upper_bound
```

In [14]:

```
mu_0, sigma_0 = normal_approximation_to_binomial(1000, 0.5)
```

In [15]:

```
assert mu_0 == 500
```

In [16]:

```
assert 15.8 < sigma_0 < 15.9
```

In [17]:

```
# (469, 531)
lower_bound, upper_bound = normal_two_sided_bounds(0.95, mu_0, sigma_0)
```

In [18]:

```
assert 468.5 < lower_bound < 469.5
```

In [19]:

```
assert 530.5 < upper_bound < 531.5
```

In [20]:

```
# 95% Bounds Based On the Assumption That p Is 0.5
lo, hi = normal_two_sided_bounds(0.95, mu_0, sigma_0)
```

In [21]:

```
# The Actual mu and sigma Are Based On p = 0.55
mu_1, sigma_1 = normal_approximation_to_binomial(1000, 0.55)
```

In [22]:

```
# A Type 2 Error Means We Fail to Reject the Null Hypothesis, Which Will Happen When X
type_2_probability = normal_probability_between(lo, hi, mu_1, sigma_1)
power = 1 - type_2_probability      # 0.887
```

In [23]:
```
assert 0.886 < power < 0.888
```

In [24]:
```
hi = normal_upper_bound(0.95, mu_0, sigma_0)
# Is 526 (< 531, Since We Need More Probability In the Upper Tail)
```

In [25]:
```
type_2_probability = normal_probability_below(hi, mu_1, sigma_1)
power = 1 - type_2_probability      # 0.936
```

In [26]:
```
assert 526 < hi < 526.1
```

In [27]:
```
assert 0.9363 < power < 0.9364
```

In [28]:
```
def two_sided_p_value(x: float, mu: float = 0, sigma: float = 1) -> float:
    """
    How likely are we to see a value at least as extreme as x (in either direction) if 
    """
    if x >= mu:
        # x Is Greater Than the Mean, so the Tail Is Everything Greater Than x
        return 2 * normal_probability_above(x, mu, sigma)
    else:
        # x Is Less Than the Mean, so the Tail Is Everything Less Than x
        return 2 * normal_probability_below(x, mu, sigma)
```

In [29]:
```
two_sided_p_value(529.5, mu_0, sigma_0)   # 0.062
```

Out[29]: 0.06207721579598835

In [30]:
```
import random
```

In [31]:
```
extreme_value_count = 0
for _ in range(1000):
    num_heads = sum(1 if random.random() < 0.5 else 0    # Counts the number of heads
                    for _ in range(1000))                # in 1000 flips
    if num_heads >= 530 or num_heads <= 470:             # and count how often
        extreme_value_count += 1                         # the number is "extreme"
```

In [ ]:
```
# p-Value Was 0.062 => ~62 Extreme Values Out of 1000
assert 59 < extreme_value_count < 65, f"{extreme_value_count}"
```

In [33]:
```
two_sided_p_value(531.5, mu_0, sigma_0)   # 0.0463
```

Out[33]: 0.046345287837786575

In [34]:
```
tspv = two_sided_p_value(531.5, mu_0, sigma_0)
```

In [35]:
```python
assert 0.0463 < tspv < 0.0464
```

In [36]:
```python
upper_p_value = normal_probability_above
lower_p_value = normal_probability_below
```

In [37]:
```python
upper_p_value(524.5, mu_0, sigma_0) # 0.061
```

Out[37]: 0.06062885772582072

In [38]:
```python
upper_p_value(526.5, mu_0, sigma_0) # 0.047
```

Out[38]: 0.04686839508859242

In [39]:
```python
p_hat = 525 / 1000
mu = p_hat
sigma = math.sqrt(p_hat * (1 - p_hat) / 1000)   # 0.0158
```

In [40]:
```python
normal_two_sided_bounds(0.95, mu, sigma)        # [0.4940, 0.5560]
```

Out[40]: (0.4940490278129096, 0.5559509721870904)

In [41]:
```python
p_hat = 540 / 1000
mu = p_hat
sigma = math.sqrt(p_hat * (1 - p_hat) / 1000) # 0.0158
normal_two_sided_bounds(0.95, mu, sigma) # [0.5091, 0.5709]
```

Out[41]: (0.5091095927295919, 0.5708904072704082)

In [42]:
```python
from typing import List
```

In [43]:
```python
def run_experiment() -> List[bool]:
    """Flips a fair coin 1000 times, True = heads, False = tails"""
    return [random.random() < 0.5 for _ in range(1000)]
```

In [44]:
```python
def reject_fairness(experiment: List[bool]) -> bool:
    """Using the 5% significance levels"""
    num_heads = len([flip for flip in experiment if flip])
    return num_heads < 469 or num_heads > 531
```

In [45]:
```python
random.seed(0)
experiments = [run_experiment() for _ in range(1000)]
num_rejections = len([experiment
                      for experiment in experiments
                      if reject_fairness(experiment)])
```

In [46]:

```
assert num_rejections == 46
```

In [47]:

```
def estimated_parameters(N: int, n: int) -> Tuple[float, float]:
    p = n / N
    sigma = math.sqrt(p * (1 - p) / N)
    return p, sigma
```

In [48]:

```
def a_b_test_statistic(N_A: int, n_A: int, N_B: int, n_B: int) -> float:
    p_A, sigma_A = estimated_parameters(N_A, n_A)
    p_B, sigma_B = estimated_parameters(N_B, n_B)
    return (p_B - p_A) / math.sqrt(sigma_A ** 2 + sigma_B ** 2)
```

In [49]:

```
z = a_b_test_statistic(1000, 200, 1000, 180)    # -1.14
```

In [50]:

```
assert -1.15 < z < -1.13
```

In [51]:

```
two_sided_p_value(z)                            # 0.254
```

Out[51]: 0.254141976542236

In [52]:

```
assert 0.253 < two_sided_p_value(z) < 0.255
```

In [53]:

```
z = a_b_test_statistic(1000, 200, 1000, 150)    # -2.94
two_sided_p_value(z)                            # 0.003
```

Out[53]: 0.003189699706216853

In [54]:

```
def B(alpha: float, beta: float) -> float:
    """A normalizing constant so that the total probability is 1"""
    return math.gamma(alpha) * math.gamma(beta) / math.gamma(alpha + beta)
```

In [55]:

```
def beta_pdf(x: float, alpha: float, beta: float) -> float:
    if x <= 0 or x >= 1:          # No weight outside of [0, 1]
        return 0
    return x ** (alpha - 1) * (1 - x) ** (beta - 1) / B(alpha, beta)
```