Prob 1 | Independence: $P(E \cap F) = P(E)P(F)$

Are $E = \{1,2,3\}$ $F = \{4,5,6\}$ independent?

The probability of E on a fair 6 sided die is $\frac{1}{6} + \frac{1}{6} + \frac{1}{6} = \frac{3}{6} \neq \frac{1}{2}$

Likewise, probability of $F = \frac{1}{2}$.

So: $P(E)P(F) = \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4}$ → if they were independent.

However, $\{1,2,3\} \cap \{4,5,6\} = \emptyset$ → there are NO elements in common here!

So $P(\emptyset) = 0 \neq \frac{1}{4}$ P.o. => so, by the definition, the event cannot be considered independent.

Conditional Probability:

$$P(E|F) = \frac{P(E \cap F)}{P(F)} = \frac{\emptyset}{1/2} = 0$$

Prob 2 | Calculation Time!

(a) $X \rightarrow X(\text{head}) = 1,\ X(\text{Tail}) = -1$

$$E[x] = \sum_{i=0}^{1} x_i P_x(x_i) = (0.5)(1) + (0.5)(-1) = \boxed{0 = E[x]}$$

$$\text{Var}[x] = \sum_{i=0}^{1} (x_i - \cancel{E[x]}^{\,0})^2 P_x(x_i) = \sum_{i=0}^{1} x_i^2 P_x(x_i)$$

$$= (0.5)(1)^2 + (0.5)(-1)^2 = \boxed{1 = \text{Var}[x]}$$

(b) $Y \rightarrow$ 11 sided fair die, $Y(j) = j$ for $j = \{1, 2 \ldots 11\}$.

$$E[Y] = \underbrace{1/11}_{\text{fair die.}}(1+2+3+4+5+6+7+8+9+10+11) = 66/11 = \boxed{6 = E[Y]}$$

$$\text{Var}[Y] = \sum_{i=0}^{10} (x_i - 6)^2 P_x(x_i).$$

$$= 1/11\big((\overset{-5}{1-6})^2 + (\overset{-4}{2-6})^2 + (\overset{-3}{3-6})^2 + (\overset{-2}{4-6})^2 + (\overset{-1}{5-6})^2 + (\overset{0}{6-6})^2$$
$$+ (\overset{1}{7-6})^2 + (\overset{2}{8-6})^2 + (\overset{3}{9-6})^2 + (\overset{4}{10-6})^2 + (\overset{5}{11-6})^2\big)$$

$$= 1/11\,(25+16+9+4+1+0+1+4+9+16+25$$

$$= 110/11 = \boxed{10 = \text{Var}[Y]}$$

So.

$$P(Z_1) \begin{cases} 1/22 \text{ if } Z = \{0, 1, 11, 12\} \\ 2/22 \text{ else} \end{cases}$$

As a sanity check:

$$P(S) = 4 \cdot \frac{1}{22} + 9 \cdot \frac{2}{22} = \frac{22}{22} = 1 \checkmark$$

Prob 3 | Markov / Cheby. → makes sense?

(a) $P(X \geq 1)$?

Markov: $P(X \geq \delta) \leq \frac{E[X]}{\delta} \Rightarrow P(X \geq 1) \leq \frac{0}{1} = 0$

→ This does not make sense. Markov & Cheby only work if X is a non-negative discrete random var.

→ Note that any calculation here will yield 0:

$P(X \geq -1) \leq \frac{0}{-1} = 0$ → that is just not true.

(b) $P(Y \geq 9)$

Markov: $P(Y \geq 9) \leq \frac{6}{9} = \frac{2}{3} \approx 66\%$.

→ This probability bound does make sense. Even just from a cursory glance of the equally likely outcomes

$\{1, 2, 3, 4, 5, 6, 7, 8, [9, 10, 11]\}$

$\frac{1}{11} + \frac{1}{11} + \frac{1}{11} = \frac{3}{11} \approx 27\%$

✓ $27\% \leq 66\%$ ←

(c) $P(|Y-6| \geq 2)$

Chebyshev: $P(|Y - E[Y]| \geq \delta) \leq \frac{Var[Y]}{\delta^2}$

$\Rightarrow P(|Y-6| \geq 2 \leq \frac{10}{4} \approx 2.5$

→ this result does not make sense. Probabilities are $\in [0, 1]$, and so 2.5 as an upper bound is not useful. The variance is so large that it renders this bound useless.

Prob 4 | Prob of Flush & Straight.

FLUSH:

Let our elementary events be selections of 5 cards.

Therefore, $|S| = {}_{52}C_5$

And as choosing any 5 card combination is equally likely,

$$P(\{\text{five cards}\}) = \frac{1}{|S|} = \frac{1}{{}_{52}C_5} = \frac{1}{\frac{52!}{(52-5)!5!}} = \frac{47!5!}{52!}$$

Now, the number of flushes in set is:

$|\text{Flushes}| = (\text{4 suits to choose from}) * (\text{Choose 5 of 13 possible cards})$.

$$|\text{Flushes}| = 4 \cdot {}_{13}C_5 = 4 \cdot \frac{13!}{(13-5)!5!} = 4 \cdot \frac{13!}{8!5!}$$

And so probability of flush is:

$$P(\text{flush}) = |\text{Flush}| \cdot \frac{1}{|S|} = 4 \cdot \frac{13!}{8!5!} \cdot \frac{47!5!}{52!}$$

$$= 4 \cdot \frac{13 \cdot 12 \cdot 11 \cdot 10 \cdot 9 \cdot 8!}{8!} \cdot \frac{47!}{52 \cdot 51 \cdot 50 \cdot 49 \cdot 48 \cdot 47!}$$

$$= \frac{4 \cdot 13 \cdot 12 \cdot 11 \cdot 10 \cdot 9}{52 \cdot 51 \cdot 50 \cdot 49 \cdot 48} \approx 0.00198$$

Therefore: $P(\text{flush}) \approx 0.20\%$

STRAIGHTS:

→ For this, we will will include ace high & low options as valid, and we will not filter out straight flushes (straights w/ all same suit).

We will keep the same elementary counts & same $P(5 \text{ cards}) = \frac{1}{{}_{52}C_5}$.

Consider the cards in order (ace high & low included):

- We also can have any card be any possible suit: C1 C2 C3 C4 C5
  $4 * 4 * 4 * 4 * 4$
  - 4^5 options of suit for all 5 cards -

So, $P(\text{Straight}) = 4^5 \cdot 10 \cdot \frac{1}{151} = 4^5 \cdot 10 \cdot \frac{47! \, 5!}{52 \cdot 51 \cdot 50 \cdot 49 \cdot 48 \cdot 47!}$

$= \frac{4^5 \cdot 10 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1}{52 \cdot 51 \cdot 50 \cdot 49 \cdot 48} \approx 0.0039 \approx$ **0.39% Chance of straight**

Prob 5 | Expected runtime of randomized insertion sort.

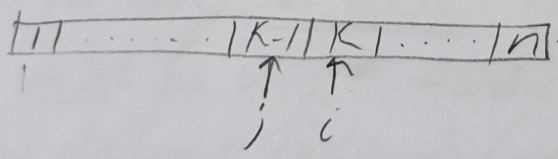
We will assume the random permutation has n distinct numbers such that $x_1 < x_2 < \ldots < x_n$.

Let X_{ij} be a variable to indicate if elements i & j are swapped during the run of insertion sort, such that:

$$X_{ij} \begin{cases} 1 \rightarrow i \& j \text{ are swapped} \\ 0 \rightarrow \text{no swap occurs.} \end{cases}$$

Since the permutation is random, the chance that 2 numbers will be $\frac{1}{2}$ ($P(X_{ij}=1) = P(X_{ij}=0) = 1/2$). In other words, there is an equal chance that 2 numbers will be swapped during each iteration or not.

The number of swaps in a run of insertion sort can be thought of by considering how it runs. Consider a K^{th} iteration.



There is the potential in this iteration to make exactly $K-1$ swaps. So, for every element of the array, we can express the potential number of swaps as:

$$\sum_{i=1}^{n-1} \sum_{j=1}^{i-1} X_{ij} \rightarrow \text{What is the expectation of } X_{ij}?$$

$$E\left[\sum_{i=1}^{n-1}\sum_{j=1}^{i-1} X_{ij}\right] = \sum_{i=1}^{n-1}\sum_{j=1}^{i-1} E[X_{ij}] = \sum_{i=1}^{n-1}\sum_{j=1}^{i-1} \frac{1}{2} = \frac{1}{2}\sum_{i=1}^{n-1} i-1 = \frac{1}{2}\sum_{i=0}^{n-2} i$$

$$E[X_{ij}] = (1)(\tfrac{1}{2}) + (0)(1/2) = \boxed{1/2}$$

$= \frac{1}{2}(1+2+3+\ldots+n-2) = \frac{1}{2}\left(\frac{(n-2)(n+1)}{2}\right)$ $n=5$

$= \frac{n^2-3n+2}{4} = \boxed{O(n^2)}$

This the expected runtime with a randomization involved. We can see that randomization does not help insertion sort in the same way it "improves" quicksort.

Prob 6 | Vertex Cover

(a) In the first iteration, edge e is chosen. By definition, the first choice has to guaranteed be at least u or v in the vertex cover C*. Worst case, we choose the endpoint that is not in minimum C* and it is an auxiliary addition to C. Best case, we choose the endpoint that is in C*. So, we have 2 choices with equal probability since randomly chosen

{ extra vertex added
  vertex in C* added } > Probability of either then must be $\frac{1}{2}$

(b) By the definition of the algorithm, the next edge endpoints cannot contain the last one added (either u or v, depending). So, we have a distinct choice with the same options listed in (a) → and similarly, at least 1 end point now considered in the pair must be in C* in order to include this edge (since distinct endpoints from one chosen in (a)). So, the probability that the end point chosen in the second iteration remains the same: $\frac{1}{2}$

(c) Let X be a random variable indicator of whether the end point added in any iteration is in C^* or not:

$X \begin{cases} 1 \rightarrow \text{is in } C^* \\ \emptyset \rightarrow \text{is NOT in } C^*. \end{cases}$

Let $k = |C^*|$. Then, the sum $\sum_{i=1}^{|C|} X_i$ cannot be more than k.

So, we can say:

$$\sum_{i=1}^{|C|} X_i \leq k.$$

Let's find the expected value of X. Thus far, and further by the same logic in parts (a) & (b), the chance that the endpoint added to C is $\frac{1}{2}$ in every iteration. Or:

$$P(X=1) = P(X=0) = \frac{1}{2}.$$

So:

$$E\left[\sum_{i=1}^{|C|} X_i\right] \leq k \rightarrow (E[k] = k)$$

$$\sum_{i=1}^{|C|} E[X_i] \leq k$$

$E[X_i] \rightarrow (0)(\frac{1}{2}) + (1)(\frac{1}{2}) = \frac{1}{2}.$

$$\sum_{i=1}^{|C|} \frac{1}{2} \leq k \Rightarrow \frac{1}{2}|C| \leq k.$$

$$|C| \leq 2k.$$

$$\boxed{E[|C|] \leq 2k.}$$

expected size of C is upper bounded by $2k$, or $2(|C^*|)$.

Prob 7 | n digit num mult.

(a) The mathematical meaning of $\Omega(n^2)$ is simplest as:
Let $f(n)$ be an arbitrary function.

$$cn^2 \leq f(n) \text{ for all } n \geq n_0, \quad c \in \mathbb{R}.$$

(b.) You can absolutely treat the multiplacation of 2 n-digit numbers as polynomial multiplacation.

Consider: $412 \times 308 = 126896$.

Let's define: $f(x) = 4x^2 + x + 2$
$g(x) = 3x^2 + 8$
} notice coefficients are $[2, 1, 4]$ & $[8, 0, 3]$.
$a_0 \longrightarrow a_n$ order

Now: $f(x) * g(x) = (4x^2 + x + 2)(3x^2 + 8)$
$= 12x^4 + 32x^2 + 3x^3 + 8x + 6x^2 + 16$.

$h(x) = f(x) * g(x) = 12x^4 + 3x^3 + 38x^2 + 8x + 16$.

And finally,

$$h(10) = 12(10)^4 + 3(10)^3 + 38(10)^2 + 8(10) + 16.$$
$$= 126896 = 412 \times 308 \checkmark$$

So, steps are to (1) represent n digit number as polynomial coefficients (appropriate ordering)!
(2) multiply the 2 resulting polynomials,
(3) evaluate resulting polynomial @ $x = 10$

You can therefor represent the multiplacation as a convolution by:

coefficients: $c_k = \sum_{i=0}^{k} a_i b_{k-i}$, where a is the first n digit number as "coefficients" and b is the second n digit number as "coefficients".

→ important note above is to pad $\vec{a}$ & $\vec{b}$ with zeros as necessary to have max(k) values. In our example prior, $\vec{a}$ & $\vec{b}$ would be [2, 1, 4, 0, 0] & [6, 0, 3, 0, 0].

(1.) The key thing is that we can use FFT to multiply 2 polynomials in O(n log n) time. And since we have now seen that we can represent an n digit number as a polynomial, we can of course use FFT to multiply 2 n digit numbers O(n log n) time.

So, all we need to do is (let n = degree of a & b).

(1) write n digit numbers a & b as coefficients of a polynomial appropriately. Let these coeff lists be A and B. Pad w/ zeros as needed to achieve equal length (and pad to at least 2n in length)

(2.) Convert A to point-form using FFT } generate 2n
Convert B to point-form using FFT } points.

(3.) multiply A & B point-wise using 2n points generated by FFT in step (2)

(4.) Use the result of (3) and run it through FFT^{-1}. to get back the resulting A*B coefficient vector.

(5.) evaluate this resulting polynomial from (4) at x = 10. (can use nested multiplacation to evaluate w/ O(n) multiplications).

The leading run time factors are steps:

(2) → 2 calls to FFT $2^*\Theta(n \log n)$.

(3) → point-wise mult is over $2n$ points, but still linear: $\Theta(n)$.

(4) Use FFT^{-1} on point-wise mult: $\Theta(n \log n)$.

(5) evaluate nested polynomial → $O(n)$.

So, run time tallies to $T(n) = 3\Theta(n \log n) + 2O(n) + C$.

$= \boxed{\Theta(n \log n)}$.

(See Cormen p. 891-892 on details for FFT & FFT^{-1} being $\Theta(n \log n)$. and generally that section for FFT algorithm).

Also, will attach below an implementation of this in a python script — unclear on how much detail I need to go into on the FFT algorithm itself, so hoping that attaching that in addition is thorough enough to answer this question. Just ignore below this if you don't need that :).

CODE/IMPLEMENTATION

```
import cmath

i = complex(0, 1)
pi = cmath.pi

# FFT implementation
def FFT(coeffs, invert):
    n = len(coeffs)
    if n == 1:
        return

    # even/odd coefficients of given coeff array
    even_coeff = coeffs[::2]
    odd_coeff = coeffs[1::2]

    # Recursive FFT on both halves
    FFT(even_coeff, invert)
    FFT(odd_coeff, invert)

    # get your roots of unity to evaluate at to make your points
    inversion_factor = 1
    if invert: inversion_factor = -1

    angle = 2 * pi / n * inversion_factor # 1/w if inverting! otherwise
normal exp
    w = complex(1, 0) # set up the omega as 1 to initialize
    wn = cmath.exp(complex(0, angle)) # e ^ 2(pi)(i)/n or e ^ -2(pi)(i)/n
(if invert)

    # get your points or coeffs
    for i in range(n // 2): # from 0 -> n/2 - 1
        t = w * odd_coeff[i]
        coeffs[i] = even_coeff[i] + t # positive pairing
        coeffs[i + n // 2] = even_coeff[i] - t # negative pairing

        if invert: # inversion adjustment
```

```
            coeffs[i] /= 2
            coeffs[i + n // 2] /= 2

        w *= wn # accumulate to next root of unity


# multiply two polynomials using FFT
def mult_poly(A, B):
    # pad out to n + m in length to fit the new polynomial of degree at
least n + m
    n = len(A)
    m = len(B)
    new_degree = 1
    while new_degree < n + m:
        new_degree <<= 1 # get this to a multiple of 2

    # pad with high order zero coefficients, convert to complex
    fftA = [complex(A[i] if i < n else 0, 0) for i in range(new_degree)]
    fftB = [complex(B[i] if i < m else 0, 0) for i in range(new_degree)]

    # apply fft to get 2n points at roots of unity
    FFT(fftA, False)
    FFT(fftB, False)

    # point-wise multiplication
    for i in range(new_degree):
        fftA[i] *= fftB[i]

    # invert! get the coeffs back and place into
    FFT(fftA, True)

    # round the reals--round off errors likely
    for i in range(new_degree):
        fftA[i] = round(fftA[i].real)

    # strip the padding from the end
    i = new_degree - 1
    while fftA[i] == 0:
        fftA.pop()
        i -= 1
```

```
    return fftA

def eval(P, x):
    # evaluate a polynomial at a given x
    # should be a list of coefficients, need to process
    # backwards since its given lowest to highest ordering
    # use horners method/nested mult to achieve linear mult
    n = len(P) - 1
    acc = P[n]
    for i in range(n - 1, -1, -1):
        acc = (acc * x) + P[i]
    return acc

# SCRIPT RUN

# comment out any A or B to try different combos

# 341 * 1 = 341
A = [1, 4, 3]
B = [1]

product = mult_poly(A, B)
sol = eval(product, 10)
print(sol)

# 21 ^ 2 = 441
A = [1, 2]
B = [1, 2]

product = mult_poly(A, B)
sol = eval(product, 10)
print(sol)

# 4037 * 65 = 262405
A = [7, 3, 0, 4]
B = [5, 6]

product = mult_poly(A, B)
sol = eval(product, 10)
```

```
print(sol)

# 12350960 * 53 = 654600880
A = [0, 6, 9, 0, 5, 3, 2, 1]
B = [3, 5]

product = mult_poly(A, B)
sol = eval(product, 10)
print(sol)

# 2 * 4 = 8
A = [4]
B = [2]

product = mult_poly(A, B)
sol = eval(product, 10)
print(sol)
```

OUTPUT

```
PS C:\Users\lutzr\CS362\HW3> python .\7_fft_poly.py
341
441
262405
654600880
8
PS C:\Users\lutzr\CS362\HW3>
```