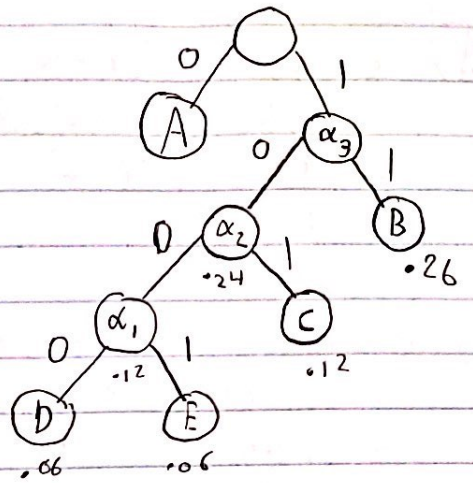
youssef
Naguib

Hw - 3

1)

a)

symbol	frequency
X A	.50
X B	.26
X C	.12
X D	.06
X E	.06
X α_1	.12
X α_2	.24
X α_3	.50



code:

symbol	binary huffman code
A	0
B	11
C	101
D	1000
E	1001

b) bits per symbol:

$= \sum$ Length(symbol) * probability

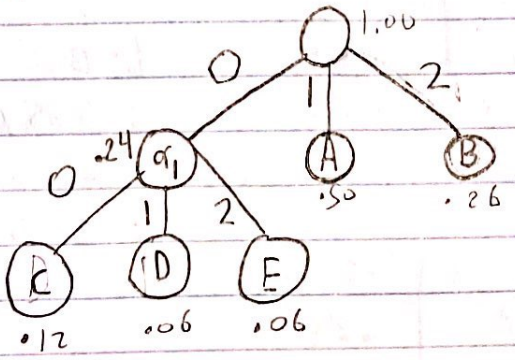
$= (.5 \times 1) + (.26 \times 2) + (.12 \times 3) + (.06 \times 4) + (.06 \times 4)$

$= .5 + .52 + .36 + .24 + .24$

$= 1.86$ bits per symbol

c)

symbol	frequency
A	.50
B	.26
X C	.12
X D	.06
X E	.06
α_1	.24



code:

symbol	ternary huffman code
A	1
B	2
C	00
D	01
E	02

d) expected trits per symbol:

$= (.50 \times 1) + (.26 \times 1) + (.12 \times 2) + (.06 \times 2) + (.06 \times 2)$

$= .5 + .26 + .24 + .12 + .12$

$= 1.24$

e) expected bits per symbol:

= expected trits per symbol $\times$ 1.585

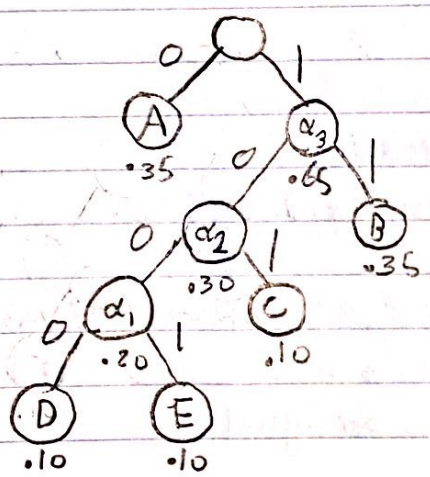
$= 1.24 \times 1.585$

$= 1.9654$ bits per symbol

part 2 of Question 1

symbol	frequency
A	.35
B	.35
C	.10
D	.10
E	.10
α_1	.20
α_2	.30
α_3	.65

a)



code:

symbol	Binary Huffman code
A	0
B	11
C	101
D	1000
E	1001

b) Bits per symbol:

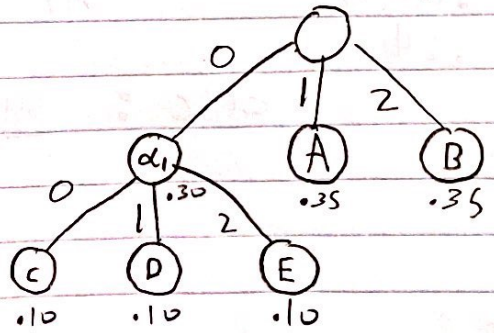
$= (.35 \times 1) + (.35 \times 2) + (.10 \times 3) + (.10 \times 4) + (.10 \times 4)$

$= .35 + .70 + .30 + .40 + .40$

$= 2.15$ Bits per symbol

c)

symbol	frequency
A	.35
B	.35
C	.10
D	.10
E	.10
α_1	.30



code:

symbol	ternary huffman code
A	1
B	2
C	00
D	01
E	02

d) expected trits per symbol:

$= (.35 \times 1) + (.35 \times 1) + (.10 \times 2) + (.10 \times 2) + (.10 \times 2)$

$= .35 + .35 + .20 + .20 + .20$

$= 1.30$ trits per symbol

e) expected bits per symbol

= expected trits per symbol $\times$ 1.585

$= 1.30 \times 1.585$

$= 2.0605$ bits per symbol

Problem 1 part 3

a) $H(x) = -\sum_x p(x) \log_2(p(x))$

$= -1 (.5 \log_2(.5) + .26 \log_2(.26) + .12 \log_2(.12) + .06 \log_2(.06) + .06 \log_2(.06))$

$= -1 (-1.8594227679976008)$

$= 1.8594$

b) $H(x) = -\sum_x p(x) \log_2(p(x))$

$= -1 (.35 \log_2(.35) + .35 \log_2(.35) + .10 \log_2(.10) + .10 \log_2(.10 + .10 \log_2(.10))$

$= -1 (-2.0567796494470395)$

$= 2.0568$

Problem 1 part 4

The higher the entropy, the more "unknown" the value is. Example 1 achieved an efficiency closer to entropy, for both the binary and ternary huffman encoding. Both binary and ternary expected efficiencies are linearly related, as demonstrated, the expected bits per symbol can be estimated by multiplying the expected trits per symbol by 1.585. Based on these takeaways, both a binary and ternary huffman code are best suited to encode a probability set that has less duplicate values and is more spread out. Example 1 had less duplicate probabilities, and higher probabilities. Higher probabilites using a low number of bits will drive down the entropy and expected bits per symbol. Although both examples had equal amounts of symbols and bits used, ultimately the example with a higher, decreasing ordering of probabilities was more efficient.

Problem 2

strategy 1: $s=0$, $F=10$ intervals = $(0,3), (0,8), (8,10)$

The strategy would select the earliest ending interval $(0,3)$ and discard $(0,8)$ because it overlaps. The strategy would then select $(8,10)$ as it is the next earliest ending and doesn't overlap. so the selected intervals would be $(0,3)$ and $(8,10)$ leaving a gap time of 5. An optimal solution would've discarted $(0,3)$ and selected $(0,8)$ alongside $(8,10)$, producing a gap time of 0. This is a proof that strategy 1 does not always work by counter example.

Strategy 2:

$s = 0 \quad f = 10 \quad$ intervals: $(0,3), (1,7), (7,10)$

This strategy would select the earliest starting interval $(0,3)$ and discard $(1,7)$ because it overlaps. The algorithm would then select the next earliest starting interval, which happens to be the only one left: $(7,10)$. The selected intervals using this method are $(0,3)$ and $(7,10)$ leaving a total gap time of 4. An optimal solution would have chosen $(1,7)$ and $(7,10)$ as the total gap time would only be 1, even though the interval selected was not the earliest to start. This is a proof that strategy 2 doesn't always work by counter example.

Strategy 3: $s = 0 \quad F = 20 \quad$ Intervals: $(0,14), (10,15), (15,17), (16,20)$

This strategy would first select the 2 non-overlapping intervals with the smallest gap in between them. This would lead to $(10,15)$ and $(15,17)$ being selected, as they have the smallest gap (0). The algorithm would then discard overlapping intervals, those being $(0,14)$ and $(16,20)$. Before even reaching the recursive call, this algorithm has discarded the optimal solution. This strategy would result in the only selected intervals being $(10,15)$ and $(15,17)$ leaving a gap time of 13. The optimal solution given these 4 intervals would be to choose $(0,14)$ and $(16,20)$, leaving a gap time of 2. This is a proof that strategy 3 may not always provide the optimal solution, by counter example.

Youssef Naguib <ymn4543@rit.edu>
CS261
Write-up for hw-3 questions 3-5

3.
My algorithm calculates the number of right triangles given n 2d points. At first I was not sure how to do this without iterating through every triplet possible and using trigonometry or linear algebra. After much thought I devised a much more efficient algorithm. The total right angles variable is set to 0 to begin with. The algorithm iterates through every point and calculates the slope to each other point, keeping track of undefined slopes. The slopes are stored in an array, and sorted using merge sort. After the array is sorted, the algorithm iterates through each slope and find the inverse slope by using the formula (-1/slope). If the slope is 0, the number of right angles possible at that angles is equal to the number of undefined slopes. Otherwise, Binary search is used on the slope list to find if the inverse slope is in the slope list. If it is, neighboring indices are also checked to see if there are duplicate inverse slopes. The right triangle variable is increased accordingly.
This algorithm is valid for the following observations:

- For every point, every slope to other point is calculated and stored in an array. Every right angle possible at that vertex can be found using these slopes, as any two inverse slopes that are both in the array are perpendicular.
- To account for floating point errors, I store slopes as mini arrays that represent fractions, where the numerator is the first element and the second is the denominator. This way the inverse of the inverse is never calculated and no floating point errors are possible. My modified merge sort stores the slopes by dividing the numerator and denominator, but the actual decimal slope is not used for comparison.
- For any slopes equal to 0, the number of right angles possible with that slope would be undefined slopes. This is a variable I keep track of. All possibilities are accounted for.

Running time analysis:

- Iterate through each point O(N)
- Iterate through all other points and find slope O(N)
- Use Merge sort on slopes O(NlogN)
- Binary search on N-1 slopes O(logN)
- Many O(1) operations in between
  Total complexity is O(N) * (O(N) + O(NlogN) + O(logN) + O(1)) = O(N) * O(NlogN)
  = O(N^2logN)

4.

The solution to the candy problem is a greedy one. We are spending all the money we have, and the goal is to purchase as many types of candies possible. Each candy has an associated price, and we want to maximize the types of candy bought, not the number of candies bought. (This means we would not want to buy the same type twice). My approach to this problem was to buy the cheapest candy available, and repeat until no more candies could be bought. This is done using a greedy algorithm. First the candy prices are stored in an array. The select_rand() function is used to find the k'th smallest element in an array, so I set k = 0 to find the cheapest candy. I then subtracted that candy's price from the allowance and removed that candy price from the array of prices, so that it's never bought again. This process is repeated, and k never changes, until an candy price is returned that is too expensive to be bought with the remaining allowance. This approach is valid because it maximizes the number of candies bought by choosing the cheapest candy each time, minimizing cost.

Running time analysis:

- Select_rand() has an expected running time of O(N), and it is repeated a constant number of times.
- Total running time = C * O(N) = O(N)

5.

Heart of the Algorithm:

tile_list is the dynamic programming array

1- **Verbal description**: tile_list[k] = number of possible tiling's for a backsplash of dimensions 2*k.

2- **Solution:** Solution will be found at tile_list(n) where n is the width of the backsplash wall.

3- **Recurrence:** tile_list[k] = (tile_list[k-1]) + (4 * tile_list[k-2]) + (2 * tile_list[k-3])

Base cases:

K	tile_list[k]
0	1
1	1
2	5

Running time analysis:

For each subproblem in range of 3 to k, where tile_list[k] represents the solution, we do a constant amount of work .

= O(N) * O(1) = O(N)