1. Solve: $T(n) = 2T(n-1) + 3T(n-2)$ for $T(0)=1$, $T(1)=2$

Solve using characteristic equation:

→ First find the characteristic equation:

Assuming a solution in the form $T(n)=r^n$
↳ substitute into the recurrance

$T(n)=2T(n-1)+3T(n-2) \Rightarrow r^n = 2r^{n-1} + 3r^{n-2}$

↳ simplify

$(r^n = 2r^{n-1} + 3r^{n-2}) \cdot 1/r^{n-2}$

↳ $r^2 = 2r + 3 \rightarrow \underline{r^2 - 2r - 3 = 0}$

now we can use $r^2 - 2r - 3 = 0$ to find r

$-3 \times 1 \quad -3 + 1 = -2$

$(r-3)(r+1) = 0$

↳ $r = -1, 3$

Since the roots are real & distinct, the solution will generally be $T(n) = A(3^n) + B(-1)^n$ where A & B are coefficients. we can find these coefficients using $T(0)$ & $T(1)$:

$T(0) = A(3^0) + B(-1)^0 = A \cdot 1 + B \cdot 1 = \underline{A + B = 1}$

$T(1) = A(3^1) + B(-1)^1 = A \cdot 3 + B \cdot (-1) = \underline{3A - B = 2}$

$A + B = 1 \rightarrow B = 1 - A$

$3A - B = 2 \rightarrow 3A - (1-A) = 2 \rightarrow 3A - 1 + A = 2 \rightarrow 4A = 3 \rightarrow A = 3/4$

$B = 1 - A \rightarrow B = 1 - (3/4) \rightarrow \underline{B = 1/4}$

Now that A & B are solved just sub into general form

From earlier: $\boxed{T(n) = 3/4(3^n) + 1/4(-1)^n}$

2. Prove that the binary addition can be reduced to prefix computation.

What are we trying to prove: We want to prove that binary addition of 2 n-bit numbers can be computed using a prefix computation technique.

↳ Binary addition with carry propagation

→ $A = a_{n-1}a_{n-2}\ldots a_0$ $B = b_{n-1}b_{n-2}\ldots b_0$ ⇒ A & B are the binary numbers of n-bit length where a_i / b_i represent a bit/digit

In binary addition we are computing $C = A+B$, which will involve computing the sum bits and carry bits for each position 'i' ← define core problem

The sum bit at position i, or s_i can be found with $s_i = a_i \oplus b_i \oplus c_i$ where c_i is the carry bit to position i

In other words, in order to find the sum bit we need the carry bit

We can make a recursive expression to represent the carry out: $c_{i+1} = (a_i \wedge b_i) \vee ((a_i \oplus b_i) \wedge c_i)$, since this is recursively defined we need to make its recursive structure more manageable.

This can be done by creating helper functions both at position i, the generate & propogate bit ← define generate and propogate

↳ Generate bit ⇒ $g_i = a_i \wedge b_i$ generates carry-out regardless of carry out

↳ Propogate bit ⇒ $p_i = a_i \oplus b_i$ passes a carry bit if one is recieved

Putting these back into the carry bit equation we get: $c_{i+1} = g_i \vee (p_i \wedge c_i)$

With this equation we now have a Uniform Recurrence at every bit position, in terms of just g_i, p_i, and the previous carry bit

So we now have a way to find the carry for each position, but we need to do some more work to make it a formal prefix computation ← prefix computation structure

The structure of $c_{i+1} = g_i \vee (p_i \wedge c_i)$ resembles a prefix computation, which is any operation over a list where each result depends on all previous elements in some associative way

To make this a formal prefix computation, we group the values at each bit into pairs → (g_i, p_i)

We then use the binary operator $\otimes$ on these pairs, written below

↳ $(g_i, p_i) \otimes (g_j, p_j) = (g_i \vee (p_i \wedge g_j), p_i \wedge p_j)$ → From recursive definition of carry bit & generate/propogate definitions

↳ This definition is used as it shows how carries propogate across bit positions, it is also associative which is required for prefix computation

We can then compute the prefix using this definition from least significant up to position i, doing so we can find the carry in for each position

(g_0, p_0), $(g_0, p_0) \otimes (g_1, p_1)$ → c_1, $(g_0, p_0) \otimes (g_1, p_1) \otimes (g_2, p_2)$ → c_2, … → c_i, $(g_0, p_0) \otimes \ldots \otimes (g_i, p_i)$ → c_0

↳ once the carry in 'c_i' for each bit is found from the prefix values we can find the sum of the bit with $s_i = a_i \oplus b_i \oplus c_i$

↳ Pulling everything together, binary addition reduces to prefix computation because the carry bits follow a recursive pattern that can be expressed using generate and propogate logic, and combined across bit positions using an associative operator. This allows the sequential carry propogation to be computed in parallel through prefix sum.

Prefix computation: We have a sequence like $X = [x_0, x_1, x_2, \ldots, x_{n-1}]$ and a binary associative operator like $\oplus$ to produce the prefix computation: $Y = [x_0, x_0 \oplus x_1, x_0 \oplus x_1 \oplus x_2, \ldots, x_0 \oplus \ldots \oplus x_{n-1}]$.

For example with +: $[1, 2, 3, 4] \rightarrow [1, 1+2, 1+2+3, 1+2+3+4] = [1, 3, 6, 10]$

Shorter summary:

Goal: Add two n-bit numbers A and B using prefix computation to handle carry bits

Define p/g:

Generate → $g_i = a_i \wedge b_i$

Propogate → $p_i = a_i \oplus b_i$

Carry Recurance:

$c_{i+1} = g_i \vee (p_i \wedge c_i)$

Define associative operator:

$(g_i, p_i) \otimes (g_j, p_j) = (g_i \vee (p_i \wedge g_j), p_i \wedge p_j)$

Then use prefix computation over (g_i, p_i) to compute all carry-ins c_i in parallel.

Compute sum bits:

$s_i = a_i \oplus b_i \oplus c_i$

The carry logic in binary addition can be found with generate and propogate bits and combined using an associative operator, allowing the entire addition to be performed in parallel through prefix computation.

3a. Define fixed length and variable length codes

i) **Fixed length code**: A fixed length code assigns each symbol the same number of bits, regardless of how frequently the symbol appears

For example, ASCII encoding uses 8 bits for every character

Advantage: Simple and fast decoding
↳ Since the position of each symbol is always predictable

Disadvantage: Not space efficient when the frequencies of symbols vary

ii) **Variable-length codes**: A variable length code assigns different numbers of bits to symbols depending on their frequency; shorter codes for more frequent symbols, and longer codes for less frequent ones

For example, Huffman coding uses variable length codes for lossless data compression

Advantage: More space-efficient for data compression

Disadvantage: Requires pre-fix free properties ∴ a more complex decoder

a set of binary codes in which no code is a prefix of another, this is important as it ensures codewords are not confused as the beginning of others

b. Design Huffman code for the following

Alphabets	a	b	c	d	e	f
Number of occurrences X 10^2	20	15	13	40	29	6

Initialize Min-Heap with all Symbols: insert the symbols into a min-heap ordered by frequency
↳ [(f, 6), (c, 13), (a, 20), (e, 29), (d, 40)]

Build tree: remove 2 lowest-frequency nodes, merge them and reinsert

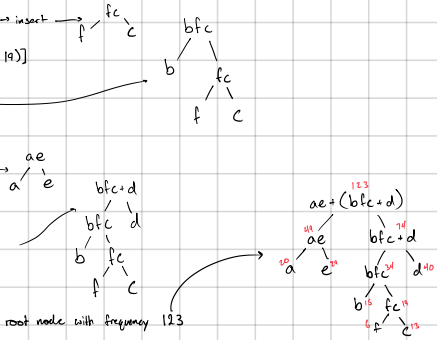
Pop (f, 6) & (c, 13) → merge → (fc, 19) → insert → [(b, 15), (a, 20), (e, 29), (d, 40), (fc, 19)]

Pop (b, 15) & (fc, 19) → (bfc, 34) → [(a, 20), (e, 29), (d, 40), (bfc, 34)]

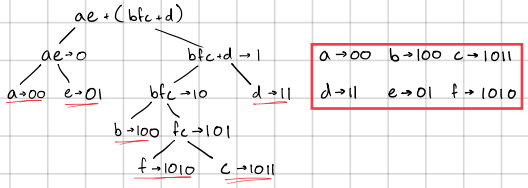
Pop (a, 20) & (e, 29) → (ae, 49) → [(d, 40), (bfc, 34), (ae, 49)]

Pop (bfc, 34) & (d, 40) → (bfc+d, 74) → [(ae, 49), (bfc+d, 74)]

Pop (ae, 49) & (bfc+d, 74) → merge into root node with frequency 123



Assign Binary codes: now assign binary digits going left=0, right=1 recursively



a→00 b→100 c→1011
d→11 e→01 f→1010

c. Compute the compression ratio

Compression ratio: A measure of how much smaller the encoded data becomes compared to the original → A larger ratio indicates better compression

Find original size:

each symbol will have a fixed-length code
↳ 6 total symbols → 3 bits per symbol

multiply the frequency of each symbol (number of occurrences) by the bit length (3)
↳ Original size = $\sum$(frequency) x3 = (20+15+13+40+29+6) x3 = 123x3 = 369 bits

Find Huffman (compressed) size:

multiply symbol frequency and code length
↳ so pretty much the same as original, but bit length changes

compressed size = 20·2 + 15·3 + 13·4 + 40·2 + 29·2 + 6·4

= 40 + 45 + 52 + 80 + 58 + 24 = 299 bits

Compute compression ratio

$\frac{L_F - L_V}{L_F} \cdot 100$ = compression ratio

$\frac{369-299}{369} \cdot 100 = \frac{70}{369} \cdot 100 \approx 18.97\%$

This means the Huffman encoding reduces the total size by roughly 18.97% compared to using a fix-length 3-bit encoding

d. Pick a five-letter word from this alphabet set. Encode and decode it.

Word = faced

Encode:

f	a	c	e	d
1010	00	1011	01	11

encoded string: 10100010110111

Decode:

since the codes for each symbol is prefix free we can just read the input bit by bit until a symbol with that code is found

encoded string: 10100010110111

1 → nothing	0 → x	1 → x	0 → x	1 → x
10 → x	00 → a	10 → x	01 → e	11 → d
101 → x	↳ 10110111	101 → x	↳ 11	↳ empty
1010 → f		1011 → c		
↳ 00101101111		↳ 0111		

combine decoded symbol → faced

4a. Define $\log^*n$, find its value for $n=10^{12}$. Let $n=2^k$ and $f(n)=n/2$. Find $f^*(n)$

Define $\log^*(n)$

$\log^*(n)$ is the iterated logarithm function which is the number of times the logarithm function ($\log_2 n$) must be applied to n before the result is ≤ 1 $\Rightarrow$ $\log^* n = \begin{cases} 0 & \text{if } n \leq 1 \\ 1+\log^*(\log_2 n) & \text{if } n > 1 \end{cases}$

Compute $\log^*(10^{12})$

Take base 2 logs:

- $\rightarrow \log_2(10^{12}) = 12 \cdot \log_2 10 \approx 12 \cdot 3.32 = 39.86 > 1$
- $\rightarrow \log_2(39.86) \approx 5.32 > 1$
- $\rightarrow \log_2(5.32) \approx 2.41 > 1$
- $\rightarrow \log_2(2.41) \approx 1.27 > 1$
- $\rightarrow \log_2(1.27) \approx 0.35 \leq 1 \rightarrow$ stop

↳ So $\log^*(10^{12}) = 5$

Find $f^*(n)$

The function $f^*(n)$ is the number of times we can apply $f(n)=n/2$ until the result is ≤ 1

$n=2^k \Rightarrow f(n) = 2^k/2 = 2^{k-1}$

$f^2(n) = f(f(n)) = 2^{k-1}/2 = 2^{k-2}$

$f^3(n) = 2^{k-2}/2 = 2^{k-3}$

$f^k(n) = 2^{k-k} = 2^0 = 1 \leq 1$ done

→ Then by definition of $f^*(n)$, it took k steps to reach 1

↳ $f^*(n) = k$

b. Let $n=2^{2^k}$. Solve $T(n) = T(\sqrt{n}) + 2$ where $T(2)=1$

Express in terms of k

$S(k) = T(2^{2^k}) = T(\sqrt{2^{2^k}}) + 2 = T(2^{2^{k-1}}) + 2 = S(k-1)+2$

↳ so we now have a simple recurrence in terms of $k \rightarrow S(k) = S(k-1)+2$

Base case

We are given $T(2)=1$, we need to find k such that $2^{2^k}=2$

↳ $2^{2^k}=2 \rightarrow \log_2(\cdot)$ both sides $\rightarrow 2^k=1 \rightarrow$ again $\rightarrow k=0$

$\Rightarrow$ So when $k=0$, we get $n=2^{2^0}=2^1=2$

$\Rightarrow S(0) = T(2) = 1$

Solve $S(k) = S(k-1)+2$

First order recurrence, meaning each term increases by a constant

↳ Solve the first few to find a pattern

- $\rightarrow S(0) = 1$
- $\rightarrow S(1) = S(0)+2 = 1+2 = 3$
- $\rightarrow S(2) = S(1)+2 = 3+2 = 5$
- $\rightarrow S(3) = S(2)+2 = 5+2 = 7$
- $\rightarrow \dots$

↳ From this we can see the pattern $S(k) = 2k+1$ which is a closed form solution to the recurrence

Convert $S(k)$ back to $T(n)$

We went from $T(n)$ to using $n=2^{2^k}$ for $S(k) = T(2^{2^k})$, so we need to express k in terms of n to get our final solution

↳ so starting with the given $n=2^{2^k}$ we take $\log_2$ of both sides

→ $\log_2 n = 2^k$

↳ take $\log_2$ again

→ $\log_2(\log_2 n) = k$

We then substitute this into $S(k)$:

↳ $T(n) = S(k) = 1+2(k) = 1+2 \cdot \log_2(\log_2 n)$

↳ So the final solution is:

$\Rightarrow T(n) = 1 + 2 \cdot \log_2(\log_2 n)$

5. Solve the six city TSP by finding the approximate tour. Describe your algorithm. Prove that the cost of the approximate tour is no more than twice the cost of optimal tour

Build complete weighted graph

The cost of travel between city pairs is the Euclidean distance → $dist(p,q) = \sqrt{(x_2-x_1)^2+(y_2-y_1)^2}$

↳ So we need to create a complete graph $G=(V,E)$

→ $V = \{a, b, c, d, e, f\}$

→ $E = \{(u, v) : u \neq v\}$

→ The edge weights will be the Euclidean distance between the points

Construct the minimum spanning tree (MST)

We need to connect all the cities with the smallest total edge cost and no cycles

↳ we can do this by using Prim's algorith to find the MST

↳ Start with node 'a'

↳ Repeatedly add the cheapest edge that connects to a new vertex without forming a cycle

iteration 1 : Tree = {a} edge = a→e cost = 2.236

2 : Tree = {a, e} edge = e→b cost = 2.828

3 : Tree = {a, e, b} edge = b→c cost = 2.236

4 : Tree = {a, e, b, c} edge = b→f cost = 2.236

5 : Tree = {a, e, b, c, f} edge = b→d cost = 3.162

Total MST cost = 2.236 + 2.828 + 2.236 + 2.236 + 3.162 = **12.698**

DFS of MST

Perform a DFS starting 'a' : a→e→b→c→f→d→a

↳ This is the approximate TSP tour

Compute approximate tour cost

a→e cost = 2.236

e→b cost = 2.828

b→c cost = 2.236

c→f cost = 4.000

f→d cost = 5.000

d→a cost = 4.123

Approximate tour cost = 2.236 + 2.828 + 2.236 + 4.000 + 5.000 + 4.123 = **20.423**

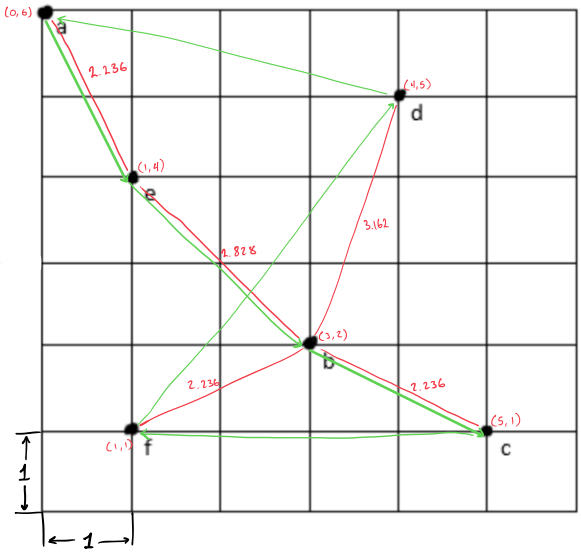
Prove 2x bound

→ The minimum spanning tree is always less than or equal to the optimal tour, meaning any tour must be at least as expensive as the MST since removing one edge from an optimal tour gives a spanning tree

→ DFS walk visits each edge at most twice, and shortcutting (skipping repeated visits) does not increase the cost due to the triangle inequality

∴ Approximate Tour cost ≤ 2 · MST

↳ 20.423 ≤ 25.396 ⇒ True, so this confirms that the tour is within the 2 times the theoretical upper bound, as guaranteed by the MST-based algorithm



All edges in complete graph

a→b = 5.0	b→c = 2.236	c→d = 4.123	d→e = 3.162	e→f = 3.0	
a→c = 7.071	b→d = 3.162	c→e = 5.00	d→f = 5.00		
a→d = 4.123	b→e = 2.828	c→f = 4.00			
a→e = 2.236	b→f = 2.236				
a→f = 5.099					

	a	b	c	d	e	f
a	0	5	$\sqrt{50}$	$\sqrt{17}$	$\sqrt{5}$	$\sqrt{26}$
b	5	0	$\sqrt{5}$	$\sqrt{10}$	$\sqrt{8}$	$\sqrt{5}$
c	$\sqrt{50}$	$\sqrt{5}$	0	$\sqrt{17}$	5	4
d	$\sqrt{17}$	$\sqrt{10}$	$\sqrt{17}$	0	$\sqrt{10}$	5
e	$\sqrt{5}$	$\sqrt{8}$	5	$\sqrt{10}$	0	3
f	$\sqrt{26}$	$\sqrt{5}$	4	5	3	0