GATE 2026 Computer Science & Information Technology (CS-2) Question Paper with Solutions

ZOLLEGE

General Instructions

(i) This booklet contains 65 questions, each provided with a complete, step-by-step solution.

(ii) It comprises 35 single-correct multiple-choice questions, 11 multiple-correct questions and 19 numerical / integer-type questions.

(iii) These questions follow the GATE Computer Science & Information Technology (CS-2) examination pattern.

(iv) Questions carry 1 or 2 marks. MCQs have negative marking (1/3 or 2/3 of the marks); MSQ and numerical-answer (NAT) questions carry no negative marking.

(v) GATE is a 3-hour paper of 65 questions (100 marks).

(vi) A virtual scientific calculator is provided; physical calculators are not allowed.

(vii) Attempt each question on your own before reviewing the given solution.

(viii) For multiple-correct questions, all correct options must be selected to earn full marks.

(ix) For numerical questions, report the answer rounded exactly as asked.

1. Expedite, Hasten, Hurry,

Fill the blank by choosing a word with a meaning similar to that of the words given above.

(A) Accelerate

(B) Retard

(C) Provide

(D) Disable

Correct Answer: (A) Accelerate

SOLUTION:

The sentence gives three words, expedite, hasten and hurry, and asks for a fourth word that carries the same sense. All three words are about speeding up an action, so the blank needs a word with that same speed related sense. Let's go through each choice.

1. **Accelerate**: this means to make something go faster, which lines up exactly with expedite, hasten and hurry.
2. **Retard**: this means to hold something back or slow it down, the reverse of what we want.
3. **Provide**: this means to give or supply something, it has no link to speed at all.
4. **Disable**: this means to stop something from working, again no link to speed.

Only "accelerate" keeps the same idea of speeding a process up, so it is the word that fits the blank.

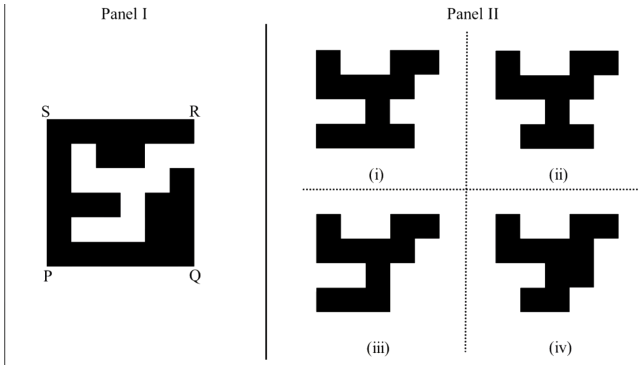
Let's summarize:

- Expedite, hasten and hurry all mean to speed something up.
- Retard means the opposite, while provide and disable are unrelated in meaning.

So the correct word for the blank is accelerate.

Quick Tip: Look for a word that means to make something happen faster, just like expedite, hasten and hurry.

2. A black square PQRS has been cut into two parts. One part of it is shown in Panel I. Which one of the shapes in Panel II is the other part?



(A)

(B)

(C)

(D)

Correct Answer: (C)

SOLUTION:

This question is about fitting two jigsaw pieces back into one square. Panel I is one piece of square PQRS, and we must find the second piece from the four shapes in Panel II that closes the square exactly.

1. **(i)**: this shape has a comb-like top that does not match the step pattern along the cut edge of Panel I, so placing it next to Panel I leaves uneven gaps.
2. **(ii)**: this is close in outline to (i) but with the teeth shifted slightly, and it still does not line up with the notches of Panel I.

3. **(iii)**: this staircase shaped piece has each step sitting flush against a notch in Panel I, so the two pieces together retrace the straight sides of the original square.
4. **(iv)**: this looks similar to (iii) but the steps are shifted by one position, so a small gap or overlap appears where the pieces should meet.

Since only shape (iii) interlocks perfectly with Panel I along every step of the cut, it is the missing part of square PQRS.

Let's summarize:

- The two pieces of a cut square must interlock at every notch with no gap or overlap.
- Shapes (i) and (ii) belong to a different outline family and do not match the cut at all.
- Shape (iv) matches the family but is offset by one step, while shape (iii) lines up exactly.

So the other part of square PQRS is shape (iii).

> **Quick Tip:** Check which shape's steps exactly fill the notches cut out of Panel I so the two pieces together retrace the square's straight sides.

• • •

3. A day can only be cloudy or sunny. The probability of a day being cloudy is 0.5, independent of the condition on other days. What is the probability that in any given four days, there will be three cloudy days and one sunny day?

(A) $\frac{1}{4}$

(B) $\frac{3}{4}$

(C) $\frac{2}{3}$

(D) $\frac{3}{8}$

Correct Answer: (A) $\frac{1}{4}$

SOLUTION:

Step 1: List the possible orders.

We want three cloudy (C) days and one sunny (S) day among four days. The sunny day could fall on day 1, day 2, day 3 or day 4, giving the four orders CCCS, CCSC, CSCC and SCCC.

Step 2: Find the probability of one such order.

Since each day is independent and cloudy or sunny each has probability 0.5, any single fixed order of three C's and one S has probability

$$(0.5) \times (0.5) \times (0.5) \times (0.5) = (0.5)^4 = \frac{1}{16}$$

Step 3: Add up all four orders.

Since these four orders cannot happen at the same time, we add their probabilities:

$$4 \times \frac{1}{16} = \frac{4}{16} = \frac{1}{4}$$

Step 4: Check this against the options.
This value of $\frac{1}{4}$ matches option (A) exactly. The other fractions, $\frac{3}{4}$, $\frac{2}{3}$ and $\frac{3}{8}$, do not come from counting the four valid orders of three cloudy days and one sunny day, so they can be ruled out.

Step 5: Conclude.

$$\boxed{\frac{1}{4}}$$

Quick Tip: Count how many of the four days can be the single sunny day, then multiply by the chance of one fixed order of three cloudy and one sunny day.

• • •

4. The values of Stock A and Stock B on a particular day are Rs. 50 and Rs. 80, respectively. An investor invests Rs. 100 in Stock A and Rs. 80 in Stock B. He sells all the stocks the next day when the value of Stock A is Rs. 55 and Stock B is Rs. 70. The profit made by the investor is Rs.

(A) 0

(B) 5

(C) 10

(D) 20

Correct Answer: (A) 0

SOLUTION:

Step 1: Work out the gain or loss per stock separately.
For Stock A, the investor bought $\frac{100}{50} = 2$ shares at Rs. 50 each and sold

them at Rs. 55 each. The gain per share is $55 - 50 = \text{Rs. } 5$, so the total gain from Stock A is

$$2 \times 5 = \text{Rs. } 10$$

Step 2: Work out Stock B's result.

For Stock B, the investor bought $\frac{80}{80} = 1$ share at Rs. 80 and sold it at Rs. 70. The loss per share is $80 - 70 = \text{Rs. } 10$, so the total loss from Stock B is

$$1 \times 10 = \text{Rs. } 10$$

Step 3: Combine the two results.

The gain of Rs. 10 from Stock A and the loss of Rs. 10 from Stock B cancel out:

$$10 - 10 = 0$$

Step 4: Check the other options.

A profit of Rs. 5, Rs. 10 or Rs. 20 would mean the gain and loss did not fully offset, but the numbers here work out to exactly equal and opposite amounts, so those values do not fit.

Step 5: Conclude.

$$\boxed{\text{Rs. } 0}$$

Quick Tip: Work out how many shares of each stock were bought, then compare the total buying cost with the total selling value.

• • •

5. 'When it is raining, peacocks dance.'
Based only on this sentence, which one of the following options is necessarily true?

(A) Peacocks dance only when it is raining.

(B) When peacocks dance, it is raining.

(C) When peacocks are not dancing, it is not raining.

(D) When it is not raining, peacocks do not dance.

Correct Answer: (C) When peacocks are not dancing, it is not raining.

SOLUTION:

The sentence "when it is raining, peacocks dance" is a plain if-then rule: raining leads to dancing. It does not say dancing only happens because of rain, and it does not say anything directly about what happens when it stops raining. To find what must be true, we test each option against this rule.

1. **Peacocks dance only when it is raining**: this would mean dancing never happens without rain, but the original sentence only tells us rain causes dancing, not that dancing has no other cause, so this is not forced to be true.
2. **When peacocks dance, it is raining**: this again reverses the original rule and claims dancing always signals rain, which the sentence does not guarantee.

3. **When peacocks are not dancing, it is not raining**: if it were raining while peacocks were not dancing, the original rule would be broken, since rain is supposed to lead to dancing. So no dancing means it cannot be raining, and this option must be true.
4. **When it is not raining, peacocks do not dance**: the original sentence says nothing about peacocks on days without rain, they could still dance for some other reason, so this is not guaranteed.

Only the third option follows directly from the original rule without adding any new assumption, so it is the one that is necessarily true.

Let's summarize:

- "Raining implies dancing" only guarantees its contrapositive, "not dancing implies not raining".
- The converse and inverse of a conditional statement are not automatically true.

So the correct option is "When peacocks are not dancing, it is not raining."

Quick Tip: For a statement "if P then Q", only the contrapositive "if not Q then not P" is guaranteed to be true, not the converse or inverse.

• • •

6. Water : P :: Food : Q

Choose the P and Q combination from the options below to form a meaningful analogy.

(A) P = Thirst; Q = Hunger

(B) P = Drink; Q = Hunger

(C) P = Thirst; Q = Satiated

(D) P = Wet; Q = Critic

Correct Answer: (A) P = Thirst; Q = Hunger

SOLUTION:

The question gives an analogy Water : P :: Food : Q and asks which pair of words keeps the same link on both sides. Let's test each option by putting the words into a full sentence.

1. **P = Thirst; Q = Hunger**: "Not having water leads to thirst" and "not having food leads to hunger." Both sentences follow the exact same pattern: missing the substance creates a specific bodily craving. This fits.
2. **P = Drink; Q = Hunger**: "Water is a drink" describes what water is, while "not having food leads to hunger" describes a craving. The two sentences use different kinds of relations, so this pair breaks the pattern.
3. **P = Thirst; Q = Satiated**: "Not having water leads to thirst" fits, but "not having food leads to satiated" makes no sense, since satiated means full, not craving. This pair fails.
4. **P = Wet; Q = Critic**: "Water makes things wet" is a property relation, and "food needs a critic" is not a bodily craving at all. Neither part matches the required pattern, and the two parts do not even match each other.

Only the first pair keeps the same lack of the substance produces this craving relation running through both halves of the analogy, so P = Thirst and Q = Hunger is correct.

Let's summarize:

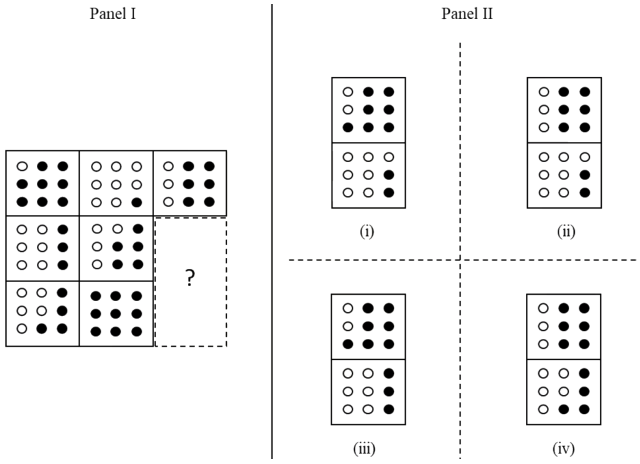
- Water is to thirst as food is to hunger, since both thirst and hunger are the cravings caused by not having the substance.
- The wrong options mix up categories (drink), states (satiated), or unrelated ideas (wet, critic) that do not follow the same relation.

So the correct combination is P = Thirst and Q = Hunger.

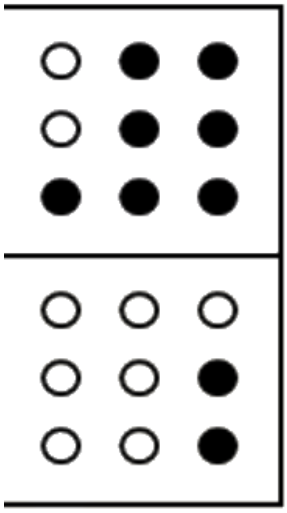
Quick Tip: Think about what feeling arises in the body when water or food is missing; that feeling is the answer for both P and Q.

• • •

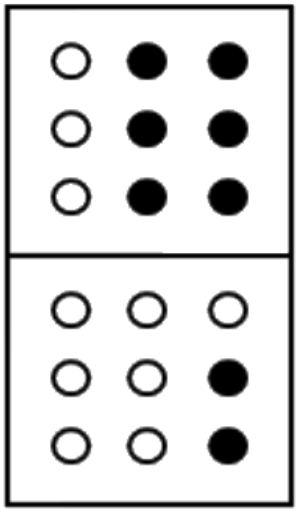
7. Two tiles are missing in Panel I. Which one of the options in Panel II is the appropriate choice for the missing tiles?



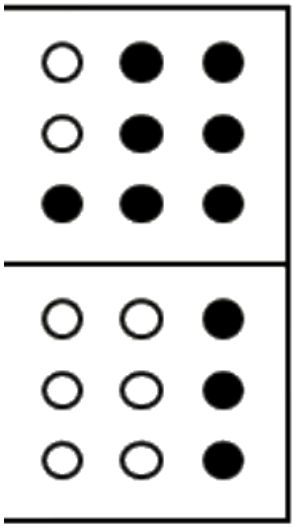
(A)



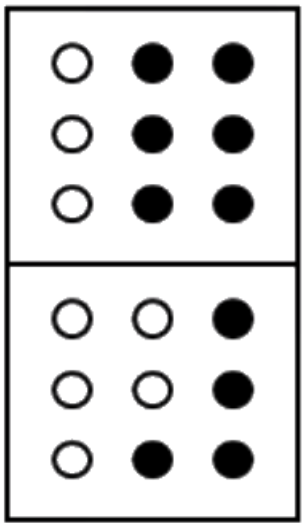
(B)



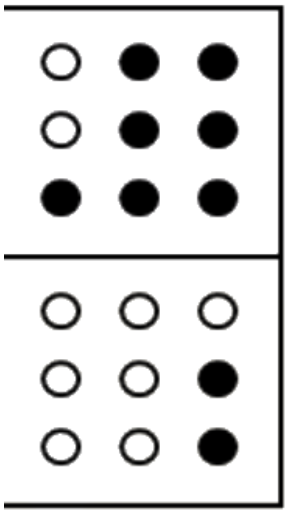
(C)



(D)



Correct Answer: (A)



SOLUTION:

Step 1: Turn each tile into a single number.

For every 3 by 3 tile, just count how many of its nine dots are filled in black. This turns Panel I into a 3 by 3 table of numbers instead of pictures.

$$\begin{matrix} 8 & 1 & 6 \\ 3 & 5 & x \\ 4 & 9 & y \end{matrix}$$

Here x is the top missing tile and y is the bottom missing tile.

Step 2: Use two known columns to find the target total.
The first column reads $8, 3, 4$ and adds to 15. The second column reads $1, 5, 9$ and also adds to 15. Since two different columns both land on the same total, the whole table must be built so that every line, whether a row or a column, adds up to 15.

Step 3: Solve for the missing tile counts using the third column and the bottom row.
The third column gives $6 + x + y = 15$, so $x + y = 9$. The bottom row gives $4 + 9 + y = 15$, so $y = 2$. Putting $y = 2$ back into $x + y = 9$ gives $x = 7$.

Step 4: Cross check with the anti diagonal.
The anti diagonal running through $6, 5, 4$ adds to 15 as well, which backs up the rule used above and shows the whole table behaves like a balanced 3 by 3 number square with every line equal to 15.

Step 5: Compare with the four choices.
We need a top tile with 7 filled dots sitting above a bottom tile with 2 filled dots. Checking the images, option (i) is the only pair with exactly 7 dots on top and 2 dots below it. The other three options give either the wrong top count or the wrong bottom count.

(i)

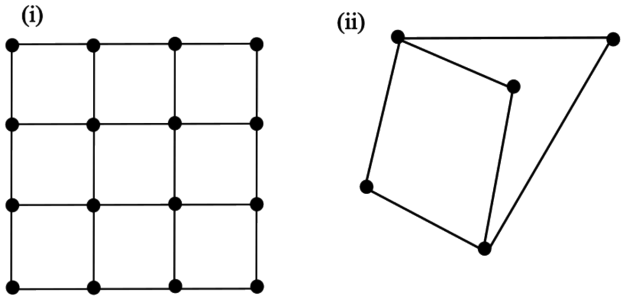
Quick Tip: Count the filled dots in each tile and see that every row and column of tiles adds to the same total.

• • •

8. Figures (i) and (ii) represent intercity highway systems. The black dots represent cities and the line segments between them represent intercity highways.

A salesperson needs to make a trip. She needs to start from a city, visit each of the remaining cities exactly once, and finally return to the same city from which she started.

Which one of the following options is then true?



(A) Such a trip is possible for (i), but not for (ii).

(B) Such a trip is possible for (ii), but not for (i).

(C) Such a trip is possible for both (i) and (ii).

(D) Such a trip is possible neither for (i) nor for (ii).

Correct Answer: (A) Such a trip is possible for (i), but not for (ii).

SOLUTION:

Step 1: Translate the trip into graph language.

A trip that starts at a city, visits every other city exactly once, and returns to the start is what graph theory calls a Hamiltonian cycle: a closed loop using the highways that touches every city exactly once.

Step 2: Use a known fact about grid networks for figure (i).

Figure (i) is a rectangular grid network with 4 rows and 4 columns of cities, each joined only to its up, down, left, and right neighbour. For this kind of grid, a round trip covering every city exists whenever the total number of cities is even, because the cities split evenly into two alternating groups, like the black and white squares of a chessboard, and a closed loop can always snake through the grid, going down one lane and up the next, ending exactly where it began. Since $4 \times 4 = 16$ is even, this round trip exists for figure (i).

Step 3: Count highways at each city in figure (ii).

In figure (ii), two of the five cities, call them V_1 and V_5, each have three highways touching them. The other three cities, V_2, V_3, V_4, each have exactly two highways, and in every case one of those two highways goes to V_1 and the other goes to V_5.

Step 4: See how many highways a round trip actually needs.

A round trip through all five cities uses exactly five highways, one connecting each consecutive pair of cities in the loop, and every city touches exactly two of those used highways. There are only six highways in the whole network, so a round trip would use all but exactly one of them.

Step 5: Show that leaving out one highway cannot fix both busy cities.

Removing a single highway can lower the highway count at, at most, two cities. But there is no highway directly joining V_1 and V_5, so any single highway we leave out touches only one of V_1 or V_5. That means the other one of V_1, V_5 is stuck with three highways in the loop, which is not allowed since a round trip needs exactly two at every city. So no round trip can use all five cities in figure (ii).

Final Answer:

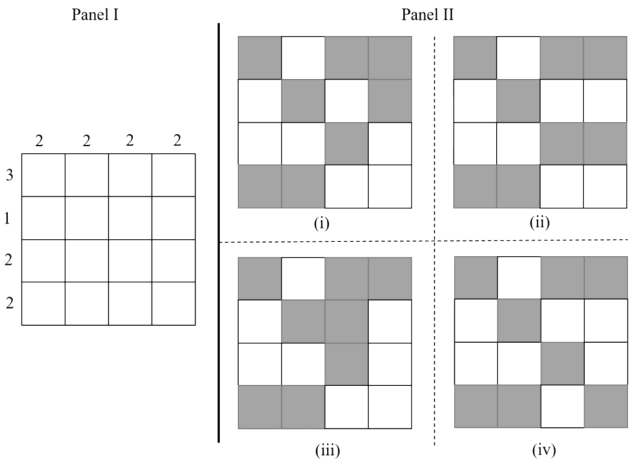
The round trip works for figure (i) but fails for figure (ii).

Possible for (i), not for (ii)

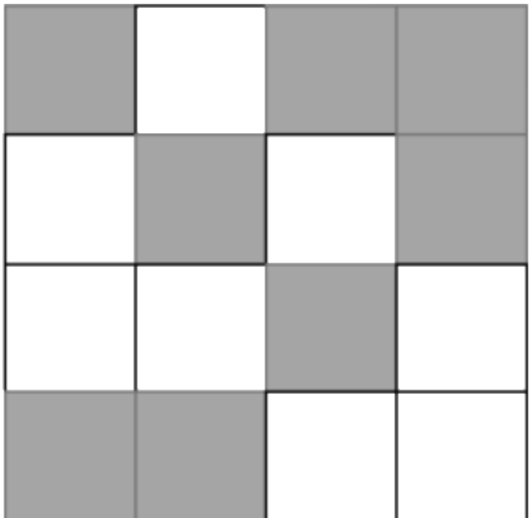
Quick Tip: A full round trip needs exactly two highways used at every city; check whether that is possible at every city in each figure.

• • •

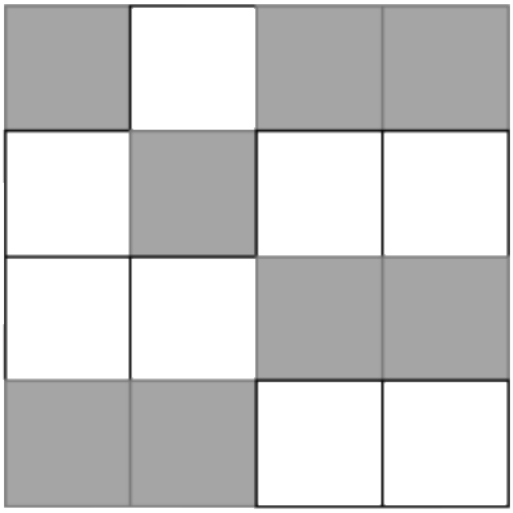
9. The figure in Panel I below is a grid of cells with four rows and four columns. The numbers on the top and on the left represent the number of cells that are to be shaded in that column and row, respectively. Which one of the options shown in Panel II below represents the grid shaded correctly?



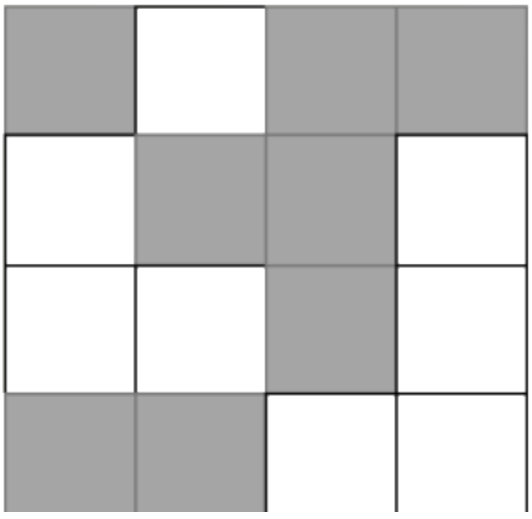
(A)



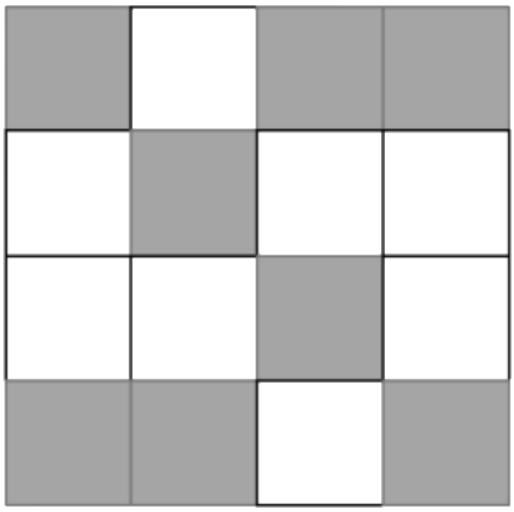
(B)



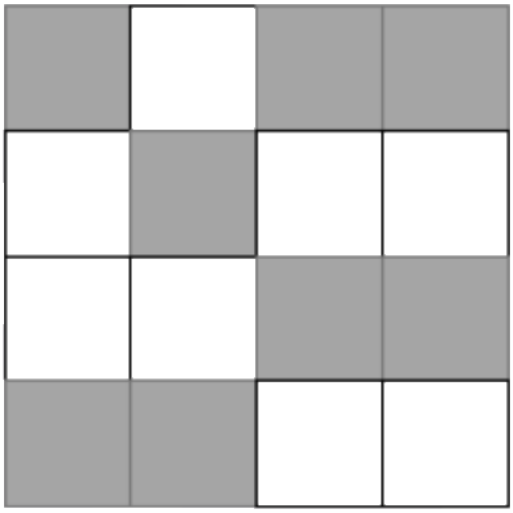
(C)



(D)



Correct Answer: (B)



SOLUTION:

Step 1: Pick the most restrictive clue first.
Look at the row clues 3, 1, 2, 2. The number 1 for row 2 is the strictest clue, since only one cell in the whole second row is allowed to be shaded. Checking row 2 first will eliminate wrong options fastest.

Step 2: Scan row 2 of every option.
In option (i), row 2 has cells shaded in the second and fourth spot, that is 2 shaded cells, already too many. Option (iii) also shades 2 cells in row 2. So (i) and (iii) are out right away.

Option (iv) shades only 1 cell in row 2, and option (ii) also shades only 1 cell in row 2, so both survive this first check.

Step 3: Use the row 3 clue, which should be 2, to separate (ii) and (iv).
In option (ii), row 3 has 2 shaded cells, matching the clue. In option (iv), row 3 has only 1 shaded cell, which breaks the row 3 clue of 2. So option (iv) is eliminated here.

Step 4: Confirm option (ii) fully.
With (i), (iii), and (iv) eliminated, only option (ii) remains. Checking all four rows of (ii) gives 3, 1, 2, 2 and all four columns give 2, 2, 2, 2, both matching Panel I completely.

Final Answer:

$\boxed{\text{(ii)}}$

Quick Tip: Start with the row that allows only one shaded cell; it eliminates most of the wrong options immediately.

• • •

10. An unbiased six-faced dice whose faces are marked with numbers 1, 2, 3, 4, 5, and 6 is rolled twice in succession and the number on the top face is recorded each time. The probability that the sum of the two recorded numbers is a prime number is _____

(A) $\frac{3}{36}$

(B) $\frac{13}{36}$

(C) $\frac{15}{36}$

(D) $\frac{19}{36}$

Correct Answer: (C) $\frac{15}{36}$

SOLUTION:

Step 1: Count outcomes, this time organised by the first roll.
There are $6 \times 6 = 36$ equally likely pairs (a, b) for the two rolls. Instead of grouping by the sum, go through each value of the first roll a and count how many values of b make $a + b$ prime.

Step 2: First roll is 1.
$a = 1$: b must make $1 + b$ prime. Checking $b = 1, \ldots, 6$ gives sums $2, 3, 4, 5, 6, 7$. The primes among these are $2, 3, 5, 7$, from $b = 1, 2, 4, 6$. That is 4 values.

Step 3: First roll is 2.
$a = 2$: sums are $3, 4, 5, 6, 7, 8$. Primes are $3, 5, 7$, from $b = 1, 3, 5$. That is 3 values.

Step 4: First roll is 3.
$a = 3$: sums are $4, 5, 6, 7, 8, 9$. Primes are $5, 7$, from $b = 2, 4$. That is 2 values.

Step 5: First roll is 4.
$a = 4$: sums are $5, 6, 7, 8, 9, 10$. Primes are $5, 7$, from $b = 1, 3$. That is 2 values.

Step 6: First roll is 5.

$a = 5$: sums are $6, 7, 8, 9, 10, 11$. Primes are $7, 11$, from $b = 2, 6$. That is 2 values.

Step 7: First roll is 6.

$a = 6$: sums are $7, 8, 9, 10, 11, 12$. Primes are $7, 11$, from $b = 1, 5$. That is 2 values.

Step 8: Add up all six counts.

$$4 + 3 + 2 + 2 + 2 + 2 = 15$$

This matches the count found by grouping by sum, confirming 15 favourable outcomes out of 36.

$$\boxed{\frac{15}{36}}$$

Quick Tip: List the possible sums from 2 to 12, keep only the prime ones, and count the ordered pairs that give each of those sums.

• • •

11. For two different persons x and y, the predicate $M(x, y)$ denotes that x knows y.

Consider the following statement.

There is a person who does not know anyone else, but that person is known by everyone else.

Which one of the following expressions represents the above statement?

(A) $(\exists y)(\forall x)((x \neq y) \rightarrow (M(x,y) \wedge \neg M(y,x)))$

(B) $(\forall y)(\exists x)((x \neq y) \rightarrow (M(x,y) \wedge \neg M(y,x)))$

(C) $(\exists y)(\exists x)((x \neq y) \rightarrow (M(x,y) \wedge \neg M(y,x)))$

(D) $(\forall y)(\forall x)((x \neq y) \rightarrow (M(x,y) \wedge \neg M(y,x)))$

Correct Answer: (A) $(\exists y)(\forall x)((x \neq y) \rightarrow (M(x,y) \wedge \neg M(y,x)))$

SOLUTION:

Step 1: A good way to check a first-order translation without trusting intuition alone is to test it against a small concrete example universe and see if the formula behaves the way the English sentence demands.

Step 2: Imagine three people, y, a, b, where y knows nobody else ($\neg M(y,a)$ and $\neg M(y,b)$) and both a and b know y ($M(a,y)$ and $M(b,y)$). This is exactly the scenario the English sentence describes, so a correct formula must come out true here, and it should not require this to hold for every person in the universe, only for the one special person y.

Step 3: Check option (D), $(\forall y)(\forall x)$. This demands the pattern $M(x,y) \wedge \neg M(y,x)$ hold for EVERY pair, so it would also require a to be known by everyone and know nobody, which is false in our example since a knows y. So a universal outer quantifier is too strong and option (D) fails on this test case.

Step 4: Check option (C), $(\exists y)(\exists x)$. This only needs one witness pair, so it would already be satisfied by the pair (y,a) alone, even if a third person c both knew y and was known by y, which would break "known by everyone else" for y. So an inner existential quantifier is too weak, it does not force the pattern across ALL other people, and option (C) fails.

Step 5: Check option (B), $(\forall y)(\exists x)$. This says every single person has some other person who knows them and whom they do not know, a claim about the entire population's social structure rather than about one special isolated-yet-famous person. This is a different and much broader statement than the one given, so option (B) does not match either.

Step 6: Only option (A), $(\exists y)(\forall x)\left((x \neq y) \rightarrow (M(x, y) \wedge \neg M(y, x))\right)$, survives: it picks out one specific y (existential outer) and forces the knows/does-not-know pattern to hold against every other person (universal inner), matching both halves of the English sentence exactly.

$$\boxed{(\exists y)(\forall x)\left((x \neq y) \rightarrow (M(x, y) \wedge \neg M(y, x))\right)}$$

Quick Tip: The sentence describes exactly one special person, so use one existential quantifier for that person and one universal quantifier ranging over everyone else.

• • •

12. The set T represents various traversals over a binary tree. The set S represents the order of visiting nodes during a traversal.

T	**S**
I: Inorder	L: left subtree, node, right subtree
II: Preorder	M: node, left subtree, right subtree
III: Postorder	N: left subtree, right subtree, node

Which one of the following is the correct match from T to S?

(A) I - L, II - M, III - N

(B) I - M, II - L, III - N

(C) I - N, II - M, III - L

(D) I - L, II - N, III - M

Correct Answer: (A) I - L, II - M, III - N

SOLUTION:

Step 1: Instead of quoting the definitions from memory, build one small binary tree and physically trace each traversal on it, then match the resulting visiting order to the patterns in S.

Step 2: Take a tree with root r, a left child p, and a right child q. Trace Inorder on this tree: visit the left subtree of r first, which is just p, then visit r itself, then visit the right subtree, which is just q. The order is p, r, q, that is left, node, right, which is pattern L. So I maps to L.

Step 3: Trace Preorder on the same tree: the rule visits the node before its subtrees, so r is visited first, then its left subtree p, then its right subtree q. The order is r, p, q, that is node, left, right, which is pattern M. So II maps to M.

Step 4: Trace Postorder on the same tree: the rule visits both subtrees before the node, so p (left) is visited, then q (right), then r last. The order is p, q, r, that is left, right, node, which is pattern N. So III maps to N.

Step 5: This trace directly gives I - L, II - M, III - N, matching option (A) exactly, with no assignment left ambiguous since each traversal produced a distinct, unmistakable visiting order on the same test tree.

Step 6: Any option that reassigns these, such as putting the node first for Inorder (option B) or claiming Postorder visits the node before its

right child (option D), contradicts what actually happened when the traversal was run on the tree in steps 2 through 4.

I - L, II - M, III - N (option A)

Quick Tip: Inorder visits the node between its subtrees, Preorder visits the node first, and Postorder visits the node last.

• • •

13. Which one of the following statements is equivalent to the following assertion?
Turing machine M decides the language $L \subseteq \{0,1\}^*$

(A) Turing machine M halts on all input strings in $\{0,1\}^*$

(B) Turing machine M accepts all input strings in L

(C) Turing machine M rejects all input strings in $\{0,1\}^* - L$

(D) Turing machine M accepts all input strings in L and rejects all input strings in $\{0,1\}^* - L$

Correct Answer: (D) Turing machine M accepts all input strings in L and rejects all input strings in $\{0,1\}^* - L$

SOLUTION:

Step 1: Split $\{0,1\}^*$, the set of all possible input strings, into exactly two disjoint pieces: L itself and its complement $\{0,1\}^* - L$. Every single string in the universe falls into exactly one of these two pieces, never both and never neither.

Step 2: "Deciding" L means giving a definite, halting yes or no verdict for every string in the whole universe, accept exactly for strings in L, and reject exactly for strings in the other piece. So a correct

restatement of "decides L" must, between the two pieces, pin down both what happens on L and what happens on its complement.

Step 3: Option (A) talks only about halting, with no mention of which strings are accepted versus rejected. A machine could halt everywhere but accept a totally unrelated language, say it accepts only even-length strings regardless of L, and (A) would still hold. So (A) misses the correctness half entirely.

Step 4: Option (B) only pins down the behavior on the L piece, must accept, leaving the complement piece completely unconstrained, including the possibility of looping forever there. Option (C) is the mirror image, it only pins down the complement piece, must reject, leaving L itself unconstrained. Each option therefore only covers one of the two pieces from Step 1, not both.

Step 5: Option (D) is the only statement that pins down both pieces at once, accept everything in L, reject everything in the complement. Since these two pieces exhaust $\{0,1\}^*$ with no overlap, this simultaneously guarantees M halts everywhere, every string gets an explicit accept or reject verdict, and guarantees the verdict is correct everywhere, which is exactly the definition of deciding L.

Option D

Quick Tip: Deciding a language needs BOTH correct acceptance on L and correct rejection on its complement; that combination alone forces halting everywhere too.

14. The probability density function $f(x)$ of a random variable X which takes real values is

$$f(x) = \frac{1}{3\sqrt{2\pi}} \exp\left(-\frac{x^2}{18}\right), \quad x \in (-\infty, +\infty)$$

Which one of the following statements is correct about the random variable X?

(A) X is an exponential random variable

(B) X is a normal random variable

(C) X is a Poisson random variable

(D) X is a uniform random variable

Correct Answer: (B) X is a normal random variable

SOLUTION:

Step 1: Rather than matching constants against the general normal formula directly, check the distinguishing fingerprints of each candidate distribution against $f(x) = \frac{1}{3\sqrt{2\pi}} \exp\left(-\frac{x^2}{18}\right)$ one property at a time.

Step 2: Fingerprint of symmetry: replacing x with $-x$ in $f(x)$ leaves it unchanged, since only x^2 appears in the exponent, so $f(x)$ is an even function, symmetric about $x = 0$. An exponential density is not symmetric, it is zero for $x < 0$ and strictly decreasing for $x \geq 0$, so this already rules out the exponential option.

Step 3: Fingerprint of domain type: $f(x)$ is a smooth function defined for every real number, giving a genuine probability density over a continuum of values. A Poisson random variable only assigns probabilities to the discrete values $0, 1, 2, 3, \ldots$ using a formula with

factorials, it has no density function over a continuum at all, so a continuous bell curve like $f(x)$ cannot represent a Poisson variable.

Step 4: Fingerprint of shape: $f(x)$ strictly decreases as $|x|$ grows away from 0, since $\exp(-x^2/18)$ shrinks as x^2 grows, giving a peak at the center and tapering tails on both sides. A uniform density is flat, constant on some interval and exactly zero outside it, it never tapers smoothly like this, so uniform is ruled out too.

Step 5: The one distribution whose density is a smooth, symmetric, real-line-supported, exponentially tapering bell curve $\frac{1}{\sigma\sqrt{2\pi}}\exp(-x^2/2\sigma^2)$ is exactly the normal distribution, here with $\sigma = 3$ since $2\sigma^2 = 18$. Every fingerprint checked, symmetry, continuous real-line support, smooth tapering peak, matches normal and mismatches the other three.

X is a normal random variable

Quick Tip: Compare the given density with the standard normal form 1/(sigma*sqrt(2*pi)) exp(-x^2/(2*sigma^2)); it matches with mean 0 and sigma = 3.

• • •

15. In the context of DBMS, consider the two sets T and S given below.

T	S
I: Logical schema	L: Views
II: Physical schema	M: File organization and indexes
III: External schema	N: Relations

Which one of the following is the correct match from T to S?

(A) I - L, II - M, III - N

(B) I - M, II - L, III - N

(C) I - N, II - M, III - L

(D) I - N, II - L, III - M

Correct Answer: (C) I - N, II - M, III - L

SOLUTION:

Step 1: Instead of matching definitions one by one, order the three schema levels by how close they sit to the physical disk versus how close they sit to an individual user, and match S in the same order.

Step 2: The level closest to the disk is the Physical schema, since it is only concerned with the mechanics of storage, how records are laid out in files and which indexes exist to speed up lookups. Among the S options, "File organization and indexes" (M) is the only one describing storage mechanics, so Physical schema (II) pairs with M.

Step 3: The level closest to an individual user or application is the External schema, since different users can be given different, restricted, or customized views of the same underlying data without touching the real tables. Among the S options, "Views" (L) is the only user-facing, customizable presentation mechanism, so External schema (III) pairs with L.

Step 4: That leaves the middle level, the Logical schema, which describes the database as a whole for the entire user community, independent of both physical storage details and any single user's restricted viewpoint. This is exactly where the actual base tables, the Relations (N), of the database live, so Logical schema (I) pairs with N.

Step 5: This ordering-based reasoning, physical to storage mechanics, external to user views, logical to the base relations in between, gives I

- N, II - M, III - L, which is option (C), the same result direct definition matching confirms.

I - N, II - M, III - L (option C)

Quick Tip: Logical schema holds the relations, physical schema is about file organization and indexes, and external schema is implemented as views.

• • •

16. Which one of the following options is not a property of Boolean Algebra?
Note: + is the OR operation, · is the AND operation, and ′ is the NOT operation.

(A) $a + b = b + a$

(B) $a \cdot a' = 1$

(C) $a + a' = 1$

(D) $a \cdot b = b \cdot a$

Correct Answer: (B) $a \cdot a' = 1$

SOLUTION:

Step 1: Boolean variables only ever take the value 0 or 1, so any claimed "law" can be checked by simply plugging in both possible values of a (and b where needed) and seeing if the equation holds in every case.

Step 2: Test option (B), $a \cdot a' = 1$. If $a = 0$, then $a' = 1$ (NOT of 0 is 1), so $a \cdot a' = 0 \cdot 1 = 0$, not 1. The claimed equation already fails for this single value of a, so option (B) cannot be a valid property, since a true

law of Boolean Algebra must hold for every possible value of its variables, not just some.

Step 3: For comparison, test option (C), $a + a' = 1$, the same way. If $a = 0$, $a' = 1$, so $a + a' = 0 + 1 = 1$, correct. If $a = 1$, $a' = 0$, so $a + a' = 1 + 0 = 1$, correct again. Both cases give 1, so $a + a' = 1$ holds for every value of a, confirming it is a genuine law, and by contrast highlighting that (B) uses the wrong operation for this identity.

Step 4: Test option (A), $a + b = b + a$, and option (D), $a \cdot b = b \cdot a$, across all four combinations of $a, b \in \{0, 1\}$. OR and AND are each symmetric operations, swapping the order of the two inputs never changes the output for either operation in any of the four cases, so both (A) and (D) hold in every case and are valid commutative laws.

Step 5: Since (A), (C), and (D) hold for every possible substitution of their variables, and (B) fails for $a = 0$, option (B), $a \cdot a' = 1$, is the one option that is not an actual property of Boolean Algebra, the correct identity being $a \cdot a' = 0$.

$$\boxed{a \cdot a' = 1 \text{ (option B) is not a valid property}}$$

Quick Tip: The complement law gives a . a' = 0 and a + a' = 1; option B swaps these two identities.

• • •

17. In a C runtime environment, which one of the following is stored in the heap?

(A) A static variable declared inside a function

(B) An array of integers declared inside a function

(C) A dynamically allocated array of integers created using a `malloc()` function call

(D) Return address of a function

Correct Answer: (C) A dynamically allocated array of integers created using a `malloc()` function call

SOLUTION:

Step 1: A reliable way to tell which memory region something lives in is to ask two questions, who is responsible for freeing this memory, and when does it get freed. The answer sorts it into stack, static/data segment, or heap.

Step 2: For option (A), a static variable inside a function, nobody ever explicitly frees it, and it is not freed automatically at function-return time either, it simply exists as long as the whole program runs, then goes away when the program exits. That "lives for the entire program, no manual free" pattern is the static/data segment, not the heap.

Step 3: For option (B), a plain local array, it is freed automatically, and immediately, the moment the function returns. Nobody calls any explicit release function, the compiler-generated code simply pops the function's stack frame. Automatic release tied exactly to function return is the signature of the stack.

Step 4: For option (D), the return address, this is also freed automatically the instant the function returns, as part of popping that same stack frame, so it follows the identical stack pattern as option (B).

Step 5: For option (C), memory from `malloc()`, nothing releases it automatically when the allocating function returns, the programmer must explicitly call `free()` on it later, possibly from a completely different function, or it stays allocated for the rest of the program's run if never freed. This "must be explicitly and manually released, independent of any function returning" pattern is exactly how heap memory behaves, and it is the only option among the four that shows this pattern.

Step 6: Since only the `malloc()`-allocated array requires an explicit, separate `free()` call rather than being cleaned up automatically by scope or program exit, option (C) is the one stored in the heap.

Option C: dynamically allocated array via malloc()

Quick Tip: Only memory obtained through malloc() (or calloc/realloc) lives on the heap and must be freed explicitly; locals, return addresses go on the stack and statics live in the data segment.

• • •

18. Consider the following two statements about interrupt handling mechanisms in a CPU.

S1: In non-vectored interrupt mechanism, it usually takes more time to start the Interrupt Service Routine (ISR) when compared to that in a vectored interrupt mechanism.
S2: In daisy-chain interrupt mechanism, the CPU polls all the input devices individually to determine the source of the interrupt.
Which one of the following options is correct with respect to S1 and S2?

(A) Both S1 and S2 are true

(B) Both S1 and S2 are false

(C) S1 is true and S2 is false

(D) S1 is false and S2 is true

Correct Answer: (C) S1 is true and S2 is false

SOLUTION:

This question checks two separate ideas: how fast a CPU can reach the correct ISR under vectored versus non-vectored interrupts, and how the daisy-chain scheme actually finds which device raised the interrupt.

1. **On S1 (vectored vs non-vectored speed)**: A vectored interrupt hands the CPU the ISR's starting address directly during the interrupt acknowledge cycle, so execution jumps to the ISR almost instantly. A non-vectored interrupt gives no such address; the CPU must first run a short identification routine, often polling each device's status flag, to work out which device interrupted, and only then branch to that device's ISR. This extra identification step is real overhead, so non-vectored handling is slower to start the ISR. S1 is correct.
2. **On S2 (how daisy chaining finds the source)**: Daisy chaining wires every interrupting device in series on one shared request and acknowledge line. When the CPU sends the acknowledge signal, it ripples down the chain in hardware, device by device, until it reaches the first device that is actually asking for service; that device grabs the signal and identifies itself, and the CPU never has to interrogate the devices in software. This is the opposite of polling. S2 is incorrect.

Since S1 is true and S2 is false, the correct choice is the option that states exactly that.

Let's summarize:

- Vectored interrupts are faster to reach the ISR because the device supplies its own vector address; non-vectored interrupts need an extra software identification step.
- Daisy chaining finds the interrupting device through a hardware acknowledge chain, not through CPU-driven polling of each device.

So the answer is option (C): S1 is true and S2 is false.

Quick Tip: Vectored interrupts hand over the ISR address directly (fast); non-vectored needs an extra identification step (slow). Daisy chaining resolves priority through a hardware chain, not CPU polling.

• • •

19. Consider the following three ANSI-C programs, P1, P2, and P3.

P1:

```
#include <stdio.h>
int a=5;
int main(){
int a=7;
return(0);
}
```

P2:

```
#include <stdio.h>
int main(){
int a=5;
```

```
int a=7;
return(0);
}
```

P3:

```
#include <stdio.h>
int main(){
int a=5;
float a=7;
return(0);
}
```

Which one of the following statements is true?

(A) Only P1 will compile without any error

(B) Only P2 will compile without any error

(C) Only P3 will compile without any error

(D) All three programs P1, P2, and P3 will compile without any error

Correct Answer: (A) Only P1 will compile without any error

SOLUTION:

All three programs revolve around one rule: a name can be reused for a different variable if the new declaration sits in a nested inner scope, but the same name cannot be declared twice inside the same scope, no matter what type the second declaration uses.

1. **P1**: declares a global variable a at file scope, then a separate local variable a inside main(). These live in two different scopes, file scope and the block scope of main, so the local a simply shadows the global one. No conflict exists, and the compiler accepts this. P1 compiles fine.

2. **P2**: declares int a=5 and then int a=7, both directly inside main(), so both sit in the exact same block scope. Declaring the same name twice in one scope is not allowed in C; the compiler reports it as a redefinition error. P2 fails to compile.
3. **P3**: declares int a=5 and then float a=7, again both directly inside main(). Changing the type from int to float does not fix anything, because the problem is the duplicate name in the same scope, not the type. This too is a redefinition, with a type conflict on top, so P3 fails to compile.

Since P2 and P3 both break the same-scope redeclaration rule and only P1 uses a legal nested-scope shadow, only P1 compiles without error.

Let's summarize:

- Shadowing, the same name in a different scope such as global versus local, is legal in C.
- Redeclaring the same name twice in the identical scope is always an error in C, whatever the type.

So the correct answer is option (A): only P1 compiles without error.

Quick Tip: Reusing a name in a nested inner scope (shadowing) is fine in C; declaring the same name twice in the very same scope is always an error, whatever the type.

• • •

20. Consider concurrent execution of two transactions T_1 and T_2 in a DBMS, both of which access a data object A. For these two transactions to not conflict on A, which one of the following statements must be true?

(A) Both T_1 and T_2 only read A

(B) T_1 reads A and T_2 writes A

(C) T_1 writes A and T_2 reads A

(D) Both T_1 and T_2 write A

Correct Answer: (A) Both T_1 and T_2 only read A

SOLUTION:

This question is about when two transactions touching the same data item A can be considered non-conflicting. In concurrency control, an operation pair conflicts only when the two operations belong to different transactions, act on the same item, and at least one of them is a write, because only then does the execution order affect the result.

1. **Both T_1 and T_2 only read A**: reading never changes the value stored in A, so however the two reads are interleaved, both transactions see the same value and the database state after both operations is identical either way. There is no dependency on order, so this pair does not conflict.
2. **T_1 reads A and T_2 writes A**: whether T_1's read happens before or after T_2's write changes what value T_1 actually sees, the old value or the new one. Since the outcome depends on order, this is a conflicting read-write pair.
3. **T_1 writes A and T_2 reads A**: this is the same situation with the roles reversed; T_2's read result depends on whether it runs before or after T_1's write, so it also conflicts.
4. **Both T_1 and T_2 write A**: the value left in A at the end depends on which write executes last, so swapping the order of the two writes changes the final state of the database. This is a conflicting write-write pair.

Only the read-read case is free of conflict, because it is the only pairing where neither operation can alter what the other one observes or leaves behind.

Let's summarize:

- A conflict needs at least one write among the two operations; a pure read-read pair can never conflict.
- Read-write, write-read and write-write are all conflicting because their outcome changes with execution order.

So the statement that must be true for T_1 and T_2 to not conflict on A is that both only read A, option (A).

Quick Tip: A conflict needs at least one write among the two operations; two transactions that only read the same item never conflict, but any pairing with a write does.

• • •

21. Consider a file of size 4 million bytes being transferred between two hosts connected via a path consisting of three consecutive links of bandwidth 2 Mbps, 500 kbps, and 1 Mbps, respectively. All processing delays and propagation delays are negligible. Assume that there is no other background traffic over the path and no other additional overhead to transfer the file.

Which one of the following is the total time (in seconds) to transfer the file?

Note: $1M = 10^6$, $1k = 10^3$

(A) 731

(B) 64

(C) 8

(D) 16

Correct Answer: (B) 64

SOLUTION:

The trick in this question is recognizing that a chain of links with different capacities behaves like a single pipe whose overall carrying capacity is set by its narrowest section, the link with the least bandwidth.

1. **Work out the data size in bits**: the file is 4×10^6 bytes. Multiplying by 8 bits per byte gives 32×10^6 bits that must cross the path.
2. **List the three link capacities in a common unit (bps)**: 2×10^6 bps, 0.5×10^6 bps, and 1×10^6 bps for the first, second and third links respectively.
3. **Pick the bottleneck**: among 2×10^6, 0.5×10^6, and 1×10^6, the smallest is 0.5×10^6 bps, the 500 kbps link. Because propagation and processing delays are zero and there is no competing traffic, the sustained end-to-end throughput of the whole three-link path equals this smallest capacity; a faster link before or after it cannot push data through the slow middle link any quicker.
4. **Divide total bits by the bottleneck rate**: $\frac{32 \times 10^6 \text{ bits}}{0.5 \times 10^6 \text{ bps}} = 64$ seconds.

This matches option (B). The other listed values correspond to using the wrong link's rate on its own, 16 s from the 2 Mbps link or roughly

32 s from the 1 Mbps link, rather than the true bottleneck, so they do not represent the actual achievable transfer rate of the path.

Let's summarize:

- A file's transfer time across several links in series, with no meaningful store-and-forward delay, is set by the link with the smallest bandwidth, the bottleneck.
- Total time equals total bits divided by that bottleneck bandwidth, here $32 \times 10^6 / 0.5 \times 10^6 = 64$ s.

So the total time to transfer the file is 64 seconds.

Quick Tip: Treat the three links as one pipe; the overall transfer rate is set by the slowest (bottleneck) link, here 500 kbps.

• • •

22. Which one of the following protocols may need to broadcast some of its messages?

(A) SMTP

(B) FTP

(C) DHCP

(D) HTTP

Correct Answer: (C) DHCP

SOLUTION:

The question is really asking which of these protocols can find itself needing to send a message without already knowing a specific

destination IP address, since broadcasting is the fallback used exactly in that situation.

1. **SMTP**: used between mail servers that already know each other's address, found through a DNS MX lookup beforehand; it always talks point to point, never by broadcast.
2. **FTP**: the client must dial a known server IP to even open the control channel; there is no stage where FTP needs to reach whoever is listening on the network, so it is always unicast.
3. **DHCP**: a host joining a network for the first time typically has no IP address of its own and does not know the DHCP server's address either. Its opening message, DHCPDISCOVER, is sent as a local broadcast, to 255.255.255.255, so any DHCP server on the segment can pick it up and reply; later messages in the handshake are often broadcast too, until the client is fully configured.
4. **HTTP**: the client resolves the web server's IP through DNS before sending any request, so it always talks directly, unicast, to that one server.

Only DHCP has a genuine reason to broadcast: it operates at a point where the requesting host cannot yet address anyone directly. The other three protocols always have a specific, already known destination.

Let's summarize:

- Broadcasting is needed when a host does not yet know who to talk to directly, exactly DHCP's situation when it first joins a network.
- SMTP, FTP, and HTTP all run over connections to an already-known server address, so they stay unicast.

So the protocol that may need to broadcast its messages is DHCP, option (C).

Quick Tip: A protocol broadcasts only when it does not yet know a specific destination address; DHCP's first message goes out before the client has an IP or knows the server's address.

• • •

23. Which one of the following CPU scheduling algorithms cannot be preemptive?

(A) Shortest Remaining Time First (SRTF) Scheduling

(B) First Come First Serve (FCFS) Scheduling

(C) Round Robin Scheduling

(D) Priority Scheduling

Correct Answer: (B) First Come First Serve (FCFS) Scheduling

SOLUTION:

The question is asking which scheduling policy, by its very definition, has no way to interrupt a running process once it has started. That rules out algorithms that only sometimes preempt and points at the one that structurally never can.

1. **SRTF (Shortest Remaining Time First)**: constantly compares the remaining burst time of the running process against any newly arrived process, and switches the CPU over the moment a shorter job appears. This constant re-evaluation makes it preemptive by nature.
2. **FCFS (First Come First Serve)**: processes are served strictly in arrival order, and whichever process is running keeps the CPU until it completes or blocks itself; nothing in the rule ever takes the CPU away early for a later arrival. There is no preemptive

variant of FCFS, since interrupting arrival order would break the "first come first serve" rule itself.

3. **Round Robin**: hands out a fixed time quantum and force-switches to the next process the instant that quantum runs out, regardless of whether the current process is done. This forced time-based switch is a textbook case of preemption.
4. **Priority Scheduling**: commonly implemented with preemption, where a newly arrived higher priority process immediately grabs the CPU from a lower priority one that is running. So it supports a preemptive mode, even though a non-preemptive version also exists.

Three of the four algorithms, SRTF, Round Robin, and Priority, are able to run preemptively. FCFS is the odd one out, since its entire definition is built around never interrupting the currently running process.

Let's summarize:

- Preemption means taking the CPU away from a running process before it finishes on its own.
- FCFS never does this by definition, while SRTF, Round Robin, and Priority scheduling all can.

So the algorithm that cannot be preemptive is FCFS, option (B).

Quick Tip: Preemption means interrupting a running process before it finishes. FCFS always runs a process to completion once started, so it has no preemptive form.

• • •

24. Consider the following functions, where n is a positive integer.

$$n^{1/3}, \quad \log(n), \quad \log(n!), \quad 2^{\log(n)}$$

Which one of the following options lists the functions in increasing order of asymptotic growth rate?
Note: Assume the base of log to be 2.

(A) $\log(n)$, $n^{1/3}$, $2^{\log(n)}$, $\log(n!)$

(B) $n^{1/3}$, $\log(n)$, $\log(n!)$, $2^{\log(n)}$

(C) $\log(n)$, $n^{1/3}$, $\log(n!)$, $2^{\log(n)}$

(D) $2^{\log(n)}$, $n^{1/3}$, $\log(n)$, $\log(n!)$

Correct Answer: (A) $\log(n)$, $n^{1/3}$, $2^{\log(n)}$, $\log(n!)$

SOLUTION:

The fastest way through this question is to rewrite every function in a form that is directly comparable, then rank them using standard growth-rate facts rather than plugging in numbers.

1. $2^{\log(n)}$: since the log is base 2, this simplifies exactly to n, because raising 2 to the power of "the exponent that gives n" just gives back n. So this term behaves like a plain linear function.
2. $\log(n!)$: using $\log(n!) = \log 1 + \log 2 + \cdots + \log n$, and comparing this sum to $n \log n$, since each of the n terms is at most $\log n$, and roughly half of them, from $n/2$ to n, are at least $\log n - 1$, $\log(n!)$ is of the order $n \log n$. This is strictly bigger than plain n for large n, since it has an extra growing factor of $\log n$ multiplied in.
3. $n^{1/3}$ **versus** $\log(n)$: any positive power of n, however small the exponent, eventually outgrows any power of $\log n$. Substituting $n = 2^m$ turns $n^{1/3}$ into $2^{m/3}$, exponential in m, and $\log(n)$ into m, linear in m, and exponential growth always eventually beats linear growth. So $n^{1/3}$ overtakes $\log(n)$ once n is large enough.

Putting the four on one scale: $\log(n)$ is the slowest, logarithmic growth, then $n^{1/3}$, since a fractional power beats any power of log, then $2^{\log(n)} = n$, since a full linear power beats a $1/3$ power, and finally $\log(n!) \sim n\log(n)$ is the fastest since it is linear times an extra logarithmic factor.

Let's summarize:

- $2^{\log(n)}$ collapses to plain n.
- $\log(n!)$ grows like $n\log(n)$, faster than plain n.
- Any positive power of n beats any power of $\log(n)$, so $n^{1/3}$ beats $\log(n)$, and n beats $n^{1/3}$.

So the increasing order is $\log(n) < n^{1/3} < 2^{\log(n)} < \log(n!)$, which is option (A).

$$\boxed{\log(n) < n^{1/3} < 2^{\log(n)} < \log(n!)}$$

Quick Tip: Simplify $2^{\log n}$ to n first, recall $\log(n!)$ grows like $n\log n$, and remember any positive power of n eventually beats any power of $\log n$.

• • •

25. Which of the following can be recurrence relation(s) corresponding to an algorithm with time complexity $\Theta(n)$?

(A) $T(n) = T(n-1) + 1, \quad T(1) = 1$

(B) $T(n) = 2T\left(\frac{n}{2}\right) + 1, \quad T(1) = 1$

(C) $T(n) = 2T\left(\frac{n}{2}\right) + n, \quad T(1) = 1$

(D) $T(n) = T(n-1) + n, \quad T(1) = 1$

Correct Answer: (A) $T(n) = T(n-1) + 1, \quad T(1) = 1$; (B) $T(n) = 2T\left(\frac{n}{2}\right) + 1, \quad T(1) = 1$

SOLUTION:

Step 1: What we are hunting for.
We want recurrences whose closed form grows in direct proportion to n, that is $\Theta(n)$. The quickest way to test each one is to write out the first few terms or draw the recursion tree and see what total work comes out.

Step 2: Option (A), a chain that adds a constant.
$T(n) = T(n-1) + 1$ ticks down by one call at a time, adding 1 unit of work each tick, starting from $T(1) = 1$. There are $n - 1$ ticks before we land on $T(1)$, so the total work is $1 + (n - 1) = n$. That is exactly $\Theta(n)$, so (A) is in.

Step 3: Option (B), a splitting tree with tiny work per node.
$T(n) = 2T(n/2) + 1$ splits into two half sized calls plus 1 unit of work at the current level. Picture the recursion tree: level 0 has 1 node doing 1 unit, level 1 has 2 nodes doing 1 unit each (total 2), level 2 has 4 nodes (total 4), and so on for $\log_2 n$ levels. Summing this geometric series gives about $2n$, which is still $\Theta(n)$. So (B) is in too.

Step 4: Option (C), the same split but with linear work per level.
$T(n) = 2T(n/2) + n$ looks similar to (B), but now each level does n units of total work (it does not shrink), and there are $\log_2 n$ levels. Total work is $n \log_2 n$, which grows faster than plain n. So (C) is out. This is the familiar merge sort time.

Step 5: Option (D), a chain that adds a growing amount.

$T(n) = T(n-1) + n$ adds n at the top, then $n-1$, then $n-2$, down to 1. That sum is $\frac{n(n+1)}{2}$, which is $\Theta(n^2)$, not linear. So (D) is out.

Step 6: Collect the answers.
Only the chain with constant added work (A) and the halving tree with constant added work (B) grow as $\Theta(n)$.

$$\boxed{\text{(A) and (B)}}$$

Quick Tip: Use the Master Theorem or expand each recurrence directly, then compare the resulting closed form against n, n log n, and n squared growth rates.

• • •

26. Let R be a binary relation on the set $\{1, 2, \ldots, 10\}$, where $(x, y) \in R$ if the product of x and y is square of an integer. Which of the following properties is/are satisfied by R?

(A) Reflexive

(B) Symmetric

(C) Transitive

(D) Antisymmetric

Correct Answer: (A) Reflexive ; (B) Symmetric ; (C) Transitive

SOLUTION:

Step 1: Restate the rule with a useful trick.
Every positive integer n can be written as $n = s \cdot k^2$, where s is the squarefree part of n (what is left after pulling out every square factor).

Two numbers multiply to a perfect square exactly when they share the same squarefree part. We use this trick instead of a full prime by prime proof.

Step 2: Reflexive check.
Take any x in the set. Then $x \cdot x = x^2$, which is obviously a perfect square. So every element is related to itself, and R is reflexive.

Step 3: Symmetric check.
If xy is a perfect square, x and y have the same squarefree part. Since "having the same squarefree part" does not care about order, y and x also share that squarefree part, so yx is a perfect square too. This makes R symmetric.

Step 4: Transitive check.
Say x and y share squarefree part s_1, and y and z share squarefree part s_1 again, since y only has one squarefree part. Then x and z both share that same squarefree part s_1, which means xz is a perfect square. So R passes the transitive test as well.

Step 5: Antisymmetric check with a concrete example.
Try $x = 2$ and $y = 8$. Both have squarefree part 2, since $2 = 2 \cdot 1^2$ and $8 = 2 \cdot 2^2$, so $xy = 16$, a perfect square, and $(2, 8) \in R$. Also $(8, 2) \in R$ by symmetry. But $2 \neq 8$, so antisymmetry fails.

Step 6: Collect the results.
R is reflexive, symmetric and transitive, but it is not antisymmetric.

Reflexive, Symmetric, Transitive

Quick Tip: Use prime factorisation or the squarefree-part idea: two numbers multiply to a perfect square exactly when they share the same squarefree part.

• • •

27. For a real number a, let $I(a) = \int_{-1}^{1} (3x^2 - ax + 1)\,dx$. Which of the following statements is/are true?

(A) The value of $I(a)$ is independent of the value of a

(B) The value of $I(a)$ can vary with the value of a

(C) There exists $a \in (-\infty, +\infty)$ such that $I(a)$ is a positive real number

(D) There exists $a \in (-\infty, +\infty)$ such that $I(a)$ is a negative real number

Correct Answer: (A) The value of $I(a)$ is independent of the value of a ; (C) There exists $a \in (-\infty, +\infty)$ such that $I(a)$ is a positive real number

SOLUTION:

Step 1: Find the antiderivative directly.
Instead of splitting the integral into pieces first, integrate the whole polynomial $3x^2 - ax + 1$ term by term to get the antiderivative

$$F(x) = x^3 - \frac{ax^2}{2} + x$$

Step 2: Plug in the upper limit $x = 1$.

$$F(1) = 1 - \frac{a}{2} + 1 = 2 - \frac{a}{2}$$

Step 3: Plug in the lower limit $x = -1$.

$$F(-1) = -1 - \frac{a}{2} - 1 = -2 - \frac{a}{2}$$

Step 4: Subtract to get $I(a)$.

$$I(a) = F(1) - F(-1) = \left(2 - \frac{a}{2}\right) - \left(-2 - \frac{a}{2}\right)$$

$$I(a) = 2 - \frac{a}{2} + 2 + \frac{a}{2}$$

The $-\frac{a}{2}$ and $+\frac{a}{2}$ cancel out completely, leaving

$$I(a) = 4$$

Step 5: Read off the truth of each claim.

Since the a term vanished on its own during the subtraction, the value 4 never depends on what a we pick. So claim (A) holds and claim (B) fails. Also 4 is a fixed positive number, so we can certainly find an a, in fact any a, that makes $I(a)$ positive, which makes claim (C) true. But $I(a)$ is never negative, so claim (D) fails.

Step 6: State the answer.

$$\boxed{I(a) = 4 \text{ for all } a}$$

The true statements are (A) and (C).

Quick Tip: The term with a is an odd power of x over a symmetric interval, so it always integrates to zero; only the even-power terms survive.

• • •

28. In a system, numbers are represented using 4-bit two's complement form. Consider four numbers $N1 = 1011$, $N2 = 1101$, $N3 = 1010$ and $N4 = 1001$ in the system. Which of the following operations will result in arithmetic overflow?

(A) $N1 + N2$

(B) $N2 + N3$

(C) $N3 - N4$

(D) $N1 + N4$

Correct Answer: (B) $N2 + N3$; (D) $N1 + N4$

SOLUTION:

Step 1: Convert every number to plain decimal first.

For 4-bit two's complement, the leftmost bit is worth -8 and the rest add normally. So

$N1 = 1011 = -8 + 2 + 1 = -5$

$N2 = 1101 = -8 + 4 + 1 = -3$

$N3 = 1010 = -8 + 2 = -6$

$N4 = 1001 = -8 + 1 = -7$

Step 2: Know the safe zone.

A 4-bit two's complement register can only hold values from -8 up to 7. If the true arithmetic answer steps outside this window, the hardware cannot show it correctly, and that mismatch is exactly what we call overflow. So the fast check is: work out the real sum or

difference, and see if it fits between -8 and 7.

Step 3: Test $N1 + N2$.
$-5 + (-3) = -8$. This sits right at the edge of the window, so it still fits. No overflow.

Step 4: Test $N2 + N3$.
$-3 + (-6) = -9$. This is one step past -8, outside the window. Overflow happens here.

Step 5: Test $N3 - N4$.
$-6 - (-7) = -6 + 7 = 1$. Comfortably inside the window. No overflow.

Step 6: Test $N1 + N4$.
$-5 + (-7) = -12$. This is well past -8, outside the window. Overflow happens here too.

Step 7: Double check with the sign rule.
As a cross check, overflow in addition happens when we add two negative numbers and the 4-bit result comes out looking positive. In both flagged cases, $N2 + N3$ and $N1 + N4$, we add two negative numbers whose true sum needs more than 4 bits to store correctly as negative, so the stored pattern flips sign by mistake. That matches the decimal check.

Step 8: Conclude.

$$\boxed{N2 + N3 \text{ and } N1 + N4}$$

Quick Tip: The 4-bit two's complement range is -8 to 7; find the true decimal sum or difference of each pair and check whether it falls outside this range.

• • •

29. Which of the following grammars is/are ambiguous?

(A) $S \rightarrow aSb \mid \epsilon$

(B) $E \rightarrow E + E \mid E * E \mid id$

(C) $S \rightarrow aS \mid Sa \mid \epsilon$

(D) $S \rightarrow aS \mid \epsilon$

Correct Answer: (B) $E \rightarrow E + E \mid E * E \mid id$; (C) $S \rightarrow aS \mid Sa \mid \epsilon$

SOLUTION:

Step 1: The test we will run.
For each grammar, pick one string in its language and try to build it in two genuinely different ways. If we succeed, the grammar is ambiguous. If every string can only be built one way, it is not.

Step 2: Grammar (A), $S \rightarrow aSb \mid \epsilon$.
This makes matched strings like $aabb$. Think of it as nested brackets: the outermost a must pair with the outermost b, then the next a with the next b, and so on inward. There is no freedom in how the nesting can be arranged, so each string has just one shape. Not ambiguous.

Step 3: Grammar (B), $E \rightarrow E + E \mid E * E \mid id$.
Take the string $id * id + id$. We can either treat the $*$ as happening first at the top, giving the shape $(id * id) + id$, or treat the $+$ as happening first at the top, giving the shape $id * (id + id)$. Since the grammar has

no rule saying $*$ binds tighter than $+$, both top level splits are legal, so we get two different trees for the same string. Ambiguous.

Step 4: Grammar (C), $S \to aS \mid Sa \mid \epsilon$.
Take the short string a. We can derive it as $S \to aS \to a\epsilon = a$, or as $S \to Sa \to \epsilon a = a$. These are two distinct derivations, using a different first rule, for the exact same one letter string. So even the simplest nonempty string already has two trees. Ambiguous.

Step 5: Grammar (D), $S \to aS \mid \epsilon$.
Take the string a. The only way to produce it is $S \to aS \to a\epsilon = a$. There is no Sa rule to mix in, so there is exactly one route for every string of a's. Not ambiguous.

Step 6: Conclude.
The grammars that allow two different builds for the same string are (B) and (C).

(B) and (C)

Quick Tip: Try to find one string in the language that has two different parse trees or two different leftmost derivations; if you can, the grammar is ambiguous.

• • •

30. The keys $5, 28, 19, 15, 26, 33, 12, 17, 10$ are inserted into a hash table using the hash function $h(k) = k \bmod 9$. The collisions are resolved by chaining. After all the keys are inserted, the length of the longest chain is _____. (answer in integer)

Correct Answer: 3

SOLUTION:

Step 1: List the slots available.

Since the hash function is $h(k) = k \bmod 9$, there are exactly 9 slots, numbered 0 through 8. We insert the keys one at a time, in the order given, and drop each one into its slot's chain.

Step 2: Insert the keys in order and track the table.

Insert 5: $5 \bmod 9 = 5$. Slot 5 now holds [5].

Insert 28: $28 \bmod 9 = 1$. Slot 1 now holds [28].

Insert 19: $19 \bmod 9 = 1$. Slot 1 now holds [28, 19].

Insert 15: $15 \bmod 9 = 6$. Slot 6 now holds [15].

Insert 26: $26 \bmod 9 = 8$. Slot 8 now holds [26].

Insert 33: $33 \bmod 9 = 6$. Slot 6 now holds [15, 33].

Insert 12: $12 \bmod 9 = 3$. Slot 3 now holds [12].

Insert 17: $17 \bmod 9 = 8$. Slot 8 now holds [26, 17].

Insert 10: $10 \bmod 9 = 1$. Slot 1 now holds [28, 19, 10].

Step 3: Read off the final table.

Slot 1: [28, 19, 10], length 3.

Slot 3: [12], length 1.

Slot 5: [5], length 1.

Slot 6: [15, 33], length 2.

Slot 8: [26, 17], length 2.

All other slots, 0, 2, 4, 7, stay empty.

Step 4: Spot the longest chain.

Slot 1 has grown the longest, holding three keys, more than any other slot.

3

Quick Tip: Compute k mod 9 for every key and group the keys that land in the same slot; the longest chain is the largest such group.

• • •

31. Consider the system of linear equations given below.

$$ax + y = b$$

$$16x + ay = 24$$

Suppose the values of a and b are chosen such that the system of linear equations produce multiple solutions. Then the product of a and b is _____. (answer in integer)

Correct Answer: 24

SOLUTION:

Step 1: Use the determinant view of infinitely many solutions.
Write the system as

$$ax + y - b = 0$$

$$16x + ay - 24 = 0$$

Two lines coincide, giving infinitely many common points, exactly when the coefficient rows are proportional, which for this pair means

$a \cdot a - 1 \cdot 16 = 0$.

Step 2: Solve the determinant equation.

$$a^2 - 16 = 0$$

$$a^2 = 16$$

$$a = 4 \text{ or } a = -4$$

This matches the requirement that the two rows of coefficients be proportional.

Step 3: Pin down b using the same scale factor, case $a = 4$.
When $a = 4$, the first equation is $4x + y = b$. For this to be a scaled copy of $16x + 4y = 24$, notice the second equation is exactly 4 times $(4x + y = 6)$, since $4 \times 4x = 16x$, $4 \times y = 4y$, and $4 \times 6 = 24$. So the first equation must read $4x + y = 6$, meaning $b = 6$.

Step 4: Pin down b for case $a = -4$.
When $a = -4$, the first equation is $-4x + y = b$. The second equation $16x - 4y = 24$ is -4 times $(-4x + y = -6)$, since $-4 \times (-4x) = 16x$, $-4 \times y = -4y$, and $-4 \times (-6) = 24$. So $b = -6$ here.

Step 5: Multiply a and b in both cases.
For $a = 4$, $b = 6$: $ab = 24$.
For $a = -4$, $b = -6$: $ab = 24$.
Both scenarios land on the exact same product, so the answer does not

depend on which sign we pick.

Step 6: State the final value.

$$\boxed{ab = 24}$$

Quick Tip: Two linear equations have infinitely many solutions when they are scalar multiples of each other; equate the ratios of x-coefficients, y-coefficients, and constants.

• • •

32. Consider an array $A = [10, 7, 8, 19, 41, 35, 25, 31]$. Suppose the merge sort algorithm is executed on array A to sort it in increasing order. The merge sort algorithm will carry out a total of 7 merge operations.

A merge operation on sorted left array L and sorted right array R is said to be void if the output of the merge operation is the elements of array L followed by the elements of array R.

The number of void merge operations among these 7 merge operations is _____.

Correct Answer: 3

SOLUTION:

Step 1: Turn the definition into a simple test.
A merge of L and R is void exactly when the final sorted list equals L followed by R. That can only happen if every element of L is already less than or equal to every element of R, because then the merge

algorithm never needs to pick an element from R before finishing L. So the quick test is: a merge is void exactly when the largest value in L is no bigger than the smallest value in R.

Step 2: Build the merge tree for the array.
The array is $A = [10, 7, 8, 19, 41, 35, 25, 31]$. Merge sort breaks it down as: $[10, 7, 8, 19]$ and $[41, 35, 25, 31]$, which further break into $[10]$, $[7]$, $[8]$, $[19]$, $[41]$, $[35]$, $[25]$, $[31]$. This gives 7 merges in total: 4 at the leaf level, 2 at the middle level, and 1 at the top.

Step 3: Apply the max-min test to the 4 leaf merges.
$L = [10], R = [7]$: the largest value in L is 10 and the smallest in R is 7. Since 10 is bigger than 7, this merge is not void.
$L = [8], R = [19]$: the largest in L is 8 and the smallest in R is 19. Since 8 is not bigger than 19, this merge is void.
$L = [41], R = [35]$: the largest in L is 41 and the smallest in R is 35. Since 41 is bigger than 35, this merge is not void.
$L = [25], R = [31]$: the largest in L is 25 and the smallest in R is 31. Since 25 is not bigger than 31, this merge is void.

Step 4: Work out the sorted results these leaf merges give.
Sorting $[10], [7]$ gives $[7, 10]$. Sorting $[8], [19]$ gives $[8, 19]$. Sorting $[41], [35]$ gives $[35, 41]$. Sorting $[25], [31]$ gives $[25, 31]$. These four results become the inputs to the middle level.

Step 5: Apply the test to the 2 middle merges.
$L = [7, 10], R = [8, 19]$: the largest in L is 10 and the smallest in R is 8. Since 10 is bigger than 8, this merge is not void.
$L = [35, 41], R = [25, 31]$: the largest in L is 41 and the smallest in R is 25. Since 41 is bigger than 25, this merge is not void.
These two merges produce $[7, 8, 10, 19]$ and $[25, 31, 35, 41]$.

Step 6: Apply the test to the final top merge.
$L = [7, 8, 10, 19], R = [25, 31, 35, 41]$: the largest in L is 19 and the smallest in R is 25. Since 19 is not bigger than 25, this merge is void.

Step 7: Add up the void merges.
Void merges happened at the leaf merge of [8], [19], the leaf merge of [25], [31], and the final top merge. That is 3 void merges out of 7 total.

$$\boxed{3}$$

Quick Tip: A merge is void exactly when every element of the left half is already smaller than every element of the right half, so compare the largest value on the left with the smallest value on the right.

• • •

33. If an IP network uses a subnet mask of 255.255.240.0, the maximum number of IP addresses that can be assigned to network interfaces is _____.

Correct Answer: 4094

SOLUTION:

Step 1: Read the mask as a CIDR prefix.
Instead of converting each octet separately, notice that a mask of 255.255.240.0 is a common CIDR value. The third octet 240 in binary is 11110000, which has 4 leading ones. So the mask has $8 + 8 + 4 = 20$ leading one bits, written as /20.

Step 2: Use the general host-count formula.

For a network with prefix length n out of 32 bits, the number of addresses that can be handed out to interfaces is

$$2^{32-n} - 2$$

The subtraction of 2 accounts for the network id and the broadcast address, which are reserved and never given to a device.

Step 3: Plug in the prefix length.
Here $n = 20$, so

$$2^{32-20} - 2 = 2^{12} - 2$$

Step 4: Work out the power of two.
$2^{12} = 4096$, since $2^{10} = 1024$ and $2^{12} = 1024 \times 4 = 4096$.

Step 5: Double check by counting the addresses directly.
A /20 block sits inside the third octet, since 20 bits covers the first two full octets ($8 + 8 = 16$ bits) plus 4 more bits into the third octet. That leaves 4 bits of the third octet plus the whole fourth octet as host bits, which is $4 + 8 = 12$ host bits, matching Step 3. Each block of this size spans 16 values in the third octet, since $2^4 = 16$, and all 256 values in the fourth octet, giving $16 \times 256 = 4096$ addresses in the block, the same total found above.

Step 6: Subtract the reserved addresses.
Out of these 4096 addresses, one is the all zero host address that names the network itself, and one is the all one host address used for

broadcasting to every device on that network. Both are set aside and are never handed to an actual interface:

$$4096 - 2 = 4094$$

Step 7: Conclude.

$$\boxed{4094}$$

Quick Tip: Count the number of 1 bits in the subnet mask to get the network bits, then use 2 raised to the number of host bits, minus 2 for the network and broadcast addresses.

• • •

34. The 32-bit IEEE 754 single precision representation of a number is $0xC2710000$. The number in decimal representation is _____. (rounded off to two decimal places)

Correct Answer: -60.25

SOLUTION:

Step 1: Expand the hex value bit by bit.
$0xC2710000$ becomes, digit by digit,
1100 0010 0111 0001 0000 0000 0000 0000 in binary, a full 32 bit word.

Step 2: Pull out the three fields.
The leftmost bit is the sign, $S = 1$. The next 8 bits are the exponent field, $E = 10000100_2$. The remaining 23 bits are the mantissa field, $M = 11100010000000000000000$.

Step 3: Convert the exponent field to a number.

$$E = 1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 = 128 + 4 = 132$$

Removing the bias of 127 for single precision gives the real exponent, $132 - 127 = 5$.

Step 4: Treat the mantissa as a fraction over 2^{23}.
Only four bits of M are set, at positions 1, 2, 3 and 7 after the point. As a fraction over 2^{23} this is

$$M_{value} = \frac{2^{22} + 2^{21} + 2^{20} + 2^{16}}{2^{23}}$$

Since $2^{22} + 2^{21} + 2^{20} + 2^{16} = 4194304 + 2097152 + 1048576 + 65536 = 7405568$, and $2^{23} = 8388608$,

$$M_{value} = \frac{7405568}{8388608} = 0.8828125$$

Step 5: Add the implied leading bit.
The significand is $1 + M_{value} = 1.8828125$.

Step 6: Scale by the exponent and apply the sign.

$$(-1)^1 \times 1.8828125 \times 2^5 = -1.8828125 \times 32$$

Step 7: Multiply out.

$1.8828125 \times 32 = 60.25$, so with the negative sign the value is -60.25.

$$\boxed{-60.25}$$

Quick Tip: Split the 32 bits into 1 sign bit, 8 exponent bits, and 23 mantissa bits, then apply the IEEE 754 formula (-1)^S x 1.mantissa x 2^(exponent-127).

• • •

35. A lexical analyzer uses the following token definitions:

- $letter \rightarrow [A - Za - z]$
- $digit \rightarrow [0 - 9]$
- $id \rightarrow letter\,(letter \mid digit)^*$
- $number \rightarrow digit^+$
- $ws \rightarrow (blank \mid tab \mid newline)^+$

For the string given below,

$$x1\ \ 23mm\ \ 78\ \ y\ \ 7z\ \ zz5\ \ 14A\ \ 8H\ \ AaYcD$$

the number of tokens (excluding ws) that will be produced by the lexical analyzer is _____.

Correct Answer: 13

SOLUTION:

Step 1: Note the two competing rules and how a scanner picks between them.

A *number* is one or more digits and nothing else, since its rule is just

digit$^+$. An *id* starts with a letter, and after that first letter it can mix letters and digits freely, since its rule is *letter* (*letter* | *digit*)*. A real scanner reads left to right and always grabs the longest run of characters that still matches a rule (this is maximal munch). So the very first character of a chunk decides which rule takes over: start on a digit and you get a *number* that stops dead the instant a letter appears, start on a letter and the whole rest of the chunk, letters and digits both, gets swallowed into one *id*.

Step 2: Work through the string in its 9 space separated chunks, using this rule.

*x*1: the scanner starts on the letter *x*, so it keeps going through the digit 1 as well, since digits are allowed inside an *id* after the first letter. The whole chunk is one *id*, giving 1 token.

23*mm*: the scanner starts on a digit, so it grabs 2 then 3 as a *number* and stops the moment it hits the letter *m*, since a *number* cannot contain letters. It then starts fresh at *m*, which begins a new *id* covering *mm*. That is a *number* plus an *id*, giving 2 tokens.

78: both characters are digits, so the scanner just keeps extending the *number* to the end of the chunk with nothing left over. One *number* token.

y: a single letter starts and ends an *id* with nothing more to add. One *id* token.

7*z*: the scanner starts on digit 7, builds a *number*, and stops at the letter *z*, exactly like the 23*mm* case. The leftover *z* becomes its own *id*. That is 2 tokens.

*zz*5: the scanner starts on the letter *z*, so once it commits to building an *id* it is allowed to keep pulling in more letters and even the digit 5 at the end. Nothing gets split off, so the whole chunk is a single *id*, 1 token.

14*A*: digits 1 and 4 build a *number* first, stopping at the letter *A*, which

then becomes a separate *id*. That is 2 tokens.
8*H*: the lone digit 8 becomes a ***number***, and the lone letter *H* becomes a separate *id*, since a ***number*** can never absorb a trailing letter. That is 2 tokens.
AaYcD: the scanner starts on the letter *A* and every character after it is also a letter, so the whole five character chunk is swallowed into a single *id*. One token.

Step 3: Lay the token counts out chunk by chunk in the order they appear.

$$1,\ 2,\ 1,\ 1,\ 2,\ 1,\ 2,\ 2,\ 1$$

These correspond in order to *x*1, 23*mm*, 78, *y*, 7*z*, *zz*5, 14*A*, 8*H*, *AaYcD*.

Step 4: Sum the list to get the total token count.

$$1 + 2 + 1 + 1 + 2 + 1 + 2 + 2 + 1 = 13$$

Step 5: Remember the whitespace is not part of this count.
The blanks between the 9 chunks are matched by the separate rule *ws* → (*blank* | *tab* | *newline*)$^{+}$, and the question specifically asks to exclude *ws* tokens from the count, so those gaps contribute nothing to the answer.

Step 6: Conclude.

$$\boxed{13}$$

Quick Tip: Use maximal munch: a number token can only contain digits and stops the instant a letter appears, while an id can mix letters and digits once it starts with a letter.

• • •

36. Consider a complete graph K_n with n vertices ($n > 4$). Note that multiple spanning trees can be constructed over K_n. Each of these spanning trees is represented as a set of edges. The Jaccard coefficient between any two sets is defined as the ratio of the size of the intersection of the two sets to the size of the union of the two sets.

Which one of the following options gives the lowest possible value for the Jaccard coefficient between any two spanning trees of K_n?

(A) $\frac{1}{n}$

(B) $\frac{1}{2n-3}$

(C) 0

(D) $\frac{1}{n-1}$

Correct Answer: (C) 0

SOLUTION:

Step 1: Recall what makes a spanning tree.

A spanning tree of K_n touches every vertex using exactly $n-1$ edges and has no cycle. Two spanning trees can share anywhere from 0 up to $n-2$ edges (they cannot share all $n-1$, since two distinct spanning trees cannot be identical).

Step 2: Express Jaccard in terms of shared edges.

Let k be the number of edges the two trees have in common. The union then holds $2(n-1)-k$ distinct edges, so

$$J = \frac{k}{2(n-1)-k}$$

This is an increasing function of k: a bigger overlap gives a bigger Jaccard value, a smaller overlap gives a smaller one. So the question really asks: what is the smallest possible overlap k between two spanning trees of K_n?

Step 3: Count how many edges K_n actually has.
K_n has $\binom{n}{2} = \frac{n(n-1)}{2}$ edges available to build trees from. Two spanning trees use at most $2(n-1)$ edges between them if they share nothing.

Step 4: Compare the two counts for $n > 4$.

$$\frac{n(n-1)}{2} - 2(n-1) = (n-1)\left(\frac{n}{2} - 2\right) = (n-1)\cdot\frac{n-4}{2}$$

For $n > 4$, both factors $(n-1)$ and $(n-4)$ are positive, so this difference is positive. That means K_n has strictly more edges than two spanning trees would need, so there is no shortage of edges that would force an overlap.

Step 5: Use the known packing result for complete graphs.
A classical result on tree packing says K_n can be split into several completely edge-disjoint spanning trees, in fact up to $\left\lfloor \frac{n}{2} \right\rfloor$ of them. Since only 2 of them need to be edge-disjoint here, and $\lfloor n/2 \rfloor \geq 2$ once $n \geq 4$, two fully disjoint spanning trees are guaranteed to exist.

Step 6: Set $k = 0$ and compute Jaccard.

$$J = \frac{0}{2(n-1) - 0} = 0$$

Step 7: Compare against the other choices.

Options like $\frac{1}{n}$, $\frac{1}{2n-3}$ and $\frac{1}{n-1}$ all assume $k \geq 1$, but we have just shown $k = 0$ is reachable, so none of them can be the true lowest value.

Step 8: Conclude.

$$\boxed{0}$$

Quick Tip: Two spanning trees of K_n can be completely edge disjoint, since the graph has far more edges than two spanning trees need together, so the smallest possible overlap is zero edges.

• • •

37. Let G be a weighted directed acyclic graph with m edges and n vertices. Given G and a source vertex s in G, which one of the following options gives the worst case time complexity of the fastest algorithm to find the lengths of shortest paths from s to all vertices that are reachable from s in G?

(A) $\Theta(m+n)$

(B) $\Theta(m+n\log(n))$

(C) $\Theta(nm)$

(D) $\Theta(n^3)$

Correct Answer: (A) $\Theta(m+n)$

SOLUTION:

Step 1: See why a DAG is a special, easier case.
In a graph with cycles, we generally cannot compute a vertex's shortest distance until we have looked at all its incoming edges, and a cycle can make that circular. A DAG has no cycles, so we can always order the vertices so that every edge points from an earlier vertex to a later one. That ordering is a topological order.

Step 2: Set up a distance recurrence.
Let $dist[v]$ be the shortest distance from s to v. If the vertices are processed in topological order, then when we reach vertex v, every predecessor u with an edge $u \to v$ has already had its final $dist[u]$ computed. So we can write

$$dist[v] = \min_{(u,v)\in E} \Big(dist[u] + w(u,v)\Big)$$

and this value is guaranteed correct the moment we compute it, because no later vertex can ever feed back into v.

Step 3: Count the work for topological sorting.
Getting a topological order, for example with depth first search, touches every vertex once and every edge once, which costs $\Theta(m+n)$.

Step 4: Count the work for filling in the distances.
Going through the vertices in this order and relaxing each outgoing edge exactly once also costs $\Theta(m+n)$ in total, since across the whole run every edge is looked at only once and every vertex is initialized once.

Step 5: Add the two phases together.

$$\Theta(m+n)+\Theta(m+n)=\Theta(m+n)$$

Step 6: Why the other listed complexities are not the fastest.
$\Theta(m+n\log n)$ belongs to Dijkstra's algorithm with a heap, meant for graphs that are not necessarily acyclic. $\Theta(nm)$ belongs to Bellman Ford, built to tolerate negative edges in graphs with cycles. $\Theta(n^3)$ belongs to Floyd Warshall, solving all pairs at once. All three do more work than the DAG structure requires.

Step 7: State the result.

$$\boxed{\Theta(m+n)}$$

This is option (A).

Quick Tip: A DAG allows shortest paths to be found with one topological sort pass plus one relaxation pass over every edge, with no priority queue needed.

38. Consider an array A of integers of size n. The indices of A run from 1 to n. An algorithm is to be designed to check whether A satisfies the condition given below.

$$\forall i, j \in \{1, \ldots, n-1\} \text{ such that } i > j,\ (A[i+1] - A[i]) > (A[j+1] - A[j])$$

Which one of the following gives the worst case time complexity of the fastest algorithm that can be designed for the problem?

(A) $\Theta(n)$

(B) $\Theta(\log(n))$

(C) $\Theta(n \log(n))$

(D) $\Theta(n^2)$

Correct Answer: (A) $\Theta(n)$

SOLUTION:

Step 1: Translate the condition into plain words.
Look at consecutive gaps in the array, $d_k = A[k+1] - A[k]$. The condition asks that a later gap is always bigger than an earlier gap, for every possible pair of positions. In other words, the gaps themselves must form a strictly increasing sequence. An array with this property is sometimes called strictly convex, since its slope keeps increasing.

Step 2: Realize a full pairwise check is wasteful.
Testing d_i against d_j for every pair with i bigger than j takes roughly $n^2/2$ comparisons. But strict inequality is transitive: once we know each gap is bigger than the one right before it, all along the sequence, every non adjacent pair is automatically true too. So testing

neighbours is exactly as strong as testing every pair.

Step 3: Turn this into a single linear scan.
Walk through the array once, keeping a running previous gap value. At each new index compute the current gap and compare it with the previous gap. If it is ever not strictly bigger, the array fails the test right there and the scan can stop early. Otherwise, update the previous gap and move on.

Step 4: Count the operations.
Computing a gap takes constant time, and comparing it to the last gap also takes constant time. Doing this once for each of the $n-1$ gaps costs $\Theta(n)$ overall, with no nested loops and no sorting step.

Step 5: Argue this is already the best possible.
Since the property depends on every element of the array, removing or changing one entry can flip the answer, so any correct checker has to touch every element at least once. That rules out anything faster than linear time, so $\Theta(n)$ is not just achievable, it is optimal.

Step 6: Why the faster and slower options do not fit.
$\Theta(\log n)$ is impossible since the whole array must be read. $\Theta(n \log n)$ suggests some sorting or divide and conquer step, which is not needed once the transitivity shortcut is used. $\Theta(n^2)$ is exactly what the brute force all pairs check would cost, which has already been improved on.

Step 7: Conclude.

$$\boxed{\Theta(n)}$$

This matches option (A).

Quick Tip: Because strict inequality is transitive, checking only consecutive differences is exactly as strong as checking every pair, so a single linear scan is enough.

39. Consider a table T, where the elements $T[i][j]$, $0 \leq i, j \leq n$, represent the cost of the optimal solutions of different subproblems of a problem that is being solved using a dynamic programming algorithm. The recursive formulation to compute the table entries is as follows:

$$T[0][k] = T[k][0] = 1 \quad \text{for } k = 0, 1, 2, \ldots, n$$

$$T[i][j] = 2T[i-1][j] + 3T[i][j-1] \quad \text{for } 1 \leq i, j \leq n$$

Consider the following two algorithms to compute entries of T. Assume that for both the algorithms, for all $0 \leq i, j \leq n$, $T[i][j]$ has been initialized to 1.

Algorithm B_1:

```
For i = 1, 2, ..., n
    For j = 1, 2, ..., n
        T[i][j] = 2*T[i-1][j] + 3*T[i][j-1]
```

Algorithm B_2:

```
For s = 2, 3, ..., 2n
    For i = 1, 2, ..., n
        For j = 1, 2, ..., n
            If (i + j == s)
                T[i][j] = 2*T[i-1][j] + 3*T[i][j-1]
```

Algorithm B_k, $k \in \{1, 2\}$ is said to be correct if and only if it calculates the correct values of $T[i][j]$, for all $0 \leq i, j \leq n$, (as per the recursive

formulation) at the end of the execution of the algorithm B_k.

Which one of the following statements is true?

(A) Both algorithms B_1 and B_2 are correct

(B) Algorithm B_1 is correct, but algorithm B_2 is incorrect

(C) Algorithm B_2 is correct, but algorithm B_1 is incorrect

(D) Both algorithms B_1 and B_2 are incorrect

Correct Answer: (A) Both algorithms B_1 and B_2 are correct

SOLUTION:

A different way to settle this is to trace both algorithms by hand on a small case, $n = 2$, and check every computed value against the recurrence directly, instead of arguing about dependency order in general.

With $n = 2$, the base values are fixed by the recurrence as $T[0][0] = T[0][1] = T[0][2] = 1$ and $T[1][0] = T[2][0] = 1$. We need to find $T[1][1], T[1][2], T[2][1], T[2][2]$ using $T[i][j] = 2T[i-1][j] + 3T[i][j-1]$.

Correct values by direct substitution, computed in the only order that respects the dependencies ($T[1][1]$ first, since it only needs base values; then $T[1][2]$ and $T[2][1]$; then $T[2][2]$ last):

$T[1][1] = 2T[0][1] + 3T[1][0] = 2(1) + 3(1) = 5$

$T[1][2] = 2T[0][2] + 3T[1][1] = 2(1) + 3(5) = 17$

$T[2][1] = 2T[1][1] + 3T[2][0] = 2(5) + 3(1) = 13$

$T[2][2] = 2T[1][2] + 3T[2][1] = 2(17) + 3(13) = 34 + 39 = 73$

Now trace B_1 (i outer =1,2; j inner =1,2) on this table, using entries as

they stand at the moment each is read:
$i = 1, j = 1$: $T[1][1] = 2T[0][1] + 3T[1][0] = 2(1) + 3(1) = 5$. Matches.
$i = 1, j = 2$: $T[1][2] = 2T[0][2] + 3T[1][1] = 2(1) + 3(5) = 17$, using the just-updated $T[1][1] = 5$. Matches.
$i = 2, j = 1$: $T[2][1] = 2T[1][1] + 3T[2][0] = 2(5) + 3(1) = 13$, using row 1's finished value $T[1][1] = 5$. Matches.
$i = 2, j = 2$: $T[2][2] = 2T[1][2] + 3T[2][1] = 2(17) + 3(13) = 73$, using $T[1][2] = 17$ from row 1 and the just-updated $T[2][1] = 13$. Matches.
Every value B_1 produces equals the value computed by direct substitution, so B_1 is correct on this test case, consistent with the general argument.

Now trace B_2 (diagonals $s = 2, 3, 4$; within a fixed s, order of (i, j) does not matter here since $n = 2$ gives at most two cells per diagonal and they do not depend on each other):
$s = 2$: only $(i, j) = (1, 1)$ satisfies $i + j = 2$ with $1 \le i, j \le 2$. $T[1][1] = 2T[0][1] + 3T[1][0] = 2(1) + 3(1) = 5$. Matches.
$s = 3$: $(i, j) = (1, 2)$ and $(2, 1)$ both satisfy $i + j = 3$. $T[1][2] = 2T[0][2] + 3T[1][1] = 2(1) + 3(5) = 17$ (uses $T[1][1] = 5$, already finished on diagonal $s = 2$). $T[2][1] = 2T[1][1] + 3T[2][0] = 2(5) + 3(1) = 13$ (also uses the finished $T[1][1] = 5$). Both match, regardless of which of the two is updated first, since neither depends on the other.
$s = 4$: only $(i, j) = (2, 2)$. $T[2][2] = 2T[1][2] + 3T[2][1] = 2(17) + 3(13) = 73$, using $T[1][2] = 17$ and $T[2][1] = 13$, both finished on diagonal $s = 3$. Matches.

Both algorithms reproduce the exact correct table for $n = 2$, and the reasoning generalises to any n because B_1's row-major order and B_2's diagonal order both always place a cell's two dependencies strictly earlier in the visiting sequence. So both algorithms are correct.

Both B_1 and B_2 are correct (Option A)

Quick Tip: Check whether each algorithm always computes $T[i-1][j]$ and $T[i][j-1]$ before it computes $T[i][j]$; B1 goes row by row and B2 goes diagonal by diagonal ($i + j$ = constant).

• • •

40. Consider the following 4-variable Boolean function

$$F(A, B, C, D) = \Sigma m(0, 1, 2, 3, 8, 9, 10, 11)$$

Consider A as MSB, D as LSB. Which one of the following options represents the minimal sum of products form for the above function?

Note: + is OR operation, . is AND operation, ' is NOT operation

(A) $A' + B' + C' + D'$

(B) B'

(C) $A'.B' + A.B$

(D) A'

Correct Answer: (B) B'

SOLUTION:

A different way to reach the same answer is to build the 4-variable Karnaugh map directly and read off the grouping, instead of spotting the pattern from the binary listing.

Lay out the K-map with rows for AB (00, 01, 11, 10) and columns for

CD (00, 01, 11, 10), which is the standard Gray code layout so adjacent cells differ in exactly one variable.

Row AB=00 (minterms 0,1,3,2 in column order 00,01,11,10): all four of 0,1,2,3 are in the given list, so this entire row is 1.
Row AB=01 (minterms 4,5,7,6): none of 4,5,6,7 are in the given list, so this entire row is 0.
Row AB=11 (minterms 12,13,15,14): none of these are in the given list, so this entire row is 0.
Row AB=10 (minterms 8,9,11,10): all four of 8,9,10,11 are in the given list, so this entire row is 1.

So the map has two full rows of 1s: the AB=00 row and the AB=10 row, and two full rows of 0s: the AB=01 row and the AB=11 row. Both 1-rows share the property B=0 (row 00 has A=0,B=0; row 10 has A=1,B=0), and together they form the largest possible group on the map, since a group covering all four columns (all of C, D) and both rows where B=0 covers all 8 ones in a single rectangle of size 8. A group of 8 cells eliminates 3 variables from the term, leaving only 1 literal.

Since the group spans both values of A (rows 00 and 10 differ in A) and both values of C and D (all four columns), those three variables drop out of the term entirely, and only B stays, fixed at 0 throughout the group, giving the single literal B'.

Checking that no smaller expression is possible: since the ones occupy exactly half the map (8 out of 16 cells) in a shape that is already the largest legal group size (a group must be a power of 2 in size, and next size up would be 16, which would mean F is always 1, which is false since the AB=01 and AB=11 rows are 0), B', a single literal, is already

the smallest possible product term, so it is the minimal SOP.

This K-map reading matches the earlier pattern-spotting approach exactly.

$$\boxed{F = B' \text{ (Option B)}}$$

Quick Tip: Write out all eight minterms in binary and look for a single variable that is 0 in every one of them.

• • •

41. Consider the canonical $LR(0)$ parsing of the grammar below using terminals $\{a, b, c\}$ and non-terminals $\{A, B, C, S\}$ with S as the start symbol.

$$S \rightarrow ACB$$

$$A \rightarrow aA \mid \epsilon$$

$$C \rightarrow cC \mid \epsilon$$

$$B \rightarrow bB \mid b$$

Which one of the following options gives the number of shift-reduce conflicts that will occur in the $LR(0)$ ACTION table?

(A) 2

(B) 3

(C) 4

(D) 5

Correct Answer: (D) 5

SOLUTION:

Step 1: Spot why conflicts can happen at all.

A shift-reduce clash in $LR(0)$ needs one item with the dot at the very end (a reduce, since $LR(0)$ ignores lookahead and reduces in every column) and another item in the same state with the dot just before a terminal (a shift on that terminal). The two nullable rules $A \to aA \mid \epsilon$ and $C \to cC \mid \epsilon$ are exactly the source of trouble, because the moment A or C is expected, the item $A \to .$ or $C \to .$ appears in the closure right next to the shift item on a or c.

Step 2: Track every place A is expected.

A is expected at the very start, right after $S \to .ACB$ closes in, giving the state $\{S \to .ACB, A \to .aA, A \to .\}$, one clash on a. It is also expected again inside its own recursive rule, after shifting an a: $\{A \to a.A, A \to .aA, A \to .\}$, a second clash on a. So the A side contributes 2 conflicting states.

Step 3: Track every place C is expected.

By the same logic, once A is reduced away, C is expected: $\{S \to A.CB, C \to .cC, C \to .\}$, a clash on c. Then after shifting a c, C is expected again: $\{C \to c.C, C \to .cC, C \to .\}$, a second clash on c. So the C side also contributes 2 conflicting states.

Step 4: Check B separately.

$B \to bB \mid b$ has no empty alternative, so on its own it would not create a clash, since a state built only from $B \to .bB$ and $B \to .b$ has two shift items and no reduce item. But once one b is consumed, we land in $\{B \to b.B, B \to b., B \to .bB, B \to .b\}$: now $B \to b.$ is a completed item sitting with the shift items on b. That is one more clash, on b, coming from the ambiguity between stopping at a single b and continuing into bB.

Step 5: Add them up.

2 conflicts from A's nullable rule, 2 from C's nullable rule, and 1 from B's two alternatives overlapping after one b, gives

$$2 + 2 + 1 = 5$$

$$\boxed{5}$$

Quick Tip: Every place a nullable non-terminal (one with an epsilon rule) is expected in the item set produces a completed item sitting beside a shift item, which is exactly a shift-reduce conflict in an LR(0) table.

42. In the context of schema normalization in relational DBMS, consider a set F of functional dependencies. The set of all functional dependencies implied by F is called the closure of F. To compute the closure of F, Armstrong's Axioms can be applied. Consider X, Y, and Z as sets of attributes over a relational schema. The three rules of Armstrong's Axioms are described as follows.

Reflexivity: If $Y \subseteq X$, then $X \rightarrow Y$
Augmentation: If $X \rightarrow Y$, then $XZ \rightarrow YZ$ for any Z
Transitivity: If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$

The additional rule of Union is defined as follows.
Union: If $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$

It can be proved that the additional rule of Union is also implied by the three rules of Armstrong's Axioms. Listed below are four combinations of these three rules. Which one of these combinations is both necessary and sufficient for the proof?

(A) Reflexivity, Augmentation, and Transitivity

(B) Reflexivity and Augmentation

(C) Transitivity

(D) Augmentation and Transitivity

Correct Answer: (D) Augmentation and Transitivity

SOLUTION:

Step 1: Restate the goal.
We know $X \rightarrow Y$ and $X \rightarrow Z$, and we want $X \rightarrow YZ$, built only from Armstrong's three rules.

Step 2: Write the proof as a short numbered list and mark the rule used at each line.

1. $X \to Y$ (given)
2. $XX \to XY$, that is $X \to XY$ (Augment line 1 with X; uses Augmentation)
3. $X \to Z$ (given)
4. $XY \to YZ$ (Augment line 3 with Y; uses Augmentation)
5. $X \to YZ$ (Combine lines 2 and 4 through Transitivity, since $X \to XY$ and $XY \to YZ$ chain together)

Step 3: Read off which rules got a tick mark.

Scanning the "uses" column above, Augmentation appears twice (lines 2 and 4) and Transitivity appears once (line 5). Reflexivity never appears, because we never had to fall back on a subset argument like $Y \subseteq X$ to manufacture a dependency out of nothing; every dependency we used was either given or built by augmenting a known one.

Step 4: Confirm sufficiency.

Since the five line proof above is complete and correct using only Augmentation and Transitivity, these two rules together are sufficient.

Step 5: Confirm necessity.

Drop Augmentation and there is no way to get from $X \to Y$ to $X \to XY$; drop Transitivity and there is no way to merge $X \to XY$ with $XY \to YZ$. So both are also necessary, and Reflexivity is extra baggage that the proof never touches.

Augmentation and Transitivity

Quick Tip: Augment each given dependency separately to build two intermediate dependencies that share a common attribute set, then chain them with Transitivity; Reflexivity is never actually invoked.

• • •

43. Consider the transmission of data bits 110001011 over a link that uses Cyclic Redundancy Check (CRC) code for error detection. If the generator bit pattern is given to be 1001, which one of the following options shows the remainder bit pattern appended to the data bits before transmission?

(A) 011

(B) 101

(C) 000

(D) 100

Correct Answer: (D) 100

SOLUTION:

Step 1: Set up the padded message.
CRC works by treating both the message and the generator as bit patterns and dividing one by the other using XOR instead of ordinary subtraction. The generator 1001 has 4 bits, so we need 3 check bits, which means padding the message with 3 zeros before dividing:
110001011000

Step 2: Slide a 4-bit window across the padded message, XOR-ing with the generator whenever the front bit is 1.
Start with the first 4 bits, 1100. The front bit is 1, so XOR with 1001:
1100 → 0101
Shift in the next bit (0): window becomes 1010; front bit 1, so XOR:

1010 → 0011
Shift in the next bit (1): window becomes 0111; front bit 0, so no XOR
Shift in the next bit (0): window becomes 1110; front bit 1, so XOR:
1110 → 0111
Shift in the next bit (1): window becomes 1111; front bit 1, so XOR:
1111 → 0110
Shift in the next bit (1): window becomes 1101; front bit 1, so XOR:
1101 → 0100
Shift in the next bit (0): window becomes 1000; front bit 1, so XOR:
1000 → 0001
Shift in the next bit (0): window becomes 0010; front bit 0, so no XOR
Shift in the last bit (0): window becomes 0100; front bit 0, so no XOR, and no bits remain

Step 3: Read the remainder.
All 9 message bits plus the 3 padding zeros have now been fed through the window, and this window always ends up 4 bits wide, one bit wider than the true remainder. The last 3 bits of the final window are the CRC remainder, since a 4-bit generator always leaves exactly a 3-bit remainder:

$$\boxed{100}$$

This 3-bit pattern, 100, is what gets tacked onto the end of the original 9-bit message before it is sent over the link.

Quick Tip: Pad the message with (generator length minus 1) zeros, then perform modulo-2 (XOR) division by the generator; the final remainder, of that same length, is the CRC bits appended before transmission.

44. Consider a processor that has 16 general purpose registers and it uses 2-byte instruction format for all its instructions. Variable-sized opcodes are permitted. There are three different types of instructions; M-type, R-type, and C-type. Each M-type instruction has 2 register operands and a 6-bit immediate operand. Each R-type instruction has 3 register operands. Each C-type instruction has a register operand and a 6-bit offset value. If there are 2 unique M-type opcodes and 7 unique R-type opcodes, which one of the following options gives the maximum number of unique opcodes possible for C-type instructions?

(A) 8

(B) 4

(C) 64

(D) 16

Correct Answer: (B) 4

SOLUTION:

Step 1: Picture the 16-bit instruction as a space of code points.
Every instruction is 16 bits wide, so there are 2^{16} possible bit patterns in total. We are going to slice this space into pieces for M-type, R-type and C-type, working out how many full instruction slots each opcode swallows.

Step 2: Work out operand widths first.
With 16 registers, a register field takes $\log_2 16 = 4$ bits. M-type needs 2 registers plus a 6-bit immediate, that is $8 + 6 = 14$ operand bits, leaving a 2-bit opcode. R-type needs 3 registers, that is 12 operand bits, leaving a 4-bit opcode. C-type needs 1 register plus a 6-bit offset, that is $4 + 6 = 10$ operand bits, leaving a 6-bit opcode.

Step 3: Convert every opcode into how many full instructions it represents.

An opcode of width k bits, combined with its own operand bits, represents a slice of the 16-bit space of size 2^{16-k}, since the remaining $16 - k$ bits are free to vary. So one M-type opcode ($k = 2$) represents 2^{14} instructions, one R-type opcode ($k = 4$) represents 2^{12} instructions, and one C-type opcode ($k = 6$) represents 2^{10} instructions.

Step 4: Add up what M-type and R-type already claim.

2 M-type opcodes claim $2 \times 2^{14} = 2^{15}$ instructions. 7 R-type opcodes claim 7×2^{12} instructions.

Step 5: Convert everything into C-type-sized units and subtract.

One C-type opcode's worth is 2^{10} instructions. In these units:

$$2^{16} = 64 \times 2^{10}, \qquad 2 \times 2^{14} = 32 \times 2^{10}, \qquad 7 \times 2^{12} = 28 \times 2^{10}$$

So the whole space is 64 units, M-type takes 32 units, and R-type takes 28 units, leaving

$$64 - 32 - 28 = 4$$

Step 6: Conclude.

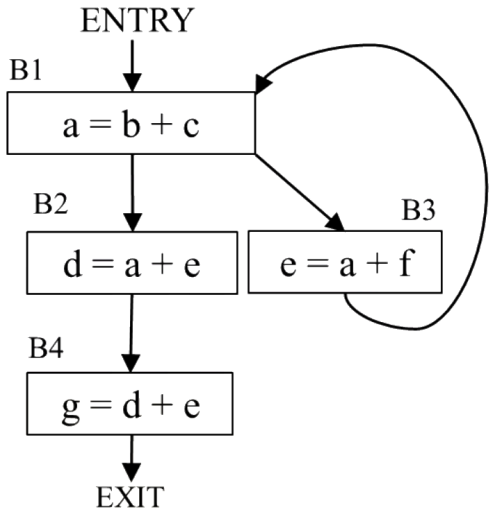
Exactly 4 C-type opcodes fit in the space that is left over.

$$\boxed{4}$$

Quick Tip: Convert every instruction type's opcode length into how many 16-bit code points it uses, 2 raised to (16 minus opcode bits) per opcode, then see how many C-type-sized slots remain once the M-type and R-type opcodes have claimed their share.

• • •

45. Consider the control flow graph given below.



Which one of the following options is the set of live variables at the exit point of each basic block?

(A) $B_1 : \{a, b, c, e, f\},\ B_2 : \{d, e\},\ B_3 : \{b, c, e, f\},\ B_4 : \emptyset$

(B) $B_1 : \emptyset,\ B_2 : \{d, e\},\ B_3 : \{a, c, f\},\ B_4 : \emptyset$

(C) $B_1 : \{a, b, c, e, f\},\ B_2 : \{d, e\},\ B_3 : \{c, e, f\},\ B_4 : \emptyset$

(D) $B_1 : \emptyset,\ B_2 : \{d, e, f\},\ B_3 : \{a, b, c, e, f\},\ B_4 : \emptyset$

Correct Answer: (A) $B_1 : \{a, b, c, e, f\},\ B_2 : \{d, e\},\ B_3 : \{b, c, e, f\},\ B_4 : \emptyset$

SOLUTION:

Step 1: List what each block reads and writes.

$B_1 : a = b + c$ reads $\{b, c\}$, writes a. $B_2 : d = a + e$ reads $\{a, e\}$, writes d.

$B_3 : e = a + f$ reads $\{a, f\}$, writes e. $B_4 : g = d + e$ reads $\{d, e\}$, writes g.

The control leaves ENTRY, passes through B_1, splits to B_2 and B_3, with B_3 looping back to B_1, while B_2 continues to B_4 and then to EXIT.

Step 2: Build a small table and fill it from the bottom up.
Since live variable analysis reads backward, start at the block closest to EXIT. B_4 has no successor, so whatever it reads is exactly what must be live coming in: $\textit{live-in}(B_4) = \{d, e\}$, and $\textit{live-out}(B_4) = \varnothing$.

Step 3: Move to B_2, whose only successor is B_4.
$\textit{live-out}(B_2) = \textit{live-in}(B_4) = \{d, e\}$. Remove what B_2 overwrites (d) and add what it reads (a, e):

$$\textit{live-in}(B_2) = (\{d, e\} \setminus \{d\}) \cup \{a, e\} = \{a, e\}$$

Step 4: The loop needs a guess-and-correct pass.
Guess $\textit{live-in}(B_3) = \varnothing$ at first. Then

$$\textit{live-out}(B_1) = \textit{live-in}(B_2) \cup \textit{live-in}(B_3) = \{a, e\}$$

$$\textit{live-in}(B_1) = (\{a, e\} \setminus \{a\}) \cup \{b, c\} = \{b, c, e\}$$

Since $B_3 \to B_1$, $\textit{live-out}(B_3) = \textit{live-in}(B_1) = \{b, c, e\}$, so

$$\textit{live-in}(B_3) = (\{b, c, e\} \setminus \{e\}) \cup \{a, f\} = \{a, b, c, f\}$$

This differs from our starting guess of $\varnothing$, so we redo the loop with this better guess.

Step 5: Second pass with the corrected *live-in*(B_3).

$$\textit{live-out}(B_1) = \{a, e\} \cup \{a, b, c, f\} = \{a, b, c, e, f\}$$

$$\textit{live-in}(B_1) = (\{a, b, c, e, f\} \setminus \{a\}) \cup \{b, c\} = \{b, c, e, f\}$$

$$\textit{live-out}(B_3) = \textit{live-in}(B_1) = \{b, c, e, f\}$$

$$\textit{live-in}(B_3) = (\{b, c, e, f\} \setminus \{e\}) \cup \{a, f\} = \{a, b, c, f\}$$

This matches the guess we fed in, so the loop values have settled.

Step 6: Write down the four live-out answers asked for.

$$B_1 : \{a, b, c, e, f\} \quad B_2 : \{d, e\} \quad B_3 : \{b, c, e, f\} \quad B_4 : \varnothing$$

Step 7: Match to the options and conclude.
This exact grouping appears as option (A); the other options either wrongly zero out B_1, drop b from B_3, or invent an f in B_2 that block never touches.

$$\boxed{B_1 : \{a, b, c, e, f\},\ B_2 : \{d, e\},\ B_3 : \{b, c, e, f\},\ B_4 : \varnothing}$$

Quick Tip: Live-out of a block is the union of live-in of its successors; since B1 and B3 form a loop, iterate the equations until the sets stop changing before reading off the live-out values.

• • •

46. An index in a DBMS is said to be dense if an index entry appears for every search-key value in the indexed file. Otherwise it is called a sparse index. Consider the following two statements.
S1: A hash index must be a dense index
S2: A B^+ tree index can be a sparse index

Which one of the following options is correct?

(A) Both S1 and S2 are true

(B) Both S1 and S2 are false

(C) S1 is true and S2 is false

(D) S1 is false and S2 is true

Correct Answer: (A) Both S1 and S2 are true

SOLUTION:

Step 1: Pin down the two words first.
Dense means the index has a separate entry for every single key value in the file. Sparse means the index only has entries for some key values, usually one per block, and it only works when the data file itself is kept sorted on that key so a search can scan forward from the nearest entry.

Step 2: Think about how a hash index actually finds a record.
A hash index turns a key straight into a bucket address using a hash

function. There is no nearby entry to fall back on, because hashed values are scattered with no order. So if the key you want has no entry, the hash index simply cannot find it. That forces every key to have its own entry, so hashing is always dense, statement S1 holds.

Step 3: Think about a B^+ tree used as the file's main (primary) index.
If the underlying data file is sorted by the search key, a B^+ tree does not need to record every key. It can record just the first key of each block, and once a search lands in the right block, a normal sequential scan inside that block finds the record. Fewer entries mean a smaller, faster index. This sparse setup only works for a primary index; a secondary B^+ tree index on an unsorted file would still need to be dense. So statement S2, which only claims a B^+ tree can be sparse, not that it always is, also holds.

Step 4: Put the two verdicts together.
S1 is true, S2 is true, so the correct choice is the one that marks both as true.

Step 5: Eliminate the rest quickly.
"Both false" ignores the reasoning entirely. "S1 true, S2 false" wrongly denies that a B^+ tree can ever be sparse. "S1 false, S2 true" wrongly claims hashing does not need to be dense, which breaks how hash lookups work.

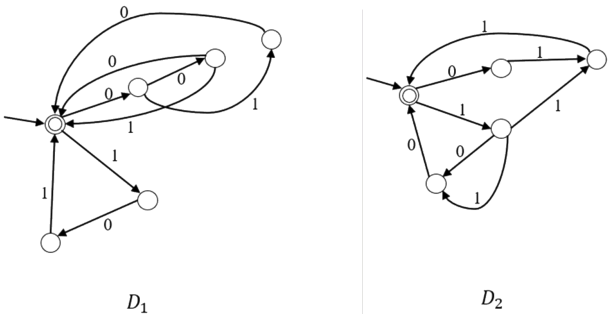
Both S1 and S2 are true

Quick Tip: A hash index has no ordering to fall back on, so it must record every key (dense), while a B+ tree used as a primary index on sorted data can

skip most keys (sparse).

• • •

47. Consider the following two finite automata D_1 and D_2.



Which of the following statements is/are true?

(A) $L(D_1) = L(D_2)$

(B) $L(D_1)$ is a proper subset of $L(D_2)$

(C) $L(D_1) \cap L(D_2) = \{\epsilon\}$

(D) $(L(D_1) \cup L(D_2))^*$ consists of all strings in $\{0, 1\}^*$ whose length is divisible by 3

Correct Answer: (C) $L(D_1) \cap L(D_2) = \{\epsilon\}$; (D) $(L(D_1) \cup L(D_2))^*$ consists of all strings in $\{0, 1\}^*$ whose length is divisible by 3

SOLUTION:

Step 1: Notice the shape both machines share.
In D_1 and in D_2, the double circle marks both the start state and the only accepting state. That means a string gets accepted only if it walks the automaton in a full loop, leaving the start and coming straight back to it.

Step 2: Think of acceptance as returning home.
Trace the arrows in D_1: any path that leaves the start state and

eventually lands back on it again always takes exactly three arrows, never one, two, four, or five. The same check on D_2 gives the same fact, every trip back home takes exactly three arrows. So both machines only ever accept strings whose length is a multiple of 3, and both accept the empty string ϵ for free, since standing still already counts as being at home.

Step 3: Test option (A) by asking if the two machines agree on every string.

The two diagrams send 0s and 1s along different arrows at corresponding points in their loops, so the actual three-letter (and longer) words that bring each machine home form different lists for D_1 and D_2. Two different lists mean $L(D_1) \neq L(D_2)$, so (A) fails.

Step 4: Test option (B) with a subset check.

A subset relationship would need every word accepted by D_1 to also be accepted by D_2. Since D_1 has words in its accepted list that are not on D_2's list, because their loop patterns differ, this containment breaks down, so (B) fails too.

Step 5: Test option (C) by hunting for a shared non-empty word.

Suppose some non-empty string worked for both. It would have to close a loop in D_1 using its arrows and close a loop in D_2 using its arrows, with the very same sequence of bits both times. Because the arrow labels at matching points in the two loops do not line up, no such shared non-empty string exists. Only ϵ survives in both lists, so

$$L(D_1) \cap L(D_2) = \{\epsilon\}$$

and (C) holds.

Step 6: Test option (D) by thinking of blocks stacked end to end. Every accepted word from either machine is a block whose length is some multiple of 3. Kleene star on $L(D_1) \cup L(D_2)$ means gluing any number of such blocks together in any order. Between the two machines' accepted words, every possible pattern of 0s and 1s of length 3 shows up in at least one of the two lists, so any string whose total length is a multiple of 3 can always be cut into three-letter chunks that each come from D_1 or D_2. Nothing of a length not divisible by 3 can ever appear, since each block forces a multiple of 3. So

$$(L(D_1) \cup L(D_2))^* = \{w \in \{0,1\}^* : |w| \equiv 0 \pmod 3\}$$

and (D) holds too.

Step 7: Conclude.

$$\boxed{\text{(C) and (D) are true}}$$

Quick Tip: Since the start state is the only accepting state in both automata, a string is accepted only if it traces a closed walk back to the start; check the length of every such walk and compare the specific bit patterns each machine allows.

• • •

48. Let $\Sigma = \{a, b, c, d\}$ and let $L = \{a^i b^j c^k d^l \mid i, j, k, l \geq 0\}$. Which of the following constraints ensure(s) that the language L is context-free?

(A) $i + k = j + l$

(B) $i = k$ and $j = l$

(C) $i = l$ and $j = k$

(D) $i + j = k + l$

Correct Answer: (A) $i + k = j + l$; (C) $i = l$ and $j = k$; (D) $i + j = k + l$

SOLUTION:

Step 1: What context-free means here.
A language is context-free if some pushdown automaton, a finite control plus one stack, accepts it, or equivalently if some context-free grammar generates it. We have four candidate extra rules on i, j, k, l for strings of the form $a^i b^j c^k d^l$, and we test each one separately.

Step 2: Test $i + k = j + l$, option A, with a stack machine.
Build a machine that pushes a marker for each a and each c, and pops a marker for each b and each d, reading the string left to right in the fixed order a's, then b's, then c's, then d's. Since the letters that add to the stack and the letters that remove from it never need to be tracked at the same time as a second, independent quantity, the stack empties exactly when $i + k = j + l$. This works, so option A is context-free.

Step 3: Test $i + j = k + l$, option D, the same way.
Push for a's and b's, pop for c's and d's. Again only one running total is ever needed at a time, so this is context-free by the same one stack argument as option A.

Step 4: Test $i = l$ and $j = k$, option C, with a grammar.
Write $S \to aSd \mid T$ and $T \to bTc \mid \varepsilon$. Expanding S a number of times matches every leading a with a trailing d, giving $i = l$, and expanding T

matches every b with a following c, giving $j = k$. Both matchings are properly nested inside each other, not crossed, so an ordinary context-free grammar builds the language directly. Option C is context-free.

Step 5: Test $i = k$ and $j = l$, option B, with the pumping lemma.
Suppose this language were context-free with pumping length p. Take the string $a^p b^p c^p d^p$, which satisfies $i = k = p$ and $j = l = p$. The pumping lemma lets us split it as $uvwxy$ with $|vwx| \leq p$ and $|vx| \geq 1$, then pump v, x together. Because $|vwx| \leq p$, the piece vwx can touch at most two of the four letter blocks, it cannot reach across three or four blocks at once. Pumping up then increases the count of at most two of the four letters and leaves the other two unchanged, which always destroys either $i = k$ or $j = l$, since those two equalities pair up NON adjacent blocks, block one with block three, block two with block four. So no valid split exists, and the language is not context-free. Option B fails.

Step 6: Collect the result.
Only the constraints that let a single stack track one running total, A and D, or that produce a properly nested bracket structure, C, survive. The crossing constraint in B does not.

Options A, C, D are context-free; option B is not

Quick Tip: Check whether each numeric constraint can be verified with a single stack acting as one running counter, or whether it forces two independent counts to be remembered at the same time.

49. Consider a binary search tree (BST) with n leaf nodes ($n > 0$). Given any node V, the key present in the node is denoted as $Val(V)$. All the keys present in the given BST are distinct. The keys belong to the set of real numbers.

For a node V, let $Suc(V)$ denote the node that is its inorder successor. If a node V does not have an inorder successor, then $Suc(V)$ is $NULL$. As there are no duplicates, if $Suc(V)$ is not $NULL$, then $Val(V) < Val(Suc(V))$.

Corresponding to every leaf node L_i that has a non-NULL $Suc(L_i)$, a new key k_i with the following property is to be inserted into the BST.

$$Val(L_i) < k_i < Val(Suc(L_i))$$

Let K represent the list of all such new keys to be inserted into the BST. Which of the following statements is/are true?

(A) K cannot have any duplicates

(B) K will have at least one element

(C) After inserting all keys from K, the height of the BST can increase at most by one

(D) Number of nodes in the BST will double after inserting all keys from K

Correct Answer: (A) K cannot have any duplicates ; (C) After inserting all keys from K, the height of the BST can increase at most by one

SOLUTION:

Step 1: Picture a small example first.
Take a BST with root 20, left leaf 10, and right leaf 30. Here $Suc(10) = 20$, which is not a leaf so it is not our concern, and $Suc(30) = NULL$ since 30 is the maximum key in the tree. So among the leaves 10 and

30, only leaf 10 gets a new key, some k with $10 < k < 20$.

Step 2: Generalize the pattern for duplicates, this checks (A).
List every leaf that has a successor, in increasing order of key. The open gaps between each leaf and its successor never overlap, because each gap sits strictly between two consecutive values of the full sorted key list of the tree, and consecutive gaps in a sorted list never share a point. So each new key is pulled from its own private gap, and the whole list K is automatically duplicate free. This confirms (A).

Step 3: Find a counterexample for at least one element, this checks (B).
A tree with just one node has that single node as both the first and last item in sorted order, so it has no successor at all. Then K is the empty list, which directly disproves (B).

Step 4: Track exactly where insertion lands, needed for (C) and (D).
A leaf L_i with a successor has no right child. The classic BST fact is that when a key has no right subtree, its successor is found by walking up until the first left turn, and this also means there is no key lying strictly between L_i and $Suc(L_i)$ anywhere in the tree. Inserting a new key from this exact open gap therefore has to follow the path to L_i and then drop in as the new right child of L_i, since L_i has room on its right.

Step 5: Bound the height change, this checks (C).
Every insertion only ever adds a node one level below some existing leaf, and different insertions target different original leaves, so they cannot pile up on the same path. The tallest possible new leaf is therefore one level below whichever original leaf was tallest, so the height grows by at most 1. This proves (C).

Step 6: Rule out doubling, this checks (D).

At most $n - 1$ new nodes get added, all leaves except the single overall maximum key, while the tree could have far more than n nodes in total once internal nodes are counted, so doubling is not guaranteed in general. (D) fails.

Step 7: Conclude.

K has no duplicates, and the height rises by at most one, so (A) and (C) hold

Quick Tip: Since a leaf with a successor has no right child, the new key always becomes that leaf's new right child, going one level deeper than the leaf itself.

50. Consider a stack S and a queue Q. Both of them are initially empty and have the capacity to store ten elements each. The elements $1, 2, 3, 4$, and 5 arrive one by one, in that order. When an element arrives, it is assigned either to S (pushed on S) or to Q (enqueued to Q). Once all the five elements are stored, the output is generated in two steps. First, stack S is emptied by popping all elements. Then queue Q is emptied by dequeueing all elements. The output obtained by following this process is **4 3 1 2 5**.

Given the output, the objective is to predict whether an element was assigned to S or Q.

Which of the following options is/are possible valid assignment(s) of the elements?

Note: In the options, the notation xS denotes that element x was assigned to S and yQ denotes that element y was assigned to Q.

(A) $1S, 2Q, 3S, 4S, 5Q$

(B) $1Q, 2Q, 3S, 4S, 5Q$

(C) $1Q, 2Q, 3Q, 4S, 5S$

(D) $1S, 2S, 3S, 4Q, 5Q$

Correct Answer: (A) $1S, 2Q, 3S, 4S, 5Q$; (B) $1Q, 2Q, 3S, 4S, 5Q$

SOLUTION:

Step 1: Split the target output at every possible point.
The full output is **4 3 1 2 5**, five numbers. It is built as the reversed stack sequence followed by the queue sequence in order, so the target string must break into a front part that is some reversal of the numbers pushed to S, and a back part that is exactly the numbers pushed to Q in order. We try each option's stack elements, reverse

them, and see if they line up with the front of the target.

Step 2: Check option A, stack elements {1, 3, 4} pushed in order 1, 3, 4. Reversing 1, 3, 4 gives 4, 3, 1, which is exactly the first three numbers of the target 4, 3, 1, 2, 5. The queue elements {2, 5} in order 2, 5 match the remaining tail 2, 5 exactly. Both halves line up, so A works.

Step 3: Check option B, stack elements {3, 4} pushed in order 3, 4. Reversing gives 4, 3, matching the first two numbers of the target. The queue elements {1, 2, 5} in order 1, 2, 5 match the remaining tail 1, 2, 5 exactly. So B also works.

Step 4: Check option C, stack elements {4, 5} pushed in order 4, 5. Reversing gives 5, 4. The target output starts with 4, 3, not 5, 4, so this already fails to match the front of the target. C does not work.

Step 5: Check option D, stack elements {1, 2, 3} pushed in order 1, 2, 3. Reversing gives 3, 2, 1, which does not match the required front 4, 3, 1 of the target either, since it starts with 3 instead of 4. D does not work.

Step 6: Conclude.
Only the assignments in A and B reverse to the correct front section and leave the correct tail section for the queue.

Options A and B are the valid assignments

Quick Tip: Reverse only the elements sent to the stack, keep the elements sent to the queue in their original arrival order, and check which split matches 4 3 1 2 5.

• • •

51. Consider three processes $P1$, $P2$, and $P3$ running identical code, as shown in the pseudocode below. A and B are two binary semaphores initialized to 1 and 0, respectively. X is a shared variable initialized to 0. Each line in the pseudocode is executed atomically.

Pseudocode of P1, P2, and P3

```
Wait(A);
Print(*);
X = X+1;
If (X == 2)
 {
   Print($);
   Signal(B);
 }
Signal(A);
Wait(B);
Print(#);
Signal(B);
```

Assume that any of the three processes can start to execute first and context switching can happen between these processes at any arbitrary time and in any arbitrary order.

Which of the following patterns is/are possible to be generated as an outcome of the execution of these three processes?

(A) **$*###

(B) **$#*##

(C) **$###*#

(D) ***$###

Correct Answer: (A) **

You can't use 'macro parameter character #' in math mode#*## ; (C) **$###*#

SOLUTION:

Step 1: Notice A acts as a lock.

Semaphore A begins at 1, so only one of the three processes can be between its $Wait(A)$ and its own later $Signal(A)$ at any moment. This forces the three processes through that stretch of code one at a time, in some order picked by the scheduler; call them the first, second, and third to pass through.

Step 2: Write down what each one prints inside the lock.

The first one to pass through sets X to 1 and prints only a star. The second one sets X to 2, so it prints a star and then a dollar sign, and it also signals B once, all before releasing the lock. The third one sets X to 3 and prints only a star, finding the condition false again.

Step 3: Fix the unavoidable order of the first four prints.

The third star cannot appear until the second process releases the lock, which happens only after that same process has already printed its dollar sign. So the first four characters printed are always star, star, dollar, star, in that fixed order, whichever physical process plays which role. Any candidate pattern with all three stars ahead of the dollar sign is impossible right away.

Step 4: See how the hash marks get freed up.
B starts at 0, so the first process to reach $Wait(B)$, right after it leaves the lock, finds B at 0 and waits there. The second process, right after printing the dollar sign, signals B once; since someone is already waiting, that signal wakes the FIRST process rather than being banked for later. The second process then tries its own wait on B and, finding the count back at 0, ends up waiting itself. From here on, every time a process finishes printing its hash mark, it signals B again, releasing whichever process is waiting next. So the three hash marks come out one at a time, each one triggered by the previous process finishing.

Step 5: See how the third star can slot in at different points.
The third process becomes free to enter the lock the moment the second process signals A, which happens right after it has already printed the dollar sign. Whether the scheduler lets the third process run immediately, or instead runs one or two of the already unblocked hash printing processes first, is completely open, since arbitrary interleaving is allowed. This single choice, of when the third star gets slotted in relative to the chain of hash marks, is what produces the different valid patterns.

Step 6: Match the three allowed slots to the given options.
If the third star runs before any hash mark, the output reads star star dollar star hash hash hash, matching option (A). If exactly one hash mark comes out first, then the third star, then the remaining two hash marks, the output reads star star dollar hash star hash hash, matching option (B). If both remaining hash marks come out before the third star finally runs, the output reads star star dollar hash hash star hash, matching option (C).

Step 7: Explain why option (D) never happens.
Option (D) needs three stars printed before any dollar sign, but Step 3 already shows the dollar sign must come out right after the second star and strictly before the third. So (D) can never be produced.

Step 8: Conclude.

Patterns (A), (B), and (C) are all reachable; (D) is not

Quick Tip: Semaphore A lets only one process print its star, and possibly the dollar sign, at a time; semaphore B then releases the three hash marks in whatever order the scheduler wakes the waiting processes.

• • •

52. Consider a system with a processor and a 4 KB direct mapped cache with block size of 16 bytes. The system has a 16 MB physical memory. Four words P, Q, R, and S are accessed by the processor in the same order 10 times. That is, there are a total of 40 memory references in the sequence $P, Q, R, S, P, Q, R, S, \ldots$

Assume that the cache memory is initially empty. The physical addresses of the words are given below (1 word = 1 byte).

$P : 0x845B32,\ Q : 0x845B26,\ R : 0x845B36,\ S : 0x846B32$

Which of the following statements is/are true?

Note: $1K = 2^{10}$ and $1M = 2^{20}$

(A) Every access to P results in a cache miss

(B) Every access to R results in a cache hit

(C) Every access to Q results in a cache miss

(D) Except the first access to S, all subsequent accesses to S result in cache hits

Correct Answer: (A) Every access to P results in a cache miss ; (B) Every access to R results in a cache hit

SOLUTION:

Step 1: Get the index and offset widths from the cache size.
Cache size is 4 KB and block size is 16 bytes, so there are $4096/16 = 256$ lines, needing 8 bits for the index. A 16 byte block needs 4 bits for the offset. So the lowest 4 bits of an address pick the byte inside a block, and the next 8 bits above that pick the cache line.

Step 2: Convert the addresses using only their low bits.
We only need the lowest 12 bits of each address, the same as the lowest 3 hex digits.
P: last three hex digits $B32$, offset $= 2$, index byte $= B3$.
Q: last three hex digits $B26$, offset $= 6$, index byte $= B2$.
R: last three hex digits $B36$, offset $= 6$, index byte $= B3$.
S: last three hex digits $B32$, offset $= 2$, index byte $= B3$.
The tag is everything above these 12 bits, which is 845 for P, Q, R and 846 for S.

Step 3: Draw the conflict picture.
Line $B2$ is used only by Q, so once it is loaded it is never disturbed again; every access after the first is a hit there. Line $B3$ is shared by three different accesses per round: P and R belong to the same block,

tag 845, while S belongs to a different block, tag 846, that maps to the same line. Since a direct mapped cache allows only one block per line, the P/R block and the S block constantly kick each other out of line $B3$.

Step 4: Follow one full round once the pattern has settled.
Suppose line $B3$ holds S's block, tag 846, at the start of a round, left over from the previous round's last access. Then: P needs tag 845, finds 846, MISS, reloads 845. Q needs line $B2$, untouched, HIT. R needs tag 845 at line $B3$, which P just loaded, HIT. S needs tag 846 at line $B3$, which currently holds 845, MISS, reloads 846. This leaves line $B3$ holding 846 again at the end of the round, exactly the condition assumed at the start, so the same four outcomes, miss, hit, hit, miss, repeat identically in every later round.

Step 5: Handle the very first round separately.
On the first round the cache starts empty rather than holding 846, but the outcome is the same: P misses, empty line, Q misses, empty line, but this is its only miss ever, R hits, block just loaded by P, S misses, different tag at line $B3$.

Step 6: Read off which statements always hold.
P: miss in every one of the 10 rounds, so every access to P is a miss is true. R: hit in every one of the 10 rounds, so every access to R is a hit is true. Q: misses only in round 1, so every access to Q is a miss is false. S: misses in every round, not just the first, so all accesses after the first being hits is false.

Step 7: Conclude.

Statements (A) and (B) are correct

Quick Tip: Work out the tag and index for each address and see which of P, Q, R, S share a cache line and which stay isolated on their own line.

• • •

53. To keep track of free blocks in a file system, one of the two approaches is generally used, using bitmaps (bit vectors) or using linked lists. Consider that the linked list approach is used to keep track of free blocks in a file system. Assume that the disk size is 16 GB, block size is 2 KB, and block numbers used are 32-bit long. A single pointer of size 4 bytes is used in each block of the list to point to the next block of the list. The number of blocks required to hold the free disk block numbers is _____. (*answer in integer*)

Note: $1K = 2^{10}$ and $1G = 2^{30}$

Correct Answer: 16417

SOLUTION:

Step 1: Count how many blocks exist on the whole disk.

$$\text{Total blocks} = \frac{16\ \text{GB}}{2\ \text{KB}} = \frac{2^{34}}{2^{11}} = 2^{23} = 8388608$$

This is the largest possible number of free block numbers we might have to store.

Step 2: Work out the raw byte requirement for storing all these numbers.

Each block number takes 4 bytes (32 bits), so storing all of them, ignoring the pointer overhead for a moment, would take

$$8388608 \times 4 = 33554432 \text{ bytes}$$

Step 3: Work out the usable space inside one list block.
Each list block is 2048 bytes, but 4 of those bytes are used up by the next block pointer, so only $2048 - 4 = 2044$ bytes in each block actually store block numbers, which is 511 numbers per block, since $2044/4 = 511$.

Step 4: Find how many list blocks are needed using a ceiling division.
We need the smallest whole number of blocks n such that $511\,n \geq 8388608$. Dividing directly,

$$n \geq \frac{8388608}{511} \approx 16416.06$$

so n must be at least 16417, the next whole number up from 16416.06, since 16416 blocks alone can only hold

$$511 \times 16416 = 8388576$$

which is 32 short of the full 8388608 block numbers we need to store.

Step 5: Confirm with the extra block.
One more block adds room for another 511 numbers, comfortably covering the missing 32, so 16417 blocks are enough and 16416 are not.

$$\boxed{16417}$$

Quick Tip: Each 2 KB list block loses 4 bytes to its next pointer, leaving room for 511 four byte block numbers per block.

• • •

54. A system has a Translation Lookaside Buffer (TLB) that has a reach of 1 MB. TLB reach is defined as the total amount of physical memory that can be accessed through the TLB entries. The paging system uses pages of size 4 KB. The virtual address space is 64 GB and physical address space is 1 GB. If each TLB entry stores a 4-bit process id, page number, frame number, and a 2-bit control field, then the size of the TLB (in bytes) is _____. (*answer in integer*)

Note: $1K = 2^{10}$, $1M = 2^{20}$, $1G = 2^{30}$

Correct Answer: 1536

SOLUTION:

Step 1: Work out the size of one TLB entry first.
An entry needs to identify a process, 4 bits, a virtual page, a physical frame, and carry 2 control bits. Let's find the two unknowns.

Step 2: Bits for the page number, from the virtual address space.
Virtual space is 64 GB, pages are 4 KB, so the number of distinct pages is

$$\frac{64 \times 2^{30}}{4 \times 2^{10}} = \frac{2^{36}}{2^{12}} = 2^{24}$$

so 24 bits are needed to number a page uniquely.

Step 3: Bits for the frame number, from the physical address space. Physical space is 1 GB, and frames match the 4 KB page size, so the number of frames is

$$\frac{1 \times 2^{30}}{4 \times 2^{10}} = \frac{2^{30}}{2^{12}} = 2^{18}$$

so 18 bits are needed for the frame number.

Step 4: Add every field of one entry in bits, then convert to bytes.

$$4\ (\text{pid}) + 24\ (\text{page}) + 18\ (\text{frame}) + 2\ (\text{control}) = 48\ \text{bits}$$

Since 8 bits make a byte, one entry takes

$$\frac{48}{8} = 6\ \text{bytes}$$

Step 5: Now find how many entries the TLB holds, from its reach. TLB reach tells us how much memory the TLB can cover if fully used, which equals the number of entries times the page size, so

$$\text{entries} = \frac{\text{TLB reach}}{\text{page size}} = \frac{1\ \text{MB}}{4\ \text{KB}} = \frac{2^{20}}{2^{12}} = 2^{8} = 256$$

Step 6: Multiply entry size by entry count for the total TLB size.

$$\text{TLB size} = 256 \times 6 = 1536\ \text{bytes}$$

1536 bytes

Quick Tip: Find the number of TLB entries from reach divided by page size, and the entry size from adding the process id, page number, frame number, and control bits.

• • •

55. Consider contiguous allocation of physical memory to processes using variable partitioning scheme. Suppose there are 8 holes in the memory of sizes 20 KB, 4 KB, 25 KB, 18 KB, 7 KB, 9 KB, 15 KB, and 12 KB. Assume that no two holes are adjacent. Two processes P1 of size 16 KB and P2 of size 9 KB arrive in that order, and they are allocated memory using the best-fit technique. After allocating space to P1 and P2, the number of holes of size less than 8 KB is _____.

Note: $1K = 2^{10}$

Correct Answer: 3

SOLUTION:

Step 1: Set the rule for best fit.
Best fit always picks the tightest hole, the smallest one that can still hold the incoming process. Starting holes: $20, 4, 25, 18, 7, 9, 15, 12$ (all in KB).

Step 2: Fit P1, which needs 16 KB.
Holes big enough for 16 KB are $20, 25, 18$. The tightest of these is 18. Placing P1 there leaves a stub of $18 - 16 = 2$ KB in place of the old 18 KB hole.

Updated list: 20, 4, 25, 7, 9, 15, 12, 2.

Step 3: Fit P2, which needs 9 KB.
Holes big enough for 9 KB are 20, 25, 9, 15, 12. The tightest is 9, an exact match, so this hole is used up completely and vanishes from the list.
Updated list: 20, 4, 25, 7, 15, 12, 2.

Step 4: Sort by size and mark the small ones.
Writing the list in order: 2, 4, 7, 12, 15, 20, 25. The values under 8 KB are 2, 4, 7, which is 3 holes.

3

Quick Tip: Use best fit: pick the smallest hole that still fits the process, split off the leftover as a new smaller hole, and remove a hole completely if the process fits it exactly.

• • •

56. Consider a system with 1 MB physical memory and a word length of 1 byte. The system uses a direct mapped cache, with block numbers starting from 0. The word with physical address 0xA2C28 is mapped to the cache block number 176_{10}. The maximum possible size of the cache (in KB) is _____.

Note: $1K = 2^{10}$ and $1M = 2^{20}$

Correct Answer: 128

SOLUTION:

Step 1: Turn everything into plain numbers.

1 MB of memory with 1 byte words means 2^{20} addressable bytes, so addresses run from 0 to $2^{20} - 1$. The given address in decimal is $\text{0xA2C28} = 666664$.

Step 2: Write the direct mapped cache formula.

If the block size is 2^w bytes and there are 2^b blocks, then the block that holds address A is

$$\text{block} = \left\lfloor \frac{A}{2^w} \right\rfloor \bmod 2^b$$

and the cache size is $2^w \times 2^b = 2^{w+b}$ bytes. We want the biggest possible $w + b$ that still gives block number 176.

Step 3: Try shrinking the word field.

Dividing 666664 by increasing powers of two and dropping the remainder is the same as chopping bits off the right end of the address. Doing this, the smallest working offset turns out to be $w = 6$, since

$$\left\lfloor \frac{666664}{2^6} \right\rfloor = \left\lfloor \frac{666664}{64} \right\rfloor = 10416$$

Step 4: Check 10416 against 176.

Write 10416 in binary: 10100010110000. Its rightmost 8 bits read 10110000, which is exactly 176. So the last 8 bits already match block number 176.

Step 5: Stretch the modulus as far as possible.

Moving from those rightmost 8 bits of 10416 further left, the next three bits are all 0, so $10416 \bmod 2^9$, $\bmod 2^{10}$, and $\bmod 2^{11}$ all still equal 176. The bit right after that is a 1, so 2^{11} is the largest power of two we can use here, meaning $b = 11$.

Step 6: Add the offset back in.

Total bits used $= w + b = 6 + 11 = 17$.

Cache Size $= 2^{17}$ bytes $= 128 \times 1024$ bytes $= 128$ KB

$$\boxed{128}$$

Quick Tip: Write 0xA2C28 as a 20-bit binary string, locate the 8-bit pattern for 176, then extend the index field upward through any leading zero bits to maximize block-index-plus-offset width.

• • •

57. A non-pipelined instruction execution unit that operates at 1.6 GHz clock takes an average of 5 clock cycles to complete the execution of an instruction. To improve the performance, the system was pipelined with a goal of achieving an average throughput of one instruction per clock cycle. However, it could operate only at 1.2 GHz due to pipeline overheads. While executing a program in the pipelined design, 30% of instructions encountered a stall of 2 cycles due to pipeline hazards. The speed-up obtained by the pipelined design over the non-pipelined one for this program is _____ (rounded off to two decimal places).

Note: $1G = 10^9$

Correct Answer: 2.30 to 2.40

SOLUTION:

Step 1: Pick a convenient batch of instructions.

Say the program has 100 instructions. This makes the percentages easy to turn into instruction counts.

Step 2: Time for the non-pipelined design.

Clock period is $\frac{1}{1.6\ \text{GHz}} = 0.625$ ns, and each instruction takes 5 cycles, so one instruction takes $5 \times 0.625 = 3.125$ ns. For 100 instructions the total time is

$100 \times 3.125 = 312.5$ ns.

Step 3: Time for the pipelined design.

Clock period here is $\frac{1}{1.2\ \text{GHz}} \approx 0.8333$ ns. Out of the 100 instructions, 70 finish in 1 cycle and 30 face a stall of 2 extra cycles, so they take 3 cycles each. Total cycles needed:

$70 \times 1 + 30 \times 3 = 70 + 90 = 160$ cycles.

Total time:

$160 \times 0.8333 \approx 133.33$ ns.

Step 4: Divide the two times.

$$\text{Speed-up} = \frac{312.5}{133.33} \approx 2.34$$

Working with actual instruction counts instead of rates gives the same result as before, which confirms the answer.

$$\boxed{2.34}$$

Quick Tip: Find execution time per instruction for both designs using time = CPI divided by clock frequency, remembering the pipelined CPI must include the extra stall cycles for 30% of instructions, then take the ratio of the two times.

58. Consider a new TCP connection between a sender and a receiver. The receiver advertised window is constant at 48 KB, the maximum segment size (MSS) is 2 KB, and the slow start threshold for TCP congestion control is 16 KB. Assume that there are no timeouts or duplicate acknowledgements. The number of rounds of transmission required for the congestion control algorithm of the TCP connection to reach the congestion avoidance phase is _____.

Note: $1K = 2^{10}$

Correct Answer: 4

SOLUTION:

Step 1: Write cwnd as a formula.
In slow start, after n rounds the window size is 2^{n-1} MSS (since it starts at 1 MSS in round 1 and doubles each round after that).

Step 2: Turn ssthresh into MSS units.
MSS is 2 KB, so ssthresh of 16 KB equals $\frac{16}{2} = 8$ MSS.

Step 3: Solve for n.
We need the smallest n such that $2^{n-1} \geq 8$. Since $8 = 2^3$, we need $n - 1 = 3$, so $n = 4$.

Step 4: Confirm with the numbers.
$n = 4$ gives $2^3 = 8$ MSS $= 16$ KB, matching ssthresh exactly. The advertised window of 48 KB (24 MSS) is never a limiting factor since the window hits ssthresh well before that size.

Step 5: State the switch point.

Once cwnd equals ssthresh at round 4, the connection leaves slow start and enters congestion avoidance from the next round onward.

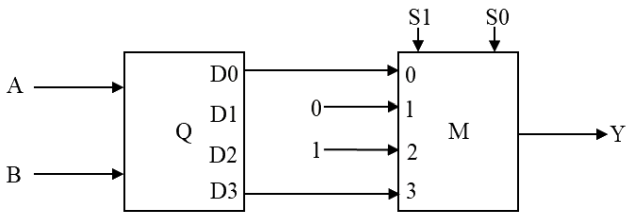
4

Quick Tip: In slow start, cwnd doubles every round starting from 1 MSS. Find how many doublings it takes for cwnd to reach the slow start threshold of 16 KB (8 MSS).

• • •

59. Consider the digital circuit shown below with two input lines A and B, two select lines S0 and S1, and an output line Y. The blocks Q and M represent an active high 2:4 decoder and a 4-to-1 multiplexer, respectively. Out of 16 possible input combinations, the number of combinations that produce Y=1 is _____.

Note: One input combination is an instance of [A B S1 S0].



Correct Answer: 6

SOLUTION:

Step 1: Understand what the mux is really doing.

The select pair $S1S0$ just chooses one of four sources for Y: source 00 is $D0$ from the decoder, source 01 is hardwired low, source 10 is

hardwired high, and source 11 is $D3$ from the decoder. So the circuit behaves like a switch with four settings.

Step 2: Handle the two hardwired settings first, since they do not depend on A, B at all.
When $S1S0 = 01$, Y is stuck at 0 no matter what A and B are, contributing 0 winning rows. When $S1S0 = 10$, Y is stuck at 1 no matter what A and B are, contributing all 4 rows, since there are 4 combinations of A and B.

Step 3: Handle the two decoder-driven settings.
When $S1S0 = 00$, Y copies $D0$, which the decoder makes 1 only for the single input pair $A = 0, B = 0$. That is 1 winning row out of 4.
When $S1S0 = 11$, Y copies $D3$, which is 1 only for $A = 1, B = 1$. That is again 1 winning row out of 4.

Step 4: Add every winning row across all four settings of S1S0.
0 (from 01) + 4 (from 10) + 1 (from 00) + 1 (from 11) = 6

Step 5: Sanity check against the total.
There are 16 total rows (4 values of A,B times 4 values of S1,S0), and only 6 of them make $Y = 1$, a bit more than a third, matching the mix of one constant-1 branch and two single-hit decoder branches.

6

Quick Tip: Write the decoder outputs D0=A'B' and D3=AB, note that mux inputs 1 and 2 are hardwired to 0 and 1, then count, separately for each S1S0 setting, how many of the 4 A,B combinations give Y=1.

• • •

60. Consider the following ANSI-C program.

```
#include <stdio.h>

int main(){
    int *ptr, a, b, c;
    a=5; b=11; c=20;
    ptr=&a; *ptr=c; ptr=&c;
    a=*(&b); c=*ptr-a;
    printf("%d",c);
    return(0);
}
```

The output of this program is _____.

Note: Assume that the program compiles and runs successfully.

Correct Answer: 9

SOLUTION:

Step 1: Make a small table and fill in the starting values.
$a = 5,\ b = 11,\ c = 20,\ ptr =$ undefined

Step 2: Update the table line by line.
After $ptr = \&a$: ptr points at a. Values unchanged: $a = 5, b = 11, c = 20$.
After $*ptr = c$: the memory that ptr points to (which is a) gets the value of c, so $a = 20$. Table now: $a = 20, b = 11, c = 20$.
After $ptr = \&c$: ptr now points at c. Values unchanged: $a = 20, b = 11, c$

$= 20$.

Step 3: Continue with the next statement.
After $a = *(\&b)$: dereferencing the address of b just returns b's own value, 11, so $a = 11$. Table now: $a = 11, b = 11, c = 20$.

Step 4: Apply the last statement carefully.
$c = *ptr - a$. Since ptr points at c, $*ptr$ reads c's value at that moment, which is still 20 (the assignment has not happened yet). We subtract the current a, which is 11. So
$c = 20 - 11 = 9$

Step 5: Read off the printed value.
printf prints c, and the final table shows $a = 11,\ b = 11,\ c = 9$.

$\boxed{9}$

Quick Tip: Track what ptr points to after each reassignment, and remember that in the statement "c=*ptr-a", the old value of c (before this line runs) is what gets read through *ptr.

61. Consider the following ANSI-C function.

```
int func(int start, int end){
    int length=end+1-start;
    if((length<1)||(start<0)||(end<0)){ return(0); }
    if(length%3==0){
        return(func(start+1, end));
    } else if(length%3==1){
        return(1+func(start, end-1));
    } else {
        return(func(start+2, end));
    }
}
```

The maximum possible value that can be returned from this function is _____.

Note: Ignore syntax errors (if any) in the function.

Correct Answer: 1

SOLUTION:

Step 1: Try a few small lengths by hand and watch the pattern.
Let $L = end + 1 - start$ be the length of the range on each call. Take a starting point where $start = 0$ so we can just vary end.
$L = 0$ or negative: returns 0 straight away (base case).
$L = 1$ (so $end = 0$): $1 \bmod 3 = 1$, so this call returns $1 + func(0, -1)$.
The inner call has $end = -1$, which is negative, so it returns 0. Total: 1.
$L = 2$ (so $end = 1$): $2 \bmod 3 = 2$, so this call returns $func(2, 1)$, whose length is 0, so it returns 0. Total: 0.
$L = 3$ (so $end = 2$): $3 \bmod 3 = 0$, so this call returns $func(1, 2)$, a new

call with length 2, which from above returns 0. Total: 0.
$L = 4$ (so $end = 3$): $4 \bmod 3 = 1$, returns $1 + func(0, 2)$. The inner call has length 3, which from above returns 0. Total: 1.

Step 2: Spot the rule from these trials.
Only lengths with remainder 1 mod 3 ever produce a nonzero result, and even then the answer is always exactly 1, never more, because every call that follows a "+1" step lands on a length with remainder 0 mod 3, and lengths with remainder 0 or 2 never trigger another "+1" (they only feed into each other, as seen in the $L = 2$ and $L = 3$ trials above, both of which returned 0).

Step 3: Confirm no larger value is possible.
Since remainder 1 can only appear once before permanently switching into the 0/2 cycle that adds nothing further, the running total can never exceed 1, for any choice of start and end.

Step 4: State the maximum.
The largest value this function can ever return is

$$\boxed{1}$$

Quick Tip: Track the recursion by the value of length modulo 3: a "+1" is only added when length is 1 mod 3, and after that step the remainder can only cycle between 0 and 2, so it can never hit 1 again.

• • •

62. The determinant of a 4×4 matrix A is 3. The value of the determinant of $2A$ is _____.

Correct Answer: 48

SOLUTION:

Step 1: Think about what multiplying a matrix by 2 does to each row.
When we form $2A$, every single entry of A gets doubled. That means every row of the matrix is scaled by 2, one row at a time.

Step 2: Recall how row scaling affects the determinant.
A basic rule of determinants says that scaling one row of a matrix by a constant c scales the whole determinant by that same c. If we scale a second row by c as well, the determinant is multiplied by c again, and so on for every row that gets scaled.

Step 3: Apply this rule four times, once for each row.
Matrix A has 4 rows, and each one is being scaled by 2 when we form $2A$. So we multiply the determinant by 2 a total of four times, once per row:

$$\det(2A) = 2 \times 2 \times 2 \times 2 \times \det(A) = 2^4 \det(A)$$

Step 4: Plug in the given determinant.
We are told $\det(A) = 3$. So

$$\det(2A) = 16 \times 3$$

Step 5: Work out the multiplication.

$$16 \times 3 = 48$$

Step 6: State the result.
The determinant of $2A$ works out to 48, since scaling every row of a 4×4 matrix by 2 scales its determinant by $2^4 = 16$.

$$\boxed{48}$$

Quick Tip: For an $n \times n$ matrix, $\det(kA) = k^n \det(A)$.

• • •

63. Suppose an unbiased coin is tossed 6 times. Each coin toss is independent of all previous coin tosses. Let E_1 be the event that among the second, fourth, and sixth coin tosses, there are at least two heads. Let E_2 be the event that among the first, second, third, and fifth coin tosses, there are equal number of heads and tails.

The conditional probability $P(E_1 \mid E_2)$ is equal to _____.
(rounded off to one decimal place)

Correct Answer: 0.5

SOLUTION:

Step 1: Count total equally likely outcomes.
Six fair coin tosses give $2^6 = 64$ equally likely outcomes in total. Instead of multiplying probabilities, this route counts outcomes directly, which gives a different way to reach the same answer.

Step 2: Restate what each event needs.

E_1 needs at least 2 heads among tosses $2, 4, 6$. E_2 needs exactly 2 heads and 2 tails among tosses $1, 2, 3, 5$. Toss 2 is shared by both events, while tosses $1, 3, 5$ belong only to E_2 and tosses $4, 6$ belong only to E_1.

Step 3: Count outcomes satisfying E_2.
Among the four tosses $1, 2, 3, 5$, we need exactly 2 heads out of 4, which can happen in $\binom{4}{2} = 6$ ways. Tosses 4 and 6 are free to be anything, giving 4 combinations. So the number of outcomes satisfying E_2 is

$$6 \times 4 = 24$$

out of 64, giving $P(E_2) = 24/64 = 3/8$, matching the direct binomial count.

Step 4: Split the 6 arrangements of E_2 by whether toss 2 is a head.
Out of the $\binom{4}{2} = 6$ ways to place 2 heads among tosses $1, 2, 3, 5$: in $\binom{3}{1} = 3$ of them toss 2 is a head (with exactly 1 of the remaining tosses $1, 3, 5$ also a head), and in $\binom{3}{2} = 3$ of them toss 2 is a tail (with exactly 2 of $1, 3, 5$ heads).

Step 5: Count how tosses $4, 6$ must fall for E_1 to hold, in each branch.
If toss 2 is a head (the 3 arrangements from Step 4), E_1 needs at least 1 more head among tosses $4, 6$. Out of the 4 combinations of (X_4, X_6), only TT fails, so 3 combinations work.
If toss 2 is a tail (the other 3 arrangements), E_1 needs both toss 4 and toss 6 to be heads, so only 1 combination out of 4 works.

Step 6: Multiply within each branch and add.
Head branch: 3 arrangements of $1, 2, 3, 5$ times 3 working combinations of $4, 6$ gives 9 outcomes.

Tail branch: 3 arrangements of 1, 2, 3, 5 times 1 working combination of 4, 6 gives 3 outcomes.
Total outcomes satisfying both E_1 and E_2:

$$9 + 3 = 12$$

Step 7: Form the conditional probability as a ratio of counts.

$$P(E_1 \mid E_2) = \frac{\text{outcomes in } E_1 \cap E_2}{\text{outcomes in } E_2} = \frac{12}{24} = \frac{1}{2}$$

Step 8: State the rounded value.
This is 0.5 when rounded to one decimal place, matching the probability based calculation.

$$\boxed{0.5}$$

Quick Tip: Condition on the outcome of the second toss, since it is shared by both events, then combine the two branches using the independence of the remaining tosses.

• • •

64. Consider a function $f : (0, 1) \to \{0, 1\}$ defined as follows.

For a real number $r \in (0, 1)$, $f(r) = 1$ if the second digit after the decimal point in r is one of the four digits 2, 3, 6 and 7. Otherwise, $f(r)$ is equal to 0.

The number of points in $(0, 1)$ at which f is discontinuous is _____.

Correct Answer: 40

SOLUTION:

Step 1: Pin down what makes f jump.
Only the second decimal digit d_2 decides $f(r)$: it is 1 when $d_2 \in \{2, 3, 6, 7\}$ and 0 otherwise. A jump in f can only happen at a point where d_2 changes value, and those change points are exactly the multiples of 0.01, since each strip $[k/100, (k + 1)/100)$ keeps d_2 fixed.

Step 2: List which single digit changes in d_2 flip f's value.
Go through $d_2 = 0$ up to $d_2 = 9$ and mark where the "is it in $\{2, 3, 6, 7\}$" status flips: $0 \to 1$ no flip, $1 \to 2$ flip, $2 \to 3$ no flip, $3 \to 4$ flip, $4 \to 5$ no flip, $5 \to 6$ flip, $6 \to 7$ no flip, $7 \to 8$ flip, $8 \to 9$ no flip, $9 \to 0$ (wrap into next d_1) no flip.
So exactly 4 of the 10 possible digit steps flip f, namely $1 \to 2$, $3 \to 4$, $5 \to 6$, and $7 \to 8$.

Step 3: Translate each flip into an actual point in $(0, 1)$.
For a fixed first digit d_1, these four flips happen where d_2 crosses from 1 to 2, from 3 to 4, from 5 to 6, and from 7 to 8. In terms of r, these boundary points are

$$r = 0.d_1 2, \quad 0.d_1 4, \quad 0.d_1 6, \quad 0.d_1 8$$

that is, $r = (10d_1 + 2)/100, (10d_1 + 4)/100, (10d_1 + 6)/100, (10d_1 + 8)/100$. Each is a genuine discontinuity since f takes one value just to the left and the other value from that point onward.

Step 4: Count how many first digits are possible.
d_1 can be any of $0, 1, 2, \ldots, 9$, so there are 10 possible values of d_1.

Step 5: Multiply digits by flips per digit.
Each of the 10 values of d_1 contributes exactly 4 discontinuity points from Step 3, and no two different values of d_1 produce the same point, since the value of d_1 fixes the hundredths position of the point. So the total count is

$$10 \times 4 = 40$$

Step 6: Confirm the boundaries between first digits add nothing extra.
At points like $r = 0.10, 0.20, \ldots, 0.90$, the second digit wraps from 9 (outside the set) to 0 (also outside the set), so f stays at 0 across these points. These are not discontinuities, so they add nothing to the count.

Step 7: State the total.
Adding the 40 genuine flip points and the 0 extra points at the decade boundaries gives 40 points of discontinuity in total.

$$\boxed{40}$$

Quick Tip: The function changes value only where the second decimal digit changes; check each digit transition from 0 through 9 to see which ones cross the boundary of the set {2,3,6,7}.

• • •

65. It is necessary to design a link-layer protocol between two hosts that are directly connected over a lossless link of length 3000 kilometers. Assume that the link bandwidth is 10^8 bits per second and that the propagation delay in the link is 5 nanoseconds per meter. Every transmitted data byte is assigned a unique sequence number.

Let N be the minimum number of bits needed for the sequence number field in the protocol header such that
i. the sequence numbers do not wrap around before 60 seconds, and
ii. the maximum utilization of the link is achieved.

The value of N is _____.

Correct Answer: 30

SOLUTION:

Step 1: Find the round trip propagation delay.
The cable stretches 3000 km, which is 3×10^6 meters. At 5 ns of delay for every meter, one pass across the link takes

$$T_p = 3 \times 10^6 \times 5 \text{ ns} = 1.5 \times 10^7 \text{ ns} = 15 \text{ ms}$$

A signal and its acknowledgement together cross the link twice, so

$$RTT = 2 \times 15 \text{ ms} = 30 \text{ ms}$$

Step 2: Start from the no wrap in 60 seconds requirement, since sending continues at full speed.

Because the second condition asks for maximum utilization, the sender is pushing bits onto the wire at the full link rate of 10^8 bits per second without any gaps. Each byte gets one sequence number, and 8 bits make one byte, so the byte rate is

$$\frac{10^8}{8} = 1.25 \times 10^7 \text{ bytes per second}$$

Step 3: Multiply the byte rate by 60 seconds.

$$1.25 \times 10^7 \times 60 = 7.5 \times 10^8 \text{ bytes}$$

This is the number of unique sequence numbers consumed inside one minute of full speed sending. The sequence number space, of size 2^N, has to be large enough to hand out a fresh number to every one of these bytes without looping back to an old value.

Step 4: Find the smallest N that covers 7.5×10^8 values.

Try $N = 29$: $2^{29} = 536{,}870{,}912$, which is less than 7.5×10^8, so it falls short.

Try $N = 30$: $2^{30} = 1{,}073{,}741{,}824$, which comfortably covers 7.5×10^8.

So this condition alone forces

$$N \geq 30$$

Step 5: Now check the maximum utilization requirement on its own.
To keep the pipe full, the sender needs enough bytes in flight, sent but not yet acknowledged, to cover the full round trip time. The number of bits that fit in one RTT at full rate is

$$10^8 \times 0.03 \text{ s} = 3 \times 10^6 \text{ bits}$$

Dividing by 8 to get bytes,

$$\frac{3 \times 10^6}{8} = 375{,}000 \text{ bytes}$$

So the sequence number space must at least cover 375,000 values for the window to be usable, which needs $2^N \geq 375{,}000$, and since $2^{19} = 524{,}288$ already clears this while $2^{18} = 262{,}144$ does not, this condition only needs

$$N \geq 19$$

Step 6: Take whichever requirement is stricter.
The field has to satisfy both requirements together, so the answer is the larger of the two lower bounds found in Step 4 and Step 5. Since 30 is bigger than 19, the no wrap around condition is the one that actually decides the answer.

Step 7: State the final bit count.

$$\boxed{N = 30}$$

Quick Tip: Find the bytes needed to fill the bandwidth-delay product for full utilization, and separately find the bytes sent in 60 seconds at full bandwidth; the sequence number field must be large enough to cover the larger of the two.