基本情報技術者試験 科目B 擬似言語プログラミング 演習問題集

本資料は、基本情報技術者試験の科目Bで出題されるアルゴリズムとプログラミング分野の対策を目的とした、オリジナルの演習問題集です。情報処理推進機構(IPA)が公開する擬似言語の記述形式に準拠し、実際の試験で問われる思考力と問題解決能力を養うために設計されています 。
各問題は、2次元配列、ビット演算、2分木、単方向リスト、オブジェクト指向といった頻出分野を網羅しており、難易度は実際の試験レベルに合わせて調整されています。まず自力で問題を解き、その後、詳細な解説を読むことで、理解を深めることを推奨します。

問**1**：**2**次元配列の対角線要素の和（難易度：**2**）

N×N の正方行列（整数型の2次元配列）matrix を受け取り、その主対角線（左上から右下）と副対角線（右上から左下）に含まれる全要素の合計値を返す関数 calcDiagonalSum を考える。ここで、配列の要素番号は1から始まり、N は3以上の奇数であるとする。行列の中央に位置する要素のように、主対角線と副対角線の両方に含まれる要素は、合計値を計算する際に一度だけ加算するものとする。
次のプログラム中の[a]に入れる正しい答えを、解答群の中から選べ。

プログラム

```
整数型: calcDiagonalSum(整数型の二次元配列: matrix, 整数型: N)
  整数型: sum ← 0
  整数型: i

  // 主対角線と副対角線の要素を加算
  for (i を 1 から N まで 1 ずつ増やす)
    sum ← sum + matrix[i, i]
    // 副対角線の要素を加算するが、中央の要素は二重に加算しない
    if (i ≠ (N + 1) / 2)
      sum ← sum + matrix[i, [ a ]]
    endif
  endfor

  return sum
```

解答群
ア. N−i
イ. N−i+1
ウ. N
エ. i+1

問**2**：オブジェクト指向における在庫管理（難易度：**3**）

商品情報を管理するクラス Product がある。このクラスは、商品名 name（文字列型）、価格 price（整数型）、在庫数 stock（整数型）をメンバ変数として持つ。コンストラクタ Product(name, price, stock) によって、インスタンス生成時にこれらの値が初期化される。
Product オブジェクトの配列 products と、最低在庫数のしきい値 threshold を受け取り、在庫数が threshold 以上である商品の在庫価値（価格 × 在庫数）の合計を返す手続 checkStockValue を作成する。
次のプログラム中の[a]に入れる正しい答えを、解答群の中から選べ。

プログラム

```
クラス Product
  文字列型: name
  整数型: price
  整数型: stock

  手続: Product(文字列型: n, 整数型: p, 整数型: s)
    this.name ← n
    this.price ← p
    this.stock ← s
  end手続
エンドクラス

整数型: checkStockValue(Productの配列: products, 整数型: threshold)
  整数型: totalValue ← 0
  Product: p

  for (p に products の要素を順に代入する)
    if (p.stock ≧ threshold)
      [ a ]
    endif
  endfor

  return totalValue
```

解答群

ア. totalValue ← p.price * p.stock
イ. totalValue ← totalValue + p.price
ウ. totalValue ← totalValue + p.stock
エ. totalValue ← totalValue + p.price * p.stock

問**3**：単方向リスト内のデータ検索（難易度：**3**）

各ノードが整数型のデータ data と次のノードへのポインタ next を持つ単方向リストがある。リストの先頭ノード head と探索する整数値 value を受け取り、リスト内に value が存在すれば true を、存在しなければ false を返す関数 findNode を考える。
次のプログラム中の[a]に入れる正しい答えを、解答群の中から選べ。

プログラム

```
クラス Node
    整数型: data
    Node: next
エンドクラス

論理型: findNode(Node: head, 整数型: value)
    Node: current ← head

    while ([ a ])
        if (current.data = value)
            return true
        endif
        current ← current.next
    endwhile

    return false
```

解答群

ア. current.next ≠ null
イ. current.data ≠ value
ウ. true
エ. current ≠ null
オ. current ≠ null and current.data ≠ value

問**4**：ビット演算による権限フラグのチェック（難易度：**4**）

あるシステムでは、ユーザの権限を8ビットの整数で管理している。各ビットは特定の権限に対応しており、ビットが1であればその権限があり、0であれば権限がないことを示す。

- ビット0（最下位ビット）：読み取り(Read)
- ビット1：書き込み(Write)
- ビット2：実行(Execute)

ユーザの権限を表す整数 permissions を受け取り、そのユーザが実行権限を持っているかどうかを判定する関数 canExecute を作成する。他の権限の有無にかかわらず、実行権限のビットが立っていれば true を返す。
次のプログラム中の[a]に入れる正しい答えを、解答群の中から選べ。

プログラム

```
論理型: canExecute(整数型: permissions)
  // 実行権限はビット2に対応する。ビット2のみが1の整数は4である。
  if ([ a ])
    return true
  else
    return false
  endif
```

解答群

ア. (permissions or 4) = 4

イ. permissions = 4

ウ. (permissions and 4) ≠ 0

エ. (permissions xor 4) = 0

問**5**：**2**分探索木の深さの計算（難易度：**4**）

各ノードが整数型のキー key、左の子へのポインタ left、右の子へのポインタ right を持つ2分木の根（ルートノード）root を受け取り、その木の深さ（高さ）を返す再帰関数 getDepth を作成する。木の深さは、根から最も遠い葉までの経路上にあるノードの数と定義する。空の木（root が null）の深さは0とし、根ノードのみからなる木の深さは1とする。
関数 max(a, b) は、a と b のうち大きい方の値を返す。
次のプログラム中の[a]に入れる正しい答えを、解答群の中から選べ。

プログラム

```
クラス TreeNode
    整数型: key
    TreeNode: left
    TreeNode: right
エンドクラス

整数型: getDepth(TreeNode: node)
    if (node = null)
        return 0
    endif

    整数型: leftDepth ← getDepth(node.left)
    整数型: rightDepth ← getDepth(node.right)

    return [ a ]
```

解答群

ア. max(leftDepth, rightDepth)
イ. 1 + leftDepth + rightDepth
ウ. 1 + min(leftDepth, rightDepth)
エ. 1 + max(leftDepth, rightDepth)

問6：2次元配列におけるパターン認識（難易度：4）

0と1から成る M×N の2次元配列 board が与えられる。この配列内に存在する「L字形」のパターンの総数を数える関数 countLShapes を作成したい。ここで「L字形」とは、2×2 の領域において、4つのセルのうちちょうど1つが0であり、他の3つが1であるパターンと定義する。配列の要素番号は1から始まる。
次のプログラム中の[a]、[b]に入れる正しい答えの組合せを、解答群の中から選べ。

プログラム

```
整数型: countLShapes(整数型の二次元配列: board, 整数型: M, 整数型: N)
  整数型: count ← 0
  整数型: i, j, sum

  for (i を 1 から M - 1 まで 1 ずつ増やす)
    for (j を 1 から N - 1 まで 1 ずつ増やす)
      // 2x2領域の要素の合計を計算
      sum ← board[i, j] + board[i, j+1] + board[i+1, j] + board[i+1, j+1]

      // 合計値がL字形の条件を満たすか判定
      if ([ a ] = [ b ])
        count ← count + 1
      endif
    endfor
  endfor

  return count
```

解答群

	a	b
ア	sum	1
イ	sum	2
ウ	sum	3
エ	board[i, j]	1
オ	board[i, j]	3
カ	count	3

問**7**：オブジェクト指向を用いた売上集計（難易度：**4**）

売上情報を表すクラス Sale があり、商品ID productId（整数型）と販売数量 quantity（整数型）をメンバ変数として持つ。Sale オブジェクトの配列 salesData を受け取り、商品IDごとに販売数量を集計する手続 aggregateSales を考える。
この手続は、集計結果を格納する2次元配列 summary を返す。summary の各行は `` という形式になる。salesData を順に処理し、summary にまだ存在しない商品IDであれば新しい行を追加し、既に存在する商品IDであればその行の合計数量を更新する。次のプログラム中の[a]に入れる正しい答えを、解答群の中から選べ。

プログラム

```
クラス Sale
   整数型: productId
   整数型: quantity
エンドクラス

整数型の二次元配列: aggregateSales(Saleの配列: salesData)
   整数型の二次元配列: summary ← {} // 空の2次元配列
   整数型: i, j
   論理型: found
   Sale: sale

   for (i を 1 から salesDataの要素数 まで 1 ずつ増やす)
      sale ← salesData[i]
      found ← false

      // summary内に同じproductIdが存在するか検索
      for (j を 1 から summaryの行数 まで 1 ずつ増やす)
         if (summary[j] = sale.productId)
            summary[j] ← summary[j] + sale.quantity
            found ← true
            break // 内側のループを抜ける
         endif
      endfor

      // 見つからなかった場合、新しい行として追加
      if ([ a ])
         // summaryの末尾に新しい行 {sale.productId, sale.quantity} を追加する
         summaryに行を追加する({sale.productId, sale.quantity})
      endif
   endfor

   return summary
```

解答群

ア. found = true
イ. found = false
ウ. summary[j] = sale.productId
エ. j > summaryの行数
オ. found

問**8**：単方向リストへのソート済み挿入（難易度：**5**）

昇順にソートされた単方向リストの先頭ノード head と、リストに挿入する新しいノード newNode を受け取り、ソート順を維持したまま newNode をリストに挿入する手続 insertSorted を作成する。リストの各ノードは整数型のデータ data と次のノードへのポインタ next を持つ。
次のプログラム中の[a]、[b]、[c]に入れる正しい答えの組合せを、解答群の中から選べ。

プログラム

```
手続: insertSorted(参照 Node: head, Node: newNode)
  Node: current, prev

  // ケース1: リストが空、またはnewNodeを先頭に挿入する場合
  if (head = null or head.data ≧ newNode.data)
    newNode.next ← head
    head ← newNode
    return
  endif

  // ケース2: リストの途中または末尾に挿入する場合
  current ← head
  prev ← null
  while ([ a ])
    prev ← current
    current ← current.next
  endwhile

  [ b ]
  [ c ]
```

解答群

	a	b	c
ア	(current ≠ null) and (current.data < newNode.data)	prev.next ← newNode	newNode.next ← current
イ	(current ≠ null) and (current.data < newNode.data)	newNode.next ← current	prev.next ← newNode
ウ	(current.next ≠ null) and (current.data < newNode.data)	prev.next ← newNode	newNode.next ← current
エ	(current ≠ null) and (current.data > newNode.data)	prev.next ← newNode	newNode.next ← current
オ	(current ≠ null) and (current.data < newNode.data)	current.next ← newNode	newNode.next ← prev
カ	current ≠ null	prev.next ← newNode	newNode.next ← current
キ	(current.next ≠ null) and (current.data < newNode.data)	newNode.next ← prev	current.next ← newNode
ク	(current ≠ null) and (current.data < newNode.data)	prev.next ← current	newNode.next ← newNode

問**9**：ビットマスクを用いたデータフィールドの抽出（難易度：**5**）

ネットワークパケットのヘッダが、16ビットの符号なし整数 header として与えられる。このヘッダのビットフィールドは次のように定義されている。

- ビット 15〜12 (4ビット): バージョン (Version)
- ビット 11〜8 (4ビット): タイプ (Type)
- ビット 7〜0 (8ビット): 長さ (Length)

16ビット整数 header から、「タイプ」と「長さ」の値を抽出し、それぞれを返す関数 parseHeader を作成する。抽出した値は、それぞれが独立した整数値として扱われる。例えば、「タイプ」の値は0から15の範囲になる。
次のプログラム中の[a]、[b]に入れる正しい答えの組合せを、解答群の中から選べ。ここで、>> はビット単位の右シフト演算子を表し、0x で始まる数値は16進数表記を表す。

プログラム

```
// 戻り値は {Type, Length} の形式の整数型配列とする
整数型の配列: parseHeader(整数型: header)
    整数型: type
    整数型: length

    // ビット11〜8を抽出してTypeを取得
    type ← [ a ]

    // ビット7〜0を抽出してLengthを取得
    length ← [ b ]

    return {type, length}
```

解答群

	a	b
ア	(header and 0x0F00)	(header and 0x00FF)
イ	(header and 0x0F00) >> 8	(header and 0x00FF)
ウ	(header and 0xF000) >> 12	(header and 0x00FF) >> 8
エ	(header and 0x0F00) >> 8	(header and 0xFF00)
オ	(header and 0x00F0) >> 4	(header and 0x000F)
カ	(header or 0x0F00) >> 8	(header or 0x00FF)
キ	(header and 0x0F00) >> 4	(header and 0x00FF)
ク	(header and 0x0F00)	(header and 0x00FF) >> 8

問**10**：**2**次元配列を用いた迷路の経路探索（難易度：**5**）

0（通路）と1（壁）で構成される2次元配列 maze で表された迷路がある。出発点 (startX, startY) からゴール (endX, endY) までの経路が存在するかどうかを判定する再帰関数 canEscape を考える。移動は上下左右の4方向にのみ可能である。一度訪れた通路のセルは、無限ループを防ぐために値を 2 に変更する。
関数は、経路が存在すれば true を、存在しなければ false を返す。
次のプログラム中の[a]に入れる正しい答えを、解答群の中から選べ。

プログラム

```
論理型: canEscape(参照 整数型の二次元配列: maze, 整数型: x, 整数型: y, 整数型: endX, 整数型: endY)
  // 範囲外、壁、または訪問済みセルの場合は失敗
  if (x < 1 or y < 1 or x > mazeの行数 or y > mazeの列数)
    return false
  endif
  if (maze[x, y] = 1 or maze[x, y] = 2)
    return false
  endif

  // ゴールに到達した場合は成功
  if (x = endX and y = endY)
    return true
  endif

  // 現在地を訪問済みとしてマーク
  maze[x, y] ← 2

  // 上下左右のいずれかの方向に進んでゴールに到達できれば成功
  if ([ a ])
    return true
  else
    // この経路は行き止まりだったので、マークを元に戻す（バックトラック）
    // ※実際のアルゴリズムでは、戻さない場合もあるが、ここでは戻す設計とする
    maze[x, y] ← 0
    return false
  endif
```

解答群
ア. canEscape(maze, x+1, y, endX, endY) and canEscape(maze, x-1, y, endX, endY) and canEscape(maze, x, y+1, endX, endY) and canEscape(maze, x, y-1, endX, endY)
イ. canEscape(maze, x+1, y, endX, endY) or canEscape(maze, x-1, y, endX, endY) or canEscape(maze, x, y+1, endX, endY) or canEscape(maze, x, y-1, endX, endY)
ウ. canEscape(maze, x, y, endX, endY)
エ. not (canEscape(maze, x+1, y, endX, endY) or canEscape(maze, x-1, y, endX, endY))

解答一覧

問題番号	出題分野	難易度	正解
1	2次元配列	2	イ
2	オブジェクト指向	3	エ
3	単方向リスト	3	エ
4	ビット演算	4	ウ
5	2分木	4	エ
6	2次元配列	4	ウ
7	オブジェクト指向, 2次元配列	4	イ
8	単方向リスト	5	ア
9	ビット演算	5	イ
10	2次元配列, 再帰	5	イ

各問題の詳細解説

問**1**：**2**次元配列の対角線要素の和

1. 正解：イ
2. アルゴリズムとロジックの解説
この問題は、N×N の正方行列において、主対角線と副対角線の要素の和を計算するものです。配列のインデックスが1から始まることに注意が必要です。

- 主対角線: matrix[i, i] のように、行番号と列番号が等しい要素です。ループ変数 i をそのまま使って matrix[i, i] とアクセスできます。
- 副対角線: 右上から左下への対角線です。行番号 i が1からNまで増加するとき、列番号はNから1まで減少します。この関係は、行番号 i と列番号 j の和が常に N+1 になること (i + j = N + 1)で表せます。したがって、行番号が i のとき、列番号は j=N−i+1 となります。
- 重複の排除: 問題の条件より、Nは奇数であるため、行列の中央に要素 matrix[(N+1)/2, (N+1)/2] が存在します。この要素は主対角線と副対角線の両方に属します。プログラムでは、主対角線の要素は無条件に加算し、副対角線の要素を加算する際に if (i ≠ (N + 1) / 2) という条件で中央の要素を二重に加算しないようにしています。

3. プログラムのトレース (N=3 の場合)
matrix を例として { {1,2,3}, {4,5,6}, {7,8,9} } とします。中央の要素は matrix の 5 です。

- i = 1:
  - sum ← sum + matrix (sum=1)
  - i ≠ (3+1)/2 (つまり 1 ≠ 2) は真。
  - 副対角線の列インデックスは 3−1+1=3。
  - sum ← sum + matrix (sum=1+3=4)
- i = 2:
  - sum ← sum + matrix (sum=4+5=9)
  - i ≠ (3+1)/2 (つまり 2 ≠ 2) は偽。if文の中は実行されない。
- i = 3:
  - sum ← sum + matrix (sum=9+9=18)
  - i ≠ (3+1)/2 (つまり 3 ≠ 2) は真。
  - 副対角線の列インデックスは 3−3+1=1。
  - sum ← sum + matrix (sum=18+7=25)
    最終的な sum は 25 となります。

4. 不正解の選択肢の分析

- ア **(N−i)**: i=1のとき3-1=2となりmatrixを指します。これは副対角線ではありません。典型的なオフバイワンエラーです。
- ウ **(N)**: iの値に関わらず常に列インデックスがNになります。これは最後の列を指すだけで、副対角線にはなりません。
- エ **(i+1)**: i=1のとき2となりmatrixを指します。これも副対角線ではありません。

問**2**：オブジェクト指向における在庫管理

1. 正解：エ
2. アルゴリズムとロジックの解説
この問題は、Product オブジェクトの配列をループで処理し、特定の条件（在庫数がしきい値以上）を満たすオブジェクトについて、その在庫価値（価格 × 在庫数）を計算し、合計値に累積していくものです。
for ループで各 Product オブジェクト p が取り出されます。if 文で p.stock が threshold 以上かを確認し、条件を満たす場合にのみ処理を行います。
空欄部分では、現在の合計値 totalValue に、今注目している商品 p の在庫価値 p.price * p.stock を加算する必要があります。代入 ← の右辺で現在の totalValue を使用して累積加算を行う totalValue ← totalValue +... という形式が求められます。
3. プログラムのトレース
products が {Product("A", 100, 5), Product("B", 200, 2)} で threshold が 3 の場合。

- totalValue は 0 で初期化。
- **1**回目のループ **(p** は 商品**"A")**:
  - p.stock (5) ≧ threshold (3) は真。
  - totalValue ← totalValue + p.price * p.stock
  - totalValue ← 0 + 100 * 5 となり、totalValue は 500 になる。
- **2**回目のループ **(p** は 商品**"B")**:
  - p.stock (2) ≧ threshold (3) は偽。if 文の中は実行されない。
- ループ終了。totalValue の 500 が返される。

4. 不正解の選択肢の分析

- ア **(totalValue ← p.price * p.stock)**: 累積加算になっていません。これでは、条件を満たす最後の商品の在庫価値だけが totalValue に残り、それまでの計算結果は上書きされてしまいます。
- イ **(totalValue ← totalValue + p.price)**: 在庫数を無視して価格だけを足しています。在庫価値の合計にはなりません。
- ウ **(totalValue ← totalValue + p.stock)**: 価格を無視して在庫数だけを足しています。これも在庫価値の合計にはなりません。

問**3**：単方向リスト内のデータ検索

1. 正解：エ
2. アルゴリズムとロジックの解説
単方向リストを先頭から末尾まで走査（トラバース）する際の基本的なアルゴリズムです。current というポインタ変数を使い、リストの各ノードを順に指していきます。
ループを続ける条件は、「まだ調べるべきノードが残っているか」です。current ポインタがリストの最後のノードの next を指すと、その値は null になります。つまり、current が null になった時点で、リストの末尾を超えたことを意味します。したがって、ループは current が null でない間、続ける必要があります。
3. プログラムのトレース
リストが 10 -> 20 -> 30 -> null で、value が 30 の場合。

- current は先頭ノード (データ 10) を指す。
- ループ**1**回目:
  - current ≠ null は真。
  - current.data (10) = 30 は偽。
  - current ← current.next (current はデータ 20 のノードを指す)。
- ループ**2**回目:
  - current ≠ null は真。
  - current.data (20) = 30 は偽。
  - current ← current.next (current はデータ 30 のノードを指す)。
- ループ**3**回目:
  - current ≠ null は真。
  - current.data (30) = 30 は真。true を返して関数終了。

もし value が 40 (存在しない) の場合:
-...ループ3回目の後、current は null になる。

- ループ**4**回目:
  - current ≠ null は偽。ループが終了する。
- return false が実行される。

4. 不正解の選択肢の分析

- ア **(current.next ≠ null)**: これは「current が最後のノードではない間」という意味になります。この条件では、最後のノードに到達した時点でループが終了してしまい、最後のノードのデータがチェックされません。これは典型的なバグです。
- イ **(current.data ≠ value)**: ループの終了条件が「値が見つかったら」になってしまいます。これでは while の後の処理が正しく機能しません。また、if 文の条件と重複しており、ロジックが破綻します。
- ウ **(true)**: 無限ループになります。リストの末尾に到達してもループが止まらず、current が null の状態で current.data にアクセスしようとしてエラー（ヌルポインタアクセス）を引き起こします。
- オ **(current ≠ null and current.data ≠ value)**: これはループを続ける条件としては正しいように見えますが、この擬似言語の構成では、ループ内で if 文を使って値の一致を判定し、return するのが一般的です。この条件を while に入れると、ループを抜けた後の処理（return false）が、値が見つかった場合にも実行される可能性があり、プログラムの構造として冗長です。

問**4**：ビット演算による権限フラグのチェック

1. 正解：ウ
2. アルゴリズムとロジックの解説
特定のビットが立っている（1である）かどうかを調べるには、ビットマスクと論理積（and）演算を使用するのが定石です。

- 実行権限: ビット2に対応します。2進数では ...00000100 となり、10進数では 4 です。この 4 がビットマスクになります。
- 論理積 **(and)**: permissions の値と、マスク 4 の間で and 演算を行います。and 演算は、両方のビットが1の場合にのみ、結果のビットが1になります。
- 判定: permissions のビット2が1であれば、permissions and 4 の結果は ...00000100 (つまり 4) となります。ビット2が0であれば、結果は ...00000000 (つまり 0) となります。したがって、and 演算の結果が 0 でない (≠ 0) ことを確認すれば、元の permissions でビット2が立っていたことが分かります。

3. プログラムのトレース

- ケース**1: permissions = 13 (2**進数**: 1101)** の場合
  - 1101 (13) and 0100 (4) を計算します。
  - 結果は 0100 (4) となります。
  - (4) ≠ 0 は真。true を返します。正しい動作です。
- ケース**2: permissions = 9 (2**進数**: 1001)** の場合
  - 1001 (9) and 0100 (4) を計算します。
  - 結果は 0000 (0) となります。
  - (0) ≠ 0 は偽。false を返します。正しい動作です。

4. 不正解の選択肢の分析

- ア **((permissions or 4) = 4)**: 論理和（or）はどちらかのビットが1なら1になるため、permissions が 0 の場合を除き、permissions or 4 の結果は 4 以外にもなり得ます（例: 1 or 4 は 5）。この条件は「実行権限のみがあり、他の権限が一切ない」場合しか正しく判定できません。
- イ **(permissions = 4)**: permissions の値が厳密に 4 であること、つまり「実行権限のみがあり、他の権限が一切ない」場合しか true になりません。問題の要件（他の権限の有無にかかわらず）を満たしません。
- エ **((permissions xor 4) = 0)**: 排他的論理和（xor）は、ビットが異なる場合に1になります。xor の結果が 0 になるのは、二つの値が完全に一致する場合です。したがって、これもイと同様に permissions が厳密に 4 の場合しか true になりません。

問**5**：**2**分探索木の深さの計算

1. 正解：エ

2. アルゴリズムとロジックの解説

木の深さを計算する問題は、再帰を用いると非常に簡潔に記述できます。アルゴリズムの考え方は以下の通りです。

1. ベースケース: もし現在のノード node が null なら、そこには木が存在しないので深さは 0 です。これが再帰の停止条件となります。
2. 再帰ステップ: もしノードが存在する場合、そのノードを根とする部分木の深さは、以下のようになります。
    - 左の子を根とする部分木の深さ（leftDepth）を再帰的に計算する。
    - 右の子を根とする部分木の深さ（rightDepth）を再帰的に計算する。
    - 木の深さは「根から最も遠い葉まで」なので、左と右の深さのうち、より大きい方 (max(leftDepth, rightDepth)) を選択します。
    - 最後に、現在のノード自身の分として 1 を加えます。
      このロジックを数式で表すと 1 + max(leftDepth, rightDepth) となります。

3. プログラムのトレース

簡単な木で考えます。根が10、左の子が5、右の子が20の木。

1. getDepth(10) が呼ばれる。
2. leftDepth ← getDepth(5) が呼ばれる。
    - getDepth(5) の中で getDepth(null) が呼ばれ 0 が返る。leftDepth は 0。
    - getDepth(5) の中で getDepth(null) が呼ばれ 0 が返る。rightDepth は 0。
    - getDepth(5) は 1 + max(0, 0) を計算し 1 を返す。
3. rightDepth ← getDepth(20) が呼ばれる。
    - getDepth(20) も同様に 1 を返す。
4. getDepth(10) は 1 + max(leftDepth, rightDepth)、つまり 1 + max(1, 1) を計算し、2 を返す。木の深さは2であり、正しい結果です。

4. 不正解の選択肢の分析

- ア **(max(leftDepth, rightDepth))**: 現在のノードの深さ（1）を足し忘れています。これでは、根から葉までのノード数ではなく、辺の数になってしまい、定義とずれます。結果は常に1だけ小さくなります。
- イ **(1 + leftDepth + rightDepth)**: 左右の深さを足し合わせています。これは木の幅に関連するような値であり、深さの定義とは全く異なります。
- ウ **(1 + min(leftDepth, rightDepth))**: 深い方ではなく浅い方の部分木の深さを選んでいます。これは「根から最も近い葉までの距離」を計算することになり、問題で問われている「深さ」の定義とは異なります。

問6：2次元配列におけるパターン認識

1. 正解：ウ

2. アルゴリズムとロジックの解説

この問題は、2×2 の領域を走査し、特定のパターンを数え上げるものです。

- パターン定義:「L字形」は、2×2 の領域に 1 が3つ、0 が1つ存在するパターンです。
- 判定方法: プログラムでは、2×2 領域内の4つの要素の値を合計しています。配列の要素は 0 か 1 なので、この合計値 sum は、領域内の 1 の個数と等しくなります。
- 条件式: L字形の定義から、1 の個数は3個です。したがって、sum の値が 3 になるかどうかをチェックすれば、その 2×2 領域がL字形であるかを判定できます。
  よって、if 文の条件は sum = 3 となります。空欄 a には sum が、空欄 b には 3 が入ります。

3. プログラムのトレース

board の一部が {{1,1}, {0,1}} である 2×2 領域を考えます。

- i と j がこの領域の左上を指しているとします。
- sum ← board[i, j] + board[i, j+1] + board[i+1, j] + board[i+1, j+1]
- sum ← 1 + 1 + 0 + 1 となり、sum は 3 になります。
- if (sum = 3) は真となり、count がインクリメントされます。

4. 不正解の選択肢の分析

- ア **(sum = 1)**: これは「逆L字形」（1が1つ、0が3つ）を数えることになります。
- イ **(sum = 2)**: これは 1 が2つあるパターン（直線、対角線など）を数えることになります。
- エ, オ, カ: if 文の左辺が sum ではなく board[i, j] や count になっています。board[i, j] は 2×2 領域の一要素に過ぎず、count は結果を格納する変数なので、これらを条件判定に使うのは論理的に誤りです。

問**7**：オブジェクト指向を用いた売上集計

1. 正解：イ
2. アルゴリズムとロジックの解説
このアルゴリズムは、売上データ salesData を一つずつ処理し、集計配列 summary を更新していくものです。各売上データ sale について、summary に同じ productId が既に存在するかどうかを調べる必要があります。

- フラグ変数 **found**: この変数は、「summary 内に sale.productId が見つかったかどうか」を記録するために使われます。ループの開始前に false で初期化されます。
- 内側のループ: summary を走査し、productId が一致する行が見つかれば、数量を加算して found を true に設定し、break でループを抜けます。
- 外側のループの続き: 内側のループが終わった後、found の値を確認します。
  - もし found が true のままなら、既に対応する行があり更新済みなので、何もしません。
  - もし found が false のままなら、それは summary 内に sale.productId が見つからなかったことを意味します。この場合に限り、新しい行として summary に追加する必要があります。
    したがって、空欄には found = false という条件が入ります。

3. プログラムのトレース
salesData が [{id:101, qty:2}, {id:102, qty:5}, {id:101, qty:3}] の場合。

- summary は {}。
- **i=1 (sale={id:101, qty:2})**:
  - found は false。内側ループは summary が空なので実行されない。
  - if (found = false) は真。summary に {101, 2} を追加。summary は {{101, 2}} に。
- **i=2 (sale={id:102, qty:5})**:
  - found は false。内側ループで 102 は見つからない。
  - if (found = false) は真。summary に {102, 5} を追加。summary は {{101, 2}, {102, 5}} に。
- **i=3 (sale={id:101, qty:3})**:
  - found は false。内側ループで j=1 のとき summary = 101 が一致。
  - summary は 2+3=5 に更新される。found は true に。break。
  - if (found = false) は偽。何もしない。
- 最終的に summary は {{101, 5}, {102, 5}} となる。

4. 不正解の選択肢の分析

- ア **(found = true)**: productId が見つかった場合に新しい行を追加するロジックになり、同じ商品IDが複数行登録されてしまいます。
- ウ **(summary[j] = sale.productId)**: 変数 j は内側のループのスコープ内のものであり、この場所では使えません。また、ロジックとしても意味を成しません。
- エ **(j > summary**の行数**)**: これも変数 j のスコープの問題があります。また、この条件が真になるのは、break せずにループを完走した場合ですが、フラグ変数を使う方が意図が明確で一般的です。
- オ **(found)**: 多くの言語では found は found = true と等価です。アと同じ誤りです。

問**8**：単方向リストへのソート済み挿入

1. 正解：ア
2. アルゴリズムとロジックの解説
ソート済みリストへの挿入はポインタ操作が鍵となります。アルゴリズムは、挿入すべき正しい位置を見つけその前後のノードの next ポインタを適切につなぎ替えることで実現されます。

1. 挿入位置の探索 **(while** ループ**)**: prev (前)と current (現在)の2つのポインタを使ってリストを走査します。prev は current の一歩後ろを追います。ループは、「current がリストの末尾に達しておらず、かつ current のデータが newNode のデータより小さい間」続行します。この条件 (current ≠ null) and (current.data < newNode.data) により、newNode を挿入すべき位置の直前でループが停止します。ループ停止時、prev は挿入位置の前のノードを、current は挿入位置の後ろのノードを指しています。
2. ポインタのつなぎ替え:
    - [b] prev.next ← newNode: まず、前のノード (prev) の next ポインタを、新しいノード (newNode) に向けます。これにより、リストの前半部分と newNode が接続されます。
    - [c] newNode.next ← current: 次に、新しいノード (newNode) の next ポインタを、現在のノード (current) に向けます。これにより、newNode とリストの後半部分が接続されます。
      この2つの操作の順序は重要です。もし逆 (c を先に) 行うと、prev から current へのリンクが失われる前に newNode を差し込むことができなくなります。

3. プログラムのトレース
リストが 10 -> 30 で、newNode (データ 20) を挿入する場合。

- head は 10、newNode.data は 20。先頭挿入の if は偽。
- current は 10、prev は null でスタート。
- **while**ループ:
    - (current(10) ≠ null) and (current.data(10) < 20) は真。
    - prev は 10 を指す。current は 30 を指す。
    - (current(30) ≠ null) and (current.data(30) < 20) は偽。ループ終了。
- ポインタ操作:
    - prev はノード 10、current はノード 30 を指している。
    - [b] prev.next ← newNode: ノード 10 の next が newNode (20) を指すようになる。(10 -> 20)
    - [c] newNode.next ← current: newNode (20) の next がノード 30 を指すようになる。(20 -> 30)
- 結果、リストは 10 -> 20 -> 30 となり正しい。

4. 不正解の選択肢の分析

- イ: b と c の順序が逆です。prev.next ← newNode を実行する前に newNode.next ← current を実行しても問題ありませんが、この選択肢では b と c の内容自体が入れ替わっており、ポインタのつなぎ替えが正しく行われません。
- ウ: while の条件が current.next ≠ null になっています。これでは最後のノードが比較対象から漏れてしまい、リストの末尾に挿入する場合に失敗します。
- エ: while の条件の不等号が > です。これでは昇順リストで正しく挿入位置を探せません。
- オ, カ, キ, ク: ポインタの代入が論理的に破綻しています。例えば、current.next ← newNode は prev を使っていないため、リストのつながりが切れてしまいます。

問**9**：ビットマスクを用いたデータフィールドの抽出

1. 正解：イ
2. アルゴリズムとロジックの解説

ビットフィールドからのデータ抽出は、①ビットマスクを用いた論理積 (and) で目的のビット群を分離し、②右シフト (>>) で値を最下位ビットに揃える、という2段階で行います。

- **[a] Type** の抽出 **(ビット 11〜8)**:
  1. マスク: Type フィールドに対応するビットのみが 1 となるマスク 0x0F00 (2進数: 0000 1111 0000 0000) を使います。
  2. **and**演算: header and 0x0F00 を計算すると、ビット11〜8以外のビットはすべて 0 になり、Type の情報だけが残ります。
  3. 右シフト: この時点での値は、まだビット8の位置にあります。これを通常の整数値（0〜15）にするため、8ビット右にシフト (>> 8) します。
  - したがって、式は (header and 0x0F00) >> 8 となります。
- **[b] Length** の抽出 **(ビット 7〜0)**:
  1. マスク: Length フィールドに対応するマスク 0x00FF (2進数: 0000 0000 1111 1111) を使います。
  2. **and**演算: header and 0x00FF を計算すると、Length の情報だけが残ります。
  3. 右シフト: Length フィールドは既に最下位ビット（ビット0）から始まっているため、シフトは不要です。
  - したがって、式は header and 0x00FF となります。

3. プログラムのトレース

header が 0xA531 (2進数: 1010 0101 0011 0001) の場合。

- **Type** の抽出:
  - header and 0x0F00: 1010 0101 0011 0001 and 0000 1111 0000 0000 = 0000 0101 0000 0000 (0x0500)
  - (0x0500) >> 8: 0000 0000 0000 0101 (0x0005)。type は 5 となる。
- **Length** の抽出:
  - header and 0x00FF: 1010 0101 0011 0001 and 0000 0000 1111 1111 = 0000 0000 0011 0001 (0x0031)。length は 49 となる。

4. 不正解の選択肢の分析

- ア: Type の抽出で右シフトを忘れています。これでは type の値が 5 ではなく 0x0500 (1280) になってしまいます。
- ウ: Type の抽出に Version 用のマスク (0xF000) を使っており、Length の抽出で不要な右シフトをしています。
- エ: Length の抽出に誤ったマスク (0xFF00) を使っています。
- カ: and ではなく or を使っています。or ではビットを分離できません。
- キ: Type のシフト量が 4 になっており、誤りです。
- ク: Type のシフト忘れと、Length の不要なシフトという二重の誤りがあります。

問**10**：**2**次元配列を用いた迷路の経路探索

1. 正解：イ
2. アルゴリズムとロジックの解説
この問題は、深さ優先探索（DFS: Depth-First Search）と呼ばれるアルゴリズムを再帰で実装したものです。考え方は「行けるところまで進んでみて、行き止まりだったら一つ前に戻って別の道を試す」というものです。

- ベースケース:
  - 迷路の範囲外、壁(1)、訪問済み(2)のいずれかであれば、その道は失敗 (false)。
  - ゴールに到達すれば、成功 (true)。
- 再帰ステップ:
  - 現在地を訪問済み (2) にマークする。
  - 上、下、左、右の4方向に対して再帰的に canEscape を呼び出す。
  - ここで重要なのは、**4**方向のうち、いずれか一つでも成功 **(true)** すれば、ゴールまでの経路が見つかったことになるという点です。複数の経路を見つける必要はなく、一つの経路の存在を確認できれば十分です。
  - この「いずれか一つでも」という論理は、論理和（or）で表現されます。
- バックトラック: if 文が偽、つまり4方向すべてが失敗に終わった場合、この地点は行き止まりです。maze[x, y] ← 0 で訪問マークを元に戻し（これをバックトラックと呼びます）、呼び出し元に失敗 (false) を伝えます。

したがって、空欄には4つの再帰呼び出しを or でつないだ式が入ります。
3. プログラムのトレース
簡単な迷路を考えます。
{ {0,0,G}, {W,0,W} } (G=ゴール, W=壁)
canEscape(start) が呼ばれると、例えば右に進む canEscape(right) を試します。
canEscape(right) はさらに右に進む canEscape(goal) を呼び出し、これが true を返します。or で繋がれているため、canEscape(right) は true を返し、最終的に canEscape(start) も true を返します。もし右が壁なら、次に下を試す、というように探索が進みます。
4. 不正解の選択肢の分析

- ア: 論理積 (and) を使っています。これは「上下左右すべての方向に進んだ結果、すべて成功である」ことを要求するもので、迷路探索のロジックとして成り立ちません。ゴールは一方向にしかないので、この条件が真になることはあり得ません。
- ウ: 自分自身を再度呼び出しています。引数が変わらないため、無限再帰に陥り、スタックオーバーフローを引き起こします。
- エ: not を使っており、論理が逆転しています。「どの方向にも進めない」場合に true を返すような意味合いになり、ゴールを探すロジックとは正反対です。

引用文献

1. 基本情報_科目B公開問題(アルゴリズムとプログラミング)_20240705.docx