

```
def compute(abcde):  
    fghi = True  
    for j in range(2, (abcde // 2) + 1):  
        if abcde % j == 0:  
            fghi = False  
    return fghi
```

1

Figure 1: primeLN

```
def compute(abcde, fgh):  
    ijklmn = 0  
    opqrst = len(abcde) - 1  
    while ijklmn <= opqrst:  
        u = (ijklmn + opqrst) // 2  
        if fgh < abcde[u]:  
            opqrst = u - 1  
        elif fgh > abcde[u]:  
            ijklmn = u + 1  
        else:  
            return u  
    return -1
```

2

Figure 2: binarySearchLN

```

def compute(abcdef: str) -> int:
    if abcdef == "0":
        return 0
    if abcdef == "1":
        return 1
    if abcdef[-1] == "0":
        return 2 * compute(abcdef[:-1])
    if abcdef[-1] == "1":
        return 1 + 2 * compute(abcdef[:-1])
    return -1

```

4

Figure 3: binaryToDecimalLT

```

def compute(abcd: str, efghijk: list[str]) -> int:
    lmnopqrstuv: int = 0
    for w in range(len(abcd)):
        for x in range(len(efghijk)):
            if abcd[w] == efghijk[x]:
                lmnopqrstuv += 1
    return lmnopqrstuv

```

2

Figure 4: countLettersLT

```
def compute(abcde):  
    for f in range(len(abcde)):  
        for g in range(f, 0, -1):  
            if abcde[g-1] > abcde[g]:  
                abcde[g-1], abcde[g] = abcde[g], abcde[g-1]  
    return abcde
```

2

Figure 5: bubbleSortLN

```
def compute(abcdefg):  
    hijklm = [0.0] * len(abcdefg)  
    for m in range(len(abcdefg)):  
        if abcdefg[m] == 0:  
            hijklm[m] = 0.0  
            continue  
        if abcdefg[m] < 0:  
            hijklm[m] = math.sqrt(-1 * abcdefg[m])  
        else:  
            hijklm[m] = math.sqrt(abcdefg[m])  
    return str(hijklm)
```

1

Figure 6: squareRootLN

```
def compute(a, b):  
    if b == 0:  
        return 1  
    if b == 1:  
        return a  
    return a * compute(a, b - 1)
```

1

Figure 7: powerLN

```
def compute(abcdef: int) -> int:  
    if abcdef == 0:  
        return 0  
    return (abcdef % 10) + compute(abcdef // 10)
```

2

Figure 8: crossSumLT

Hat mir am
besten gefallen.

```
def compute(a, b, c, d):
    if a > b:
        b, a = a, b
    if c > d:
        d, c = c, d
    if a > c:
        c, a = a, c
    if b > d:
        d, b = b, d
    if b > c:
        c, b = b, c
    return [a, b, c, d]
```

1

Figure 9: unrolledSortLN

```
def compute(abcdef, ghijk, lmn):
    opqrst = 0
    uvwx = abcdef.split(" ")
    for y in range(len(uvwx)):
        zab = int(uvwx[y])
        cdefg = (zab >= ghijk and zab <= lmn)
        if cdefg:
            opqrst += 1
    return opqrst
```

2

Figure 10: countIntegerIntervallLN

```
def compute(abcde: list[int], f: int) -> int:
    for g in range(len(abcde)):
        if abcde[g] == f:
            return g
    return -1
```

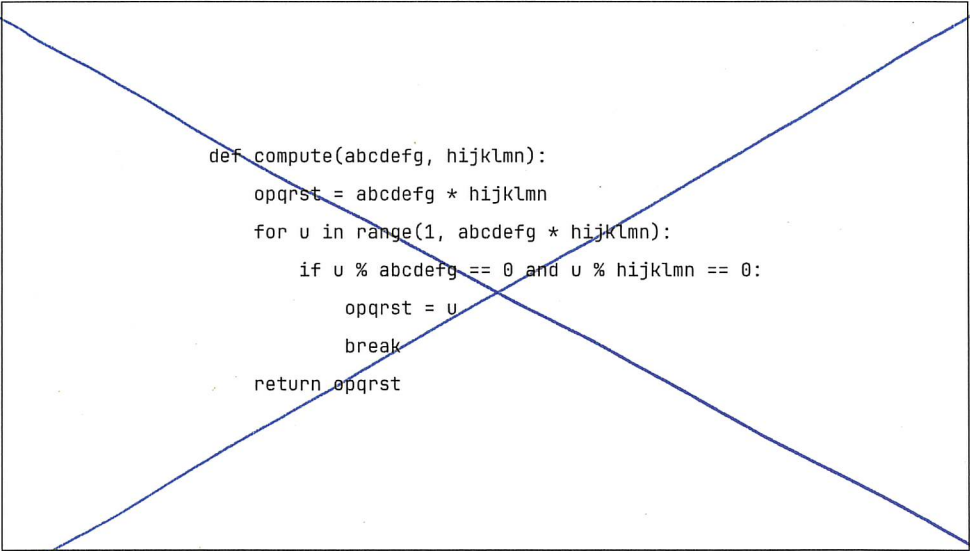
2

Figure 11: linearSearchLT

```
def compute(abcdefgh: str) -> str:
    ijkln: list[int] = []
    opqrs: list[str] = abcdefgh.split()
    for t in range(len(opqrs)):
        if t > 0:
            ijkln.append(" ")
            ijkln.append(opqrs[t][0].upper() + opqrs[t][1:])
    return "".join(ijkln)
```

5

Figure 12: capitalizeFirstLetterLT



```
def compute(abcdefg, hijklmn):  
    opqrst = abcdefg * hijklmn  
    for u in range(1, abcdefg * hijklmn):  
        if u % abcdefg == 0 and u % hijklmn == 0:  
            opqrst = u  
            break  
    return opqrst
```

Figure 13: leastCommonMultipleLN

```
def compute(abcdefg: list[int]) -> float:  
    hijklmn: int = 0  
    opq: int = 0  
    while hijklmn < len(abcdefg):  
        opq = opq + abcdefg[hijklmn]  
        hijklmn = hijklmn + 1  
    rstuvw: float = opq / hijklmn  
    return rstuvw
```

Figure 14: arrayAverageLT

```

def compute(abcde):
    f = ""
    g = ""
    for i in range(len(abcde) - 1, -1, -1):
        f = abcde[i] + f
        g = g + abcde[i]
    return f + g

```

Figure 15: forwardBackwardLN

```

def compute(abcd: str) -> bool:
    efghij: bool = True
    for k in range(0, len(abcd) // 2):
        l: int = len(abcd) - 1 - k
        if abcd[k] != abcd[l]:
            efghij = False
            break
    return efghij

```

Figure 16: palindromeLT


```

def compute(abcd, efghijklm):
    for n in range(len(abcd)):
        for o in range(len(efghijklm)):
            if n + o >= len(abcd):
                break
            if abcd[n + o] != efghijklm[o]:
                break
        else:
            if o == len(efghijklm) - 1:
                return True
    return False

```

Figure 17: containsSubstringLN

```

def compute(abcdefghijk, lmonpqrstuvx):
    yzabcde = 0
    if len(abcdefghijk) < len(lmonpqrstuvx):
        fghijklmnopqrstuvwxyz = len(abcdefghijk)
    else:
        fghijklmnopqrstuvwxyz = len(lmonpqrstuvx)
    for y in range(fghijklmnopqrstuvwxyz):
        if abcdefghijk[y] == lmonpqrstuvx[y]:
            yzabcde += 1
    return yzabcde

```

Figure 18: commonCharsLN

```
def compute(abcde):
    if abcde == 1:
        return 1
    return compute(abcde - 1) * abcde
```

Figure 19: factorialLN

```
def compute(abcdefgh: str) -> bool:
    ijklm: list[str] = []
    for n in abcdefgh:
        if n == '(':
            ijklm.append('(')
        elif n == '{':
            ijklm.append('{')
        elif n == '[':
            ijklm.append '[')
        elif len(ijklm) == 0 or ijklm.pop() != n:
            return False
    return len(ijklm) == 0
```

Figure 20: validParenthesesLT