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

Figure 1: factorialLN

```
def compute(abcdef):
    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
```

Figure 2: binaryToDecimalLN

```
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
```

6

Figure 3: palindromeLT

```
def compute(abcde: list[int]) -> list[int]:
    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
```

3

Figure 4: bubbleSortLT

29206

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

1.

↳ ich weiß nicht mehr ob ich f oder g ausgebe.

~~Aber ich~~

Deswegen habe ich leider 3 argument + (f)

anstelle von 2 (g)

Aber war einfach nur dummer fehler

Figure 5: linearSearchLT

```
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]
```

2

Figure 6: unrolledSortLN

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

3

Figure 7: leastCommonMultipleLT

```
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 8: countLettersLT

2926

```
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)
```

2

Figure 9: squareRootLN

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

5

Figure 10: capitalizeFirstLetterLT liegt ab bei mir, falsch nachgedacht

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

5

da war ich überfragt. ich habe den code nicht in der Zeit verstanden.

Figure 11: binarySearchLT

```
def compute(abcdefgh):
    ijklm = []
    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
```

2

Figure 12: validParenthesesLN

29206

```
def compute(abcd: str, efghijklm: str) -> bool:
    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 13: containsSubstringLT

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

Figure 14: countIntegerIntervalLT

```
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(abcde: int) -> bool:
    fghi: bool = True
    for j in range(2, (abcde // 2) + 1):
        if abcde % j == 0:
            fghi = False
    return fghi
```

Figure 16: primeLT

```
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 17: commonCharsLN

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

Figure 18: crossSumLT

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

Figure 19: arrayAverageLN

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

Figure 20: powerLN