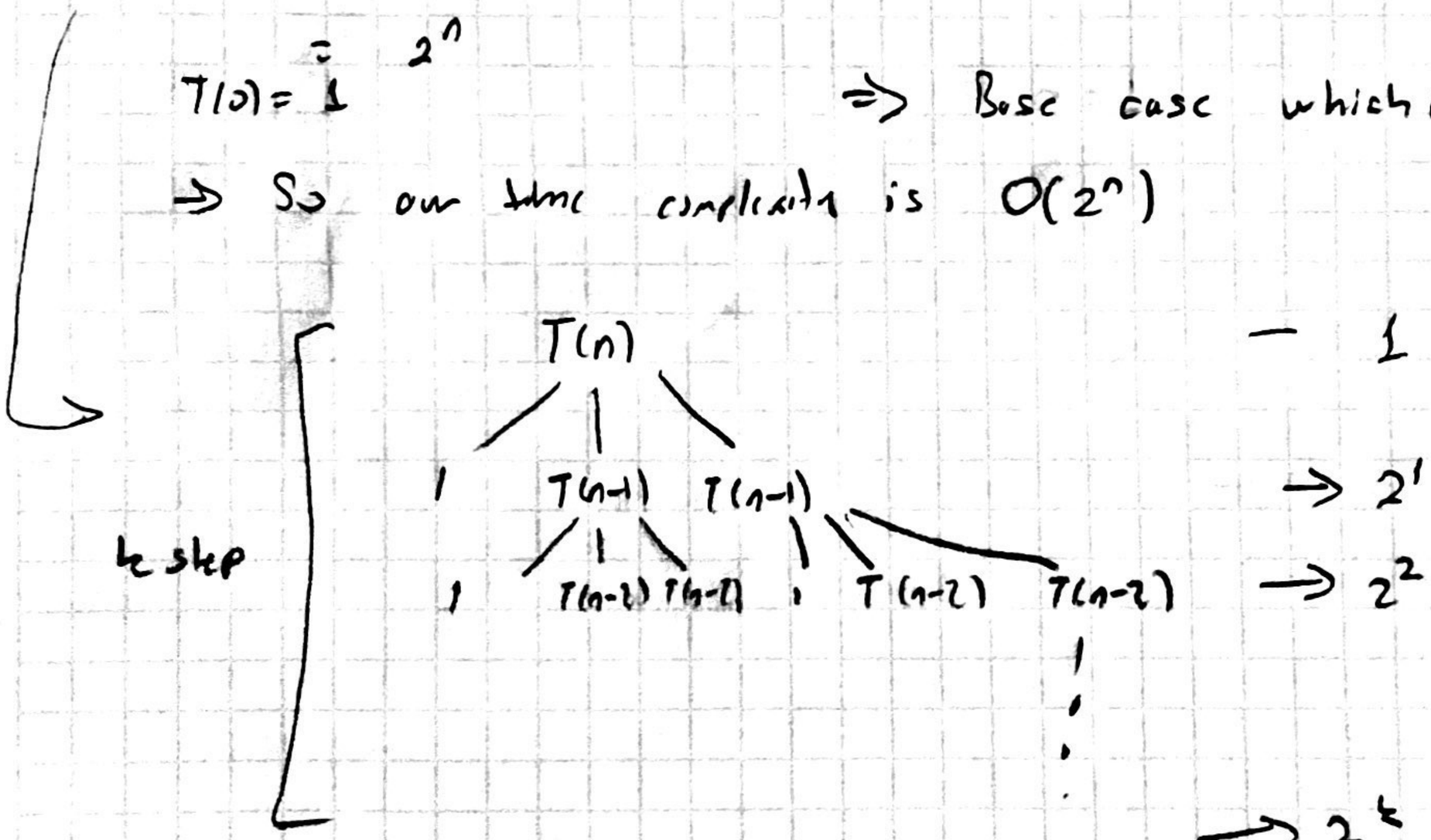
Q1-) ⇒ Let's say that our time complexity is $T(n)$. In code we have recursive functions and each call iterates $n-1$ times so:

$$T(n) = 2 * T(n-1) + 1$$
$$= 2 * (2 * T(n-2) + 1) + 1$$
$$\vdots$$
$$= 2^n$$

⇒ 2 comes from recursive call because of every iteration we call higher discount function twice.

$T(0) = 1$ ⇒ Base case which is put a

⇒ So our time complexity is $O(2^n)$



$$1 + 2^1 + 2^2 + \dots + 2^k = 2^{k+1} - 1$$

assume $n = k$

$$\Rightarrow O(2^{k+1} - 1) \Rightarrow O(2^n)$$

Q2-) n: users n: processes n: processors

=> Short the code (pseudo.)

- Worst-Case Time Complexity : $O(n^3)$

=> This situation occurs when finding the optimal schedule needs exploring the all possible assignment. Also each assignment needs looping in [users, processes, processors] which is contain every n iterations.

=> So, overall complexity of $n^1 n^1 n = n^3$

- Best-Time Time Complexity : $O(n^2)$

=> The best case scenario would be finding the optimal schedule early in the search. So we want to find in 1 iteration of loop. In this case, the algorithm would only need to iterate through a subset of users, process, and processors before returning the optimal solution, which is $O(n^2)$.

- Average-Time Complexity : $O(n^3)$

=> Because of exhaustive nature of the algorithm, estimating the average case complexity is depending on distribution of costs. and the probability of finding the optimal solution early. It is so hard. The average case complexity is $O(n^3)$

Q3) In pseudocode

→ energyCost function to calculate energy cost of moving from the current position to assemble part.

→ Worst-Case Time Complexity

↳ The algorithm explores all possible combinations of assembling the parts, leading to a factorial time complexity. If there are n parts assembled, the worst case will be $O(n!)$

↳ Mathematical solution

T(n) represents time complexity of algorithm.

→ in each step it has n parts, and after step it considers (n-1) Pcs. We say that remaining part in each.

$$T(n) = n * T(n-1) = n * (n-1) * T(n-2) = n!$$

→ Best-Case Time Complexity

↳ In this scenario, the optimal sequence might be found early in the search. However, because this algorithm exhaustively explores all possibilities, the best-case time complexity remains the same as the worst-case, $O(n!)$

→ Average-Case Time Complexity

↳ Algorithm checks all possibilities so it is also $O(n!)$

Q4-1 in pseudo code:

- minCoins
  - ↳ Recursive function that finds minCount
- coins
  - ↳ coin array

⇒ Function updates min with minimum of its value with recursive call.

- Explanation
  - Check if target is equal zero that is mean that no coins needed. Base Case
  - Every iteration, check if coin less than or equal from target
    - ↳ call recursively function again with reduced value and update count value. (target - coin)
  - After updating count value check count is smaller than minCount update minimum count value and return it.

- Time Complexity analysis
  - Best-Case Complexity O(1)
    - ↳ In the best case if target equal zero return value. O(1)
  - Worst-Case Complexity $O(n^t)$

    n = coins.length t = target

    ↳ Function explores all possible combinations for the remaining part. At each calls (iteration) of recursive algorithm considers n coins and makes recursive call for each coin. Also depth is considering as target so

$$T(n) \in O(n^t)$$

Q5-) Algorithm solves problem by using recursive calls with dividing array first and second halfes it even calls until low equals high or high +1 equals low.

recurrence relation ⇒ $T(n) = 2T(n/2) + 2$, $T(2) = 1$, $T(1) = 0$

↳ 2 recursive calls, ↳ divide two ↳ base case

$T(n) = 2T(n/2) + 2$

$T(n) = 2\,[2T(n/4) + 2) + 2$

$= 2\,(2\,[2T(n/4) + 2) + 2) + 2$

⋮

$T(n) = 2^k T(n/2^k) + 2 + 2^2 + \dots + 2^k$

$\dots \Rightarrow n = 2^{k+1}/2^k \cdot 2$

$T(n) = \frac{2^{k+1}}{2} + \frac{2^{k+1}}{2^k} + 2 + 2^2 + \dots + 2^k$

$= n \cdot \frac{n}{2} \cdot 2n - 2 \quad 2(n/2 - 1) \quad 2n - 2$

$= \frac{n}{2} + 2n - 2$

$T(n) = \frac{5n}{2} - 2 \Rightarrow T(n) \in O(n)$