1. Solve the following recurrence relations. For parts (a) and (b), give an exact solution. For parts (c) and (d), give an asymptotic one. In both cases, justify your solution.

   (a) $A(n) = A\left(\frac{1}{2}n\right) + n^3$

   Solution: We simply find the work done at every level. At level k we have $(\frac{k}{2^k})^3 = \frac{1}{2^{3k}}k^3$ work. Thus the exact amount of work done is (using the fact that we have a geometric sequence)

   $$\begin{aligned} A(n) &= n^3 + \frac{1}{8}n^3 + \frac{1}{64}n^3 + \cdots + 1 \\ &= n^3 \cdot \frac{1 - 1/(8n^3)}{1 - 1/8} \\ &= \frac{8n^3 - 1}{7} \end{aligned}$$

   ■

   (b) $B(n) = 2B\left(\frac{1}{4}n\right) + \sqrt{n}$

   Solution: We again find the work done at every level. At level k we have $2^k\sqrt{\frac{n}{4^k}} = \sqrt{n}$. Since there are $\log_4 n = \frac{\log n}{2}$ levels, we simply have that $B(n) = \frac{\log n}{2}\sqrt{n}$ ■

   (c) $C(n) = C\left(\frac{1}{3}n\right) + C\left(\frac{2}{3}n\right) + O(n)$

   Solution: We again find the work done at every level. At level k we have

   $$O(n) \cdot \frac{1}{3^k}\sum_{i=0}^{k}\left[\binom{k}{i}2^i\right] = O(n)$$

   work. Since there is $O(\log n)$ number of levels, we conclude an asymptotic bound of $C(n) = O(n) \cdot O(\log n) = O(n \log n)$. ■

   (d) $D(n) = D\left(\frac{1}{15}n\right) + D\left(\frac{1}{10}n\right) + 2D\left(\frac{1}{6}n\right) + \sqrt{n}$

   Solution: We can get a lower and upper bound by using the number of deepest and shallowest leaves, noticing that at leaves we have constant amount of work. The deepest leaf will be at level $k = \log_6 n$, and we can upper bound the number of leaves by $4^k = 4^{\log_6 n} = n^{\log_6 4}$. So we have overestimate $D(n) = O(n^{\log_6 4})$.

   On the other hand, the shallowest leaf will be at level $k = \log_{15} n$, and we can lower bound the number of leaves by $4^k = 4^{\log_{15} n} = n^{\log_{15} 4}$. So we have underestimate $D(n) = \Omega(n^{\log_{15} 4})$. ■

2. Suppose you are given a sorted sequence of distinct integers $a = [a_1, a_2, \ldots, a_n]$ drawn from 1 to m where $n < m$. Give an $O(\lg(n))$ algorithm to find the *smallest* integer $\leq m$ that is not present in a.

Solution: The idea is, since array a is sorted and elements are distinct. So if $a^i = i+1$, all elements up to i were present. If $a^i \neq i+1$, the partial array up to i missed integers. We can define a binary search function to find the smallest missing integer, where a denotes searching array.

```
FindMatch(ℓ, r):
    left, right ← 0, len(a) − 1
    while left ≤ right:
        mid = (left+right)//2
        if a[mid] == mid + 1:
            left = mid + 1
        else:
            right = mid − 1
    return left + 1
```

Binary search algorithm shrink searching window to half in each iteration, so the time complexity is $O(\log n)$. ■

3. Let $\boldsymbol{A}, \boldsymbol{B}$ be two $n \times n$ matrices of real numbers. In your Linear Algebra class, you might've learned the brute-force algorithm to compute $\boldsymbol{AB}$ in $O(n^3)$ time. However, using a similar strategy to Karatasuba's algorithm adapted to matrices, a mathematician named Volker Strassen invented a divide-and-conquer algorithm to perform the multiplication in $O(n^{2.8074})$ time, dutifully named **Strassen's Algorithm**. From now on, assume you're given a black box, Strassen$(\boldsymbol{A}, \boldsymbol{B})$, which computes the matrix product of two matrices $\boldsymbol{A}, \boldsymbol{B} \in \mathbb{R}^{n \times n}$. We assume that multiplication and addition of two real numbers is constant.

   Note: No Linear Algebra knowledge should be used to solve this problem

   (a) Suppose you needed to hand-compute the "power of 2" matrix power $\boldsymbol{A}^{2^k}$ for some matrix $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ and $k \in \mathbb{N}$. Give an efficient algorithm to solve this problem and determine it's time complexity in terms of $m = 2^k$, that is in terms of the power on the matrix $\boldsymbol{A}$.

   Solution: The key idea is that $\boldsymbol{A}^{2^{k+1}} = \boldsymbol{A}^{2^k}\boldsymbol{A}^{2^k}$, from which the recursive relation follows. We call our recursive relation Pow2$(\boldsymbol{A}, k)$ and express it as:

$$\text{Pow2}(\boldsymbol{A}, k) = \begin{cases} \text{Strassen}(\text{Pow2}(\boldsymbol{A}, k-1), \text{Pow2}(\boldsymbol{A}, k-1)) & k > 1 \\ \boldsymbol{A} & k = 1 \end{cases}$$

   The runtime of this algorithm satisfies the following recurrence relation for the variable k:

$$T(k) = T(k-1) + O(n^{2.8074})$$

   Assuming that $T(1) = 1$, we get a runtime of $O(kn^{2.8074})$ after unrolling. Since $k = \log_2(m)$, we get a final runtime of $O(\log_2(m)n^{2.8074})$. ■

   (b) How would you modify this to compute $\boldsymbol{A}^m$ for any $m \in \mathbb{N}$, that is compute any positive integer matrix power? Would this change the time complexity?

   Solution: In this case, we can not assume that m is a power of 2. However, we can still make this recursive by cleverly splitting it up for when m is even and odd. If m is even, we can do $\boldsymbol{A}^m = \boldsymbol{A}^{0.5m}\boldsymbol{A}^{0.5m}$ and if m is odd, we can do $\boldsymbol{A}^m = \boldsymbol{A}\boldsymbol{A}^{0.5(m-1)}\boldsymbol{A}^{0.5(m-1)}$. We call our recursive relation Pow$(\boldsymbol{A}, m)$ such that:

$$\text{Pow}(\boldsymbol{A}, m) = \begin{cases} \text{Strassen}(\text{Pow}(\boldsymbol{A}, \frac{m}{2}), \text{Pow}(\boldsymbol{A}, \frac{m}{2})) & m \equiv 0 \pmod 2 \\ \text{Strassen}(\boldsymbol{A}, \text{Pow}(\boldsymbol{A}, m-1)) & m \equiv 1 \pmod 2 \end{cases}$$

   The analysis is slightly trickier than before. Assume $T(1) = 1$ and notice that if m is odd, then it takes 2 matrix multiplications to get from $\lfloor \frac{m}{2} \rfloor$ to m. This is our worst case, so we can bound the recurrence by:

$$T(m) \leq T(\lfloor \frac{m}{2} \rfloor) + 2O(n^{2.8074})$$

   Again using unrolling we see that we achieve a runtime of $O(\log_2(m)n^{2.8074})$. Thus, the asymptotic performance doesn't change. ■

(c) Say your friend George was able to magically calculate the matrix product AB using 6 matrix multiplications of $\frac{n}{2} \times \frac{n}{2}$ matrices and 37400 additions of $n \times n$ matrices. What's the runtime of his algorithm? (*Note*: It takes $O(n^2)$ to calculate $\boldsymbol{A} + \boldsymbol{B}$).

Solution: The first step in a divide-and-conquer runtime problem is always to formulate a recurrence relation. In this case, we have 6 recursive subproblems of size $\frac{n}{2}$ and $37400\,O(n^2) = O(n^2)$ amount of work at each level. This can be formulated as the following recurrence relation where $c \in \mathbb{R}$ is some positive constant.

$$T(n) = 6T(\frac{n}{2}) + cn^2$$

We assume that $T(1) = 1$ and we can apply Master's Theorem to this recurrence where $a = 6$ and $b = 2$. Doing the math, we find we're in the "leaf-dominated" case and thus our runtime is $O(n^{\log_2(6)}) \approx O(n^{2.585})$. ■

(d) George coded a correct version of the algorithm in Python, but when he benchmarked it against Strassen's algorithm he found his implementation was much slower. Why might that be?

Solution: The overhead of 37400 matrix additions outweighs the improvement he made by saving a single matrix multiplication. Asymptotically, George's solution might be faster, but for the (relatively) small matrices being tested on his computer it's much slower. For reference, Strassen's algorithm uses around 18 additions. ■

4. We are given a array of n steel rods of integer length where the i^{th} piece has length $L[i]$. We seek to cut them so that we end up with k pieces of exactly the same length, in addition to other fragments. And we want this collection of k pieces to be as large as possible.

   For instance, the largest collection ($k = 4$) pieces you can get from the collection: $L = \{10, 6, 5, 3\}$ is 5.

   Give a correct and efficient algorithm that, for a given L and k, returns the maximum possible length of the k equal pieces cut from the initial n sticks.

Solution: First, let l be the length of the pieces after cutting the steel rods

$$L[i] = a_i \times l + b_i \tag{1}$$

which means that we would end up with a_i steel rods with length l and one rod with length $b_i < l$ after cutting i^{th} rod, for a given l.

Next, what are the possible values for l that we should consider?

$$1 \leq l \leq \max(L) \tag{2}$$

How do we check if a given value of l works for us or not? To check this, we can try cutting the rods as in (1) and see if $\sum_{i=1}^{|L|} a_i \geq k$.

```
⟨⟨Check if a given value of l works for L and k.⟩⟩
IsValid(L, k, l):
  n ← 0
  for i ← 1 to |L|
      n ← n + ⌊L[i]/l⌋
  if n ≥ k
      return True
  else
      return False
```

A simple solution is to check every l, starting from 1 to $\max(L)$ and return the largest l that is valid, according to IsValid.

```
FindLargestCutLength(L, k):
  l_max ← 0
  for l ← 1 to max(L)
      if IsValid(L, k, l)
          l_max ← l
  return l_max
```

The complexity of IsValid is $O(|L|)$. For FindLargestCutLength, the for loop iterates from 1 to $\max(L)$. Hence, its complexity is $O(\max(L) \times |L|)$.

Can we make FindLargestCutLength faster? Let's take two values l_i and l_j such that $l_j > l_i$. If IsValid(L, k, l_i) is False, IsValid(L, k, l_j) will also be False. This means that we can use binary search to find the largest valid l.

```
FindLargestCutLength(L, k):
  l_max ← 0
  l_left ← 1
  l_right ← max(L)
  while l_left ≤ l_right
      l_mid ← ⌊(l_left + l_right)/2⌋
      if IsValid(L, k, l_mid)
          l_max ← l_mid
          l_left ← l_mid + 1
      else
          l_right ← l_mid
  return l_max
```

Since we are discarding half the search space in every iteration, the while loop will run $\log(\max(L))$ times. Hence, the complexity of FindLargestCutLength is $O(\log(\max(L)) \times |L|)$.

■