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) $W(n) = W(n-1) + 2\log(n) + 1; W(0) = 0$

   Solution: We obtain an exact closed-form solution for $W(n)$ by unrolling. From $W(n) = (W(n-2) + 2\log(n-1) + 1) + 2\log n + 1 = W(n-2) + 2(\log n + \log(n-1)) + 2$, we observe that

   $$W(n) = W(n-k) + 2(\log n + \log(n-1) + \cdots + \log(n-k+1)) + k$$

   for $1 \le k \le n$. Thus,

   $$W(n) = 2(\log n + \log(n-1) + \cdots + \log 1) + n = 2\log(n!) + n.$$

   ■

   (b) $X(n) = 5X(n-1) + 3; X(1) = 3$

   Solution: We obtain an exact closed-form solution for $X(n)$ by unrolling. From $X(n) = 5(5X(n-2) + 3) + 3 = 5^2 X(n-2) + 5 \cdot 3 + 3$, we observe that $X(n) = 5^k X(n-k) + 3(5^{k-1} + \cdots + 5^1 + 5^0)$ for $1 \le k \le n-1$. Thus,

   $$X(n) = 3(5^{n-1} + 5^{n-2} + \cdots + 5^1 + 1) = \frac{3(5^n - 1)}{5-1} = \frac{3}{4}(5^n - 1).$$

   ■

   (c) $Y(n) = Y(n/2) + 2Y(n/3) + 3Y(n/4) + n^2$

   Solution: We obtain a tight asymptotic bound for $Y(n)$ using a recursion tree. Observe that the sum of node values in the recursion tree for $Y(n)$ for any complete level k is $\left(\frac{95}{144}\right)^k n^2$. Since the sum over the levels is a decreasing geometric series, $Y(n)$ is dominated by the root n^2. This tells us that $Y(n) = O(n^2)$.

   On the other hand, n^2 is a lower bound of $Y(n)$ by definition, giving us $T(n) = \Omega(n^2)$.

   We conclude $Y(n) = \Theta(n^2)$. ■

(d) $Z(n) = Z(n/15) + Z(n/10) + 2Z(n/6) + \sqrt{n}$

Solution: We obtain lower and upper bounds on the asymptotic solution for $Y(n)$ using a recursion tree. We observe that the sum of node values for any complete level k in the recursion tree for $Z(n)$ is $(\sqrt{1/15} + \sqrt{1/6} + 2/\sqrt{6})^k \sqrt{n}$. Since the level-by-level sum is an increasing geometric series, $Z(n)$ is dominated by the sum of the nodes values in the bottom level of its recursion tree.

To obtain an upper bound for $Z(n)$, we overestimate $Z(n)$ by extending the recursion tree down to the level of the deepest leaf. The deepest leaf is at level $\Theta(\log_6 n)$, so we can upper bound the number of leaves in the recursion tree as $O(4^{\log_6 n})$ or equivalently, $O(n^{\log_6 4})$. Since the leaves correspond to the base case, the label on each leaf is $O(1)$. Thus, $Z(n) = O(n^{\log_6 4})$.

To obtain a lower bound, we underestimate $Z(n)$ by extending the tree down to the level of the shallowest leaf. The shallowest leaf is at level $\Theta(\log_{15} n)$, so we can lower bound the number of leaves as $\Omega(4^{\log_{15} n})$ or equivalently, $\Omega(n^{\log_{15} 4})$.

Therefore, we conclude that $Z(n) = \Omega(n^{\log_{15} 4})$ and $Z(n) = O(n^{\log_6 4})$.

On the other hand, we can obtain the tight asymptotic bound by applying the Akra-Bazzi method. The equation $(1/15)^\rho + (1/10)^\rho + 2(1/6)^\rho = 1$ has solution $\rho \approx 0.6596$. We have

$$\int_1^n \frac{f(u)}{u^{\rho+1}} du = \frac{2u^{\frac{1}{2}-p}}{1-2\rho}\Bigg|_{u=1}^{n} = \frac{2n^{\frac{1}{2}-\rho} - 2}{1-2\rho} = \Theta(n^{\frac{1}{2}-\rho}).$$

Therefore, we get

$$Z(n) = \Theta\left(n^\rho \left(1 + \Theta(n^{\frac{1}{2}-\rho})\right)\right)$$

and $Z(n) = \Theta(n^\rho)$. ■

2. Suppose you are given a stack of n pancakes of different sizes. You want to sort the pancakes so that smaller pancakes are on top of larger pancakes. The only operation you can perform is *flip* - insert a spatula under the top k pancakes, for some integer k between 1 and n, and flip them all over.

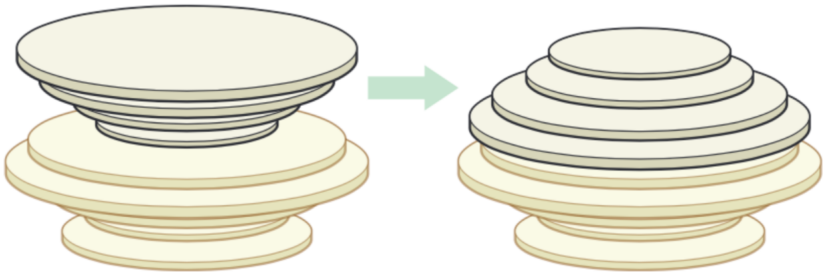

Figure 1.19. Flipping the top four pancakes.

(a) Describe an algorithm to sort an arbitrary stack of n pancakes using $O(n)$ flips. *Exactly* how many flips does your algorithm perform in the worst case? [*Hint: This problem has nothing to do with the Tower of Hanoi.*]

Solution: We can solve this problem with recursion.

In each recursive step, we can place the largest pancake to the bottom and recurse to the stack above it.

We can represent the stack of pancakes with A[1..n], where A[1] represents the pancake at the top and A[n] represents the pancake at the bottom.

Suppose that in each recursive step, we can access and modify the original stack of pancakes. The algorithm PancakeSort(A[1..n], k) goes as follows:

```
if (k > 1)
    i = FindLargestPancake(A[1..n], k)
    flip(A[1..n], i) //Place the largest pancake to the top
    flip(A[1..n], k) Place the largest pancake to the bottom
    PancakeSort(A[1..n], k − 1)
end if
```

To solve the problem, we can call PancakeSort(A[1..n], n).

The algorithm performs $2 * (n-1)$ flips in the worst case. ■

(b) For every positive integer n, describe a stack of n pancakes that requires $\Omega(n)$ flips to sort.

Solution: There are many solutions to this problem. Here is one example.

Given a stack of n pancakes where each pancake has a different size, create a new stack from empty in this order: append the largest pancake, append the smallest pancake, append the second largest pancake, append the second smallest pancake, ...

This stack requires $\Omega(n)$ flips to sort.

■

(c) Now suppose one side of each pancake is burned. Describe an algorithm to sort an arbitrary stack of n pancakes, so that the burned side of every pancake is facing down, using $O(n)$ flips. *Exactly* how many flips does your algorithm perform in the worst case?

Solution: We can solve this problem by modifying the PancakeSort algorithm.

In each recursive step, we want to ensure that the burned side of the largest pancake is facing up before placing the pancake to the bottom.

The algorithm BurnedPancakeSort(A[1..n], k) goes as follows:

```
if (k > 1)
    i = FindLargestPancake(A[1..n], k)
    flip(A[1..n], i) //Place the largest pancake to the top
    if BurnedSideFacingDown(A[1])
            flip(A[1..n], 1)
            Make sure the burned side is facing up
    end if
    flip(A[1..n], k) Place the largest pancake to the bottom
    PancakeSort(A[1..n], k − 1)
end if
```

To solve the problem, we can call BurnedPancakeSort(A[1..n], n).

The algorithm performs $3 * (n-1)$ flips in the worst case. ■

3. Suppose we are given an array $A[1..n]$ of n integers, which could be positive, negative, or zero, sorted in increasing order so that $A[1] \leq A[2] \leq \cdots \leq A[n]$. Suppose we wanted to count the number of times some integer value x occurs in A. Describe an algorithm (as fast as possible) which returns the number of elements containing value x.

Solution: Dumb Approach: We could simply iterate through the array and count the number of times x appears. This would take $O(n)$ time.

Better Approach: First we can use binary search to find an instance of x. Then since A is sorted, All values of x appear next to one-another. Hence, if we find one instance of x, we can interate over the block of x instances and count the size. This will take $O(log(n)+k)$ time where k is the number of array elements containing x. The one issue is that if k is large, i.e. on the order of n, then the runtime reduces to $O(log(n)+k) = O(log(n)+O(n)) = O(n)$.

Best Approach: We can slightly modify binary search to find the leftmost array element that contains x (the left-bound of the array block):

```
FindLeftBound(A[1..n], x, i):
  if A[1] = x
      return i
  else
      if A[n/2] ≥ x
          return FindLeftBound(A[1, ..., n/2], x, i)
      else
          return FindLeftBound(A[n/2 + 1, ..., n], x, i + n/2)
```

i is a variable to keep track of the original position of the sub-array beign currently evaluated. We do the same to find the right bound and subtract the two values from one another to find the number of instances of x. ■

4. Given an arbitrary array $A[1..n]$, describe an algorithm to determine in $O(n)$ time whether A contains more than $n/4$ copies of any value.

Solution: The algorithm is formally described below. We use the fact that the selection problem can be solved in linear time. That is, given an unsorted array A of n values and an index j between 1 and n, we can find the j-th ranked element in A in $O(n)$ time. We denote this black box algorithm as SELECT(A[1..N], j) which returns the value of the j-th ranked element in A. To determine whether an element appears more than $n/4$ times, we select values with rank $n/4$, $2n/4$, and $3n/4$. If an element x appears more than $n/4$ times, it follows that at least one of these selected values is equal to x. Thus, we can scan and count the number of occurrences of each of these selected values.

```
Contains4Duplicates(A[1..N])
    x_1 ← SELECT(A, ⌈N/4⌉)
    x_2 ← SELECT(A, ⌈2N/4⌉)
    x_3 ← SELECT(A, ⌈3N/4⌉)
    for (i ← 1 : 3)
        count ← 0 for (j ← 1 : N)
            if (A[j] = x_i) then count++1
        if (count > N/4) then return True
    return False
```

Since SELECT runs in $O(n)$ time, finding x_1, x_2, and x_3 also takes $O(n)$ time. Looping over the array of length n a total of 3 times takes $O(n)$ time. Thus, this algorithm runs in the required $O(n)$ time.

To prove correctness of the algorithm, we must show that if an element appears more than $n/4$ times, it must be at least one of the selected values with rank $\lceil n/4 \rceil$, $\lceil 2n/4 \rceil$, or $\lceil 3n/4 \rceil$. Assume an element x appears $i > n/4$ times. Then, there must be consecutive ranks $j, ..., j+i-1$ with value x. Without loss of generality, consider the number of values of rank between $\lceil n/4 \rceil$ and $\lceil 2n/4 \rceil$ (excluding the outside values). Since $\lceil n/4 \rceil \geq n/4$ and $\lceil 2n/4 \rceil \leq 2n/4+1$, the maximum number of values is given by $(2n/4+1)-(n/4)-1 = n/4$. Thus, there are at most only $n/4$ spots for more than $n/4$ values. By pigeonhole principle, one of the selected values must be equal to x. ■