1)

The recursive algorithm for solving the Tower of Hanoi problem can be described as follows:

1. Base Case: If there is only one disk, move it from the source peg to the destination peg.
2. Recursive Case: If there are more than one disk, move the n-1 disks from the source peg to the spare peg (using the destination peg as a spare), then move the largest disk from the source peg to the destination peg, and finally move the n-1 disks from the spare peg to the destination peg (using the source peg as a spare).

Here is the implementation of the recursive algorithm in Python:

```
def tower_of_hanoi(n, source, destination, spare):
    if n == 1:
        print("Move disk 1 from", source, "to", destination)
        return
    tower_of_hanoi(n-1, source, spare, destination)
    print("Move disk", n, "from", source, "to", destination)
    tower_of_hanoi(n-1, spare, destination, source)
```

To solve a Tower of Hanoi problem with n disks, we can call the function with the following arguments:

```
tower_of_hanoi(n, "Peg 1", "Peg 3", "Peg 2")
```

The algorithm requires 2^n - 1 moves to solve the problem. Therefore, for n disks, the algorithm will make 2^n - 1 moves. For example, for 3 disks, the algorithm will make 2^3 - 1 = 7 moves.

2)

(a) For the case k = 3, we can modify the recursive algorithm for the classic Tower of Hanoi problem. Let f(n, i, j) be a function that moves n disks from peg i to peg j using the third peg. We can then define the following recursive algorithm:

- If n = 1, move the disk from peg i to peg j.
- If n > 1, first move the top n - 1 disks from peg i to peg 2 (using peg 3), then move the bottom disk from peg i to peg j, and finally move the top n - 1 disks from peg 2 to peg j (using peg 1 and peg 3).

This algorithm makes 2^n - 1 moves.

(b) For the case k = n + 1, we can use a divide-and-conquer approach. Let f(n, i, j) be a function that moves n disks from peg i to peg j using the other k - 2 pegs. We can then define the following recursive algorithm:

- If n ≤ k - 2, use the algorithm from part (a) to move the disks from peg i to peg j.

- If n > k - 2, first move the top k - 2 disks from peg i to peg k - 1 (using pegs j and k), then move the bottom n - (k - 2) disks from peg i to peg j (using pegs k and k - 1), and finally move the top k - 2 disks from peg k - 1 to peg j (using pegs i and k).

This algorithm requires at most O(n^3) moves, since each recursive call involves moving at most n - 1 disks and there are O(n) recursive calls.

(c) For the case k = n + 1, we can improve the algorithm from part (b) by using a more efficient way to move the top k - 2 disks. Let g(n, i, j, k) be a function that moves n disks from peg i to peg j using the other k - 2 pegs, where the top k - 2 disks are initially on peg k. We can then define the following recursive algorithm:

- If n ≤ k - 2, use the algorithm from part (a) to move the disks from peg i to peg j.
- If n > k - 2, first move the top (k - 2)/2 disks from peg i to peg k (using pegs j and k - 1), then move the bottom n - (k - 2)/2 disks from peg i to peg j (using pegs k and k - 1), then move the top (k - 2)/2 disks from peg k to peg j (using pegs i and k - 1), and finally move the remaining (k - 2)/2 disks from peg k to peg j (using pegs i and j).

This algorithm requires at most O(n^2) moves, since each recursive call involves moving at most (k - 2)/2 disks and there are O(log n) recursive calls.

(d) For the case k = √n, we can use a similar approach as in part (c), but with a smaller number of pegs. Let h(n, i, j) be a function that moves n disks from peg i to peg j using the other 2√n - 2 pegs. We can then define the following recursive algorithm:

- If n ≤ 2√n - 2, use the algorithm from part (a) to move the disks from peg i to peg j.
- If n > 2√n - 2, first move the top √n - 1 disks from peg i to peg k (using pegs j and k - 1), then move the bottom n - (√n - 1) disks from peg i to peg j (using pegs k and k - 1), and finally move the top √n - 1 disks from peg k to peg j (using pegs i and k - 1).

This algorithm requires at most O(n^1.5) moves, since each recursive call involves moving at most √n - 1 disks and there are O(√n) recursive calls.

(e) For arbitrary n and k, we can use a similar divide-and-conquer approach as in parts (b) and (c). Let f(n, i, j) be a function that moves n disks from peg i to peg j using the other k - 2 pegs. We can then define the following recursive algorithm:

- If n ≤ k - 2, use the algorithm from part (a) to move the disks from peg i to peg j.
- If n > k - 2, first move the top (k - 2)/2 disks from peg i to peg k (using pegs j and k - 1), then move the bottom n - (k - 2)/2 disks from peg i to peg j (using pegs k and k - 1), then move the top (k - 2)/2 disks from peg k to peg j (using pegs i and k - 1), and finally move the remaining (k - 2)/2 disks from peg k to peg j (using pegs i and j).

The number of moves required by this algorithm depends on the value of k. If k = O(log n), then the number of moves is O(n log n). If k = O(n^ε) for some ε < 1/2, then the number of moves is O(n^{1+ε}). If k = O(n), then the number of moves is O(n^2). Finally, if k = √n, then the number of moves is O(n^{1.5}). Therefore, the smallest value of k that makes the number of moves bounded by a polynomial in n is k = O(n^ε) for some ε < 1/2.

4)

(a) Here's an algorithm to sort an arbitrary stack of n pancakes using O(n) flips:

1. Start with the stack of n pancakes.
2. For i from n to 1, do the following: a. Find the index j of the largest pancake among the first i pancakes. b. If j is not equal to i, flip the top j pancakes to move the largest pancake to the bottom of the first i pancakes. c. Flip the top i pancakes to move the largest pancake to the top of the stack.

At each iteration of the loop, the largest pancake in the remaining unsorted stack is moved to the top of the stack. After n iterations, the stack will be sorted with the smallest pancake at the bottom and the largest pancake at the top. Since each iteration involves at most two flips, the algorithm uses O(n) flips.

(b) For any positive integer n, we can construct a stack of n pancakes that requires Ω(n) flips to sort by arranging the pancakes in descending order of size. In this case, we cannot move any pancake to the top of the stack without first flipping all the pancakes above it, and each flip can only bring the largest pancake down one position. Therefore, we need at least n - 1 flips to move the largest pancake from the bottom to the top of the stack. Since each flip can move at most two pancakes, we need at least Ω(n) flips to sort the stack.

5)

The problem of counting the number of inversions in an n-element array can be solved efficiently in O(n log n) time using the Divide and Conquer approach.

The algorithm works as follows:

1. Divide the array into two halves, and recursively count the number of inversions in each half.
2. Merge the two halves and count the number of split inversions, which are pairs (i, j) where i is in the left half and j is in the right half, and A[i] > A[j].
3. Return the sum of the number of inversions in the left half, the number of inversions in the right half, and the number of split inversions.

Here is the Python code for the algorithm:

```
def count_inversions(arr):
    n = len(arr)
    if n <= 1:
        return 0

    mid = n // 2
    left = arr[:mid]
    right = arr[mid:]
```

```
    # Recursively count the number of inversions in the left and right halves
    inversions = count_inversions(left) + count_inversions(right)

    # Merge the two halves and count the number of split inversions
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            i += 1
        else:
            j += 1
            inversions += len(left) - i

    # Return the total number of inversions
    return inversions
```

The time complexity of the algorithm is O(n log n), as each recursive call takes O(log n) time, and each merge operation takes O(n) time. The space complexity is O(n), as we need to store the two halves of the array during the merge operation.

6)

We can solve this problem using the Divide and Conquer approach. The idea is to split the set of line segments into two halves, recursively count the number of intersections in each half, and then count the number of intersections between the two halves.

Here's how the algorithm works:

1. Sort the set of line segments by their x-coordinates.
2. Divide the set of line segments into two halves.
3. Recursively count the number of intersections in each half.
4. Merge the two halves and count the number of intersections between the two halves.
5. Return the sum of the number of intersections in each half and the number of intersections between the two halves.

To count the number of intersections between the two halves, we can use the following algorithm:

1. Create two pointers, one for each half, and initialize them to point to the first line segment in each half.
2. Compare the two line segments pointed to by the two pointers. If they intersect, increment the count of intersections.

3. Move the pointer that points to the line segment with the lower x-coordinate one step to the right.
4. Repeat steps 2-3 until one of the pointers reaches the end of its half.

Here is the Python code for the algorithm:

```
def count_intersections(segments):
    n = len(segments)
    if n <= 1:
        return 0

    # Sort the segments by their x-coordinates
    segments = sorted(segments, key=lambda x: x[0])

    # Divide the segments into two halves
    mid = n // 2
    left = segments[:mid]
    right = segments[mid:]

    # Recursively count the number of intersections in each half
    left_count = count_intersections(left)
    right_count = count_intersections(right)

    # Count the number of intersections between the two halves
    i = j = count = 0
    while i < len(left) and j < len(right):
        if left[i][1] > right[j][1]:
            count += len(left) - i
            j += 1
        else:
            i += 1

    # Return the total number of intersections
    return left_count + right_count + count
```

The time complexity of the algorithm is O(n log n), as each recursive call takes O(log n) time, and each merge operation takes O(n) time. The space complexity is O(n), as we need to store the two halves of the set of line segments during the merge operation.

7)

To determine whether there exist two elements in S whose sum is exactly x in O(n log n) time, we can sort the elements in S using a standard O(n log n) sorting algorithm, such as merge sort or quicksort. Once we have sorted the elements, we can use a two-pointer approach to search for pairs of elements whose sum is exactly x.

The algorithm can be described as follows:

1. Sort the elements in S using a standard O(n log n) sorting algorithm.
2. Initialize two pointers, **left** and **right**, to the beginning and end of the sorted array, respectively.
3. While **left** < **right**, compare the sum of the elements at indices **left** and **right** to x:

a. If the sum is equal to x, then we have found a pair of elements whose sum is x. Return True.

b. If the sum is less than x, increment **left**.

c. If the sum is greater than x, decrement **right**.

4. If no pair of elements whose sum is x is found, return False.

The time complexity of this algorithm is dominated by the sorting step, which takes O(n log n) time. The two-pointer search takes O(n) time, which is linear and therefore does not affect the overall time complexity. Therefore, the overall time complexity of the algorithm is O(n log n).

8)

One possible algorithm to compute the weighted median of a given weighted set in O(n) time is as follows:

1. Combine the two input arrays S and W into a single array A, where A[i] = (S[i], W[i]) for $1 \leq i \leq n$.
2. Sort the array A in non-decreasing order of value, using any comparison-based sorting algorithm that runs in O(n log n) time, such as quicksort or mergesort. In case of ties, sort the tied elements by increasing weight.
3. Initialize two variables, w_left = 0 and w_right = w(S), where w(S) = w(S[1..n]) is the total weight of the input set.
4. For i from 1 to n, do the following: a. Let x = A[i] = (S[i], W[i]). b. If w_left + x.weight > w_right / 2, return x.value as the weighted median. c. Otherwise, update w_left ← w_left + x.weight and w_right ← w_right - x.weight.

At each iteration of the loop, we check if x.value is the weighted median by computing the total weight of elements with value less than or equal to x.value (w_left + x.weight) and the total weight of elements with value greater than x.value (w_right - x.weight), and comparing them to half of the total weight of the input set. If w_left + x.weight exceeds w_right / 2, then x.value must be the weighted median, since any element with value less than or equal to x.value has a combined weight of at most w_left + x.weight, which is less than half of the total weight of the input set. Otherwise, we update w_left and w_right to include x.weight and continue to the next element.

The algorithm makes n iterations of the loop, each taking O(1) time, so the overall running time is O(n). The sorting step takes O(n log n) time, which is dominated by the linear time complexity of the algorithm. Therefore, the algorithm runs in O(n) time.

9)

(a) One approach to solve this problem is to use the Boyer-Moore majority vote algorithm.

The Boyer-Moore majority vote algorithm is an algorithm for finding the majority element in an array in O(n) time and O(1) space. The majority element is defined as the element that appears more than n/2 times in an array of length n.

The algorithm works by maintaining a count variable that starts at 0 and a candidate variable that starts as null. It then iterates over the array and for each element, it checks if the count is 0. If it is, it sets the candidate to the current element. If it isn't, it checks if the current element is equal to the candidate. If it is, it increments the count. If it isn't, it decrements the count.

At the end of the iteration, the candidate variable will contain a possible majority element. We then need to verify if it appears more than n/4 times in the array. We can do this in a second iteration by counting the number of occurrences of the candidate element in the array. If it appears more than n/4 times, we return true. Otherwise, we return false.

Here is the algorithm:

Initialize a count variable to 0 and a candidate variable to null.

Iterate over the array A from left to right.

For each element A[i], do the following:

a. If the count is 0, set the candidate to A[i].

b. If A[i] is equal to the candidate, increment the count.

c. If A[i] is not equal to the candidate, decrement the count.

Initialize a counter variable to 0.

Iterate over the array A again from left to right.

For each element A[i], if it is equal to the candidate, increment the counter.

If the counter is greater than n/4, return true.

Otherwise, return false.

The time complexity of this algorithm is O(n) since we iterate over the array twice, each time in O(n) time. The space complexity is O(1) since we only use a constant amount of extra space to store the count and candidate variables.

Here's the C code for the algorithm:

```
#include <stdio.h>

int more_than_n_over_4(int A[], int n) {
  int count = 0, candidate = -1;

  // Step 1: Find candidate using Boyer-Moore majority vote algorithm
  for (int i = 0; i < n; i++) {
    if (count == 0) {
      candidate = A[i];
    }
    if (A[i] == candidate) {
      count++;
    } else {
      count--;
    }
  }

  // Step 2: Check if candidate appears more than n/4 times
  int counter = 0;
  for (int i = 0; i < n; i++) {
    if (A[i] == candidate) {
      counter++;
    }
  }
```

```
    if (counter > n/4) {
        return 1;
    } else {
        return 0;
    }
}

int main() {
    int A[] = {1, 2, 3, 4, 5, 5, 5, 5, 6, 7, 8};
    int n = sizeof(A) / sizeof(A[0]);
    if (more_than_n_over_4(A, n)) {
        printf("Array A contains more than n/4 copies of some element\n");
    } else {
        printf("Array A does not contain more than n/4 copies of any element\n");
    }
    return 0;
}
```

The more_than_n_over_4 function takes an array A and its length n as input, and returns 1 if the array contains more than n/4 copies of some element, and 0 otherwise.

In the main function, we initialize an example array A and call the more_than_n_over_4 function on it. The output will depend on the contents of A.

(b) One way to approach this problem is to sort the array A and then iterate through it, counting the number of occurrences of each value. Whenever we encounter a value that has more than k occurrences, we can immediately return True. If we reach the end of the array without finding any such value, we can return False.

Here's the pseudocode for this algorithm:

```
function contains_more_than_k_copies(A, n, k):
    sort(A)
    count = 1
    prev_val = A[1]
```

```
    for i in range(2, n+1):
        if A[i] == prev_val:
            count += 1
            if count > k:
                return True
        else:
            count = 1
            prev_val = A[i]
    return False
```

The sorting step takes O(n log n) time using a typical comparison-based sorting algorithm, such as quicksort or mergesort. The subsequent loop takes O(n) time, since we iterate over each element of the sorted array exactly once. Therefore, the overall running time of the algorithm is O(n log n + n) = O(n log n).

Note that this algorithm does not depend on the precise input values, only on their order. This is because sorting is a purely order-based operation that does not change the number of occurrences of each value. Therefore, this algorithm satisfies the requirement of the problem statement.

10)

To compute the median of an array A[1, . . . , 5] of distinct numbers using at most 6 comparisons, we can use the following algorithm:

We begin by comparing A[1] and A[2]. If A[1] < A[2], we know that A[1] is smaller than or equal to three other elements in the array, and A[2] is smaller than or equal to two other elements in the array. Therefore, the median must be one of A[3], A[4], or A[5]. We can represent this comparison with a binary tree as follows:

```
    A[1] < A[2]?
     /    \
  A[1..2]  A[3..5]
```

If A[1] > A[2], we can reason similarly to conclude that the median must be one of A[1], A[4], or A[5]. This can be represented as follows:

A[1] < A[2]?

**/ **

A[1..2] A[3..5]

**/ **

A[3..4] A[5]

In either case, we have reduced the problem to finding the median of a subarray of three elements. We can do this with one more comparison, by comparing A[3] and A[4]. Depending on the result of this comparison, we can determine the median as follows:

A[1] < A[2]?

**/ **

A[1..2] A[3..5]

**/ **

A[3] < A[4]? A[5]

**/ **

A[3] A[4] Median is A[4]

| |

A[4] A[3] Median is A[4]

Overall, this algorithm requires at most six comparisons, which is the maximum number needed to compare any five distinct elements.

11)

We can use a binary search algorithm to find the peak entry p in O(log n) time. The algorithm works as follows:

1. Initialize two pointers, left and right, to the first and last indices of the array A.
2. While left < right, do the following: a. Calculate the midpoint index, mid = (left + right) // 2. b. If A[mid] < A[mid + 1], then the peak entry must be to the right of mid. Set left = mid + 1. c. Otherwise, the peak entry must be to the left of mid. Set right = mid.

At the end of the loop, left and right will converge to the peak entry p. We can return either left or right.

Here is the implementation of the algorithm in Python:

```
def find_peak_entry(A):
    n = len(A)
    left, right = 0, n-1
```

```
while left < right:
    mid = (left + right) // 2
    if A[mid] < A[mid + 1]:
        left = mid + 1
    else:
        right = mid
return left  # or return right
```

The algorithm makes O(log n) comparisons in the worst case, since at each iteration the size of the remaining search space is halved. Therefore, the algorithm runs in O(log n) time.

12)

To find the smallest element in an n-element bitonic array in O(log n) time, we can use a modified binary search algorithm. The key idea is to find the point at which the array switches from being increasing to being decreasing, which we will call the "peak" of the array. Once we have found the peak, we can perform two binary searches: one to search for the smallest element in the increasing portion of the array, and one to search for the smallest element in the decreasing portion of the array.

The algorithm can be described as follows:

1. Initialize two variables, **left** and **right**, to the first and last indices of the array, respectively.
2. While **left** < **right**, compute the middle index **mid** as **(left + right) // 2**.
3. If **A[mid] > A[mid+1]**, then we have found the peak of the array. Set **peak** to **mid** and break out of the loop.
4. If **A[mid] < A[mid+1]**, then the peak must be in the right half of the array. Set **left** to **mid+1** and continue the loop.
5. If **A[mid] > A[mid+1]**, then the peak must be in the left half of the array. Set **right** to **mid** and continue the loop.
6. Once we have found the peak, perform two binary searches to find the smallest element in each half of the array.
7. For the increasing portion of the array (from index 0 to the peak), perform a standard binary search to find the smallest element.
8. For the decreasing portion of the array (from the peak to the last index), perform a reverse binary search (i.e., search from the right to the left) to find the smallest element.
9. Compare the smallest elements found in steps 7 and 8, and return the smaller of the two.

The time complexity of this algorithm is O(log n) because we are performing two binary searches, each of which takes O(log n) time. The overall time complexity of the algorithm is therefore O(log n).

13)

We can use a divide and conquer approach to solve this problem in O(n log n) time. The algorithm works as follows:

1. Divide the array of stock prices into two halves, left and right.
2. Recursively find the best buying and selling days for the left half and the right half.
3. Find the minimum stock price in the left half and the maximum stock price in the right half.
4. Calculate the difference between the minimum stock price in the left half and the maximum stock price in the right half. This is the maximum profit that can be made by buying on or before the day of the minimum price and selling on or after the day of the maximum price.
5. Compare the maximum profits obtained from the left half, the right half, and the combined left and right halves. Return the maximum of these three profits.

Here is the implementation of the algorithm in Python:

```
def max_profit(prices):
    n = len(prices)
    if n < 2:
        return "Cannot make a profit with less than 2 prices"
    if n == 2:
        if prices[0] < prices[1]:
            return "Buy on 1, sell on 2"
        else:
            return "Cannot make a profit with only 2 prices"
    mid = n // 2
    left_min = min(prices[:mid])
    right_max = max(prices[mid:])
    left_profit = max_profit(prices[:mid])
    right_profit = max_profit(prices[mid:])
    cross_profit = right_max - left_min
    if left_profit >= right_profit and left_profit >= cross_profit:
        return left_profit
    elif right_profit >= left_profit and right_profit >= cross_profit:
        return right_profit
    else:
        return cross_profit
```

The algorithm works by dividing the input array into smaller subarrays until the subarrays have only two or fewer elements. This can be done in O(log n) time since the array is halved at each level of recursion. Once we have the subarrays, we can find the maximum profit that can be made in each subarray in O(n) time by finding the minimum and maximum prices in each subarray. The maximum profit that can be made by buying and selling within a subarray is the difference between the maximum and minimum prices in the subarray. Finally, we combine the results from the left and right subarrays and the cross section of the array that spans the middle of the array to find the maximum profit that can be made overall. This can also be done in O(n) time. Therefore, the overall time complexity of the algorithm is O(n log n).

14)

One approach to solve this problem is to use the binary search algorithm to iteratively partition the values into smaller subsets until we find the median.

1. Start by querying the median value of each database (i.e., k=n).
2. Let x and y be the values returned by the queries to database 1 and database 2, respectively.
3. If x is less than y, then the median value must be in the set of values larger than x in database 1 and the set of values smaller than y in database 2. Therefore, we repeat the algorithm on these two sets of values.
4. If y is less than x, then the median value must be in the set of values larger than y in database 2 and the set of values smaller than x in database 1. Therefore, we repeat the algorithm on these two sets of values.
5. If x and y are equal, then we have found the median and we can return it.
6. Repeat steps 2-5 until we find the median.

Each iteration of the algorithm reduces the size of the search space by half, so the total number of queries is O(log n). Therefore, this algorithm finds the median using at most O(log n) queries.

15)

We can use a divide-and-conquer approach to solve this problem in O(n log n) time with only invocations of the equivalence tester.

Let the set of n cards be denoted as S. We can use the following recursive algorithm:

1. If S contains only one card, return that card as the majority card.
2. Divide S into two equal-sized subsets, S1 and S2.
3. Recursively find the majority cards of S1 and S2.
4. If both S1 and S2 have the same majority card M, then M is the majority card of S and we can return it.
5. If not, we need to determine which of the two majority cards is the majority card of S, which requires comparing the number of cards equivalent to each of the two majority cards.
6. We can do this by counting the number of cards in S equivalent to M1 and M2, respectively, by invoking the equivalence tester n/2 times for each majority card.

7. If the number of cards equivalent to M1 is greater than n/2, then M1 is the majority card of S; otherwise, M2 is the majority card of S.
8. Return the majority card of S.

The correctness of this algorithm can be proven by induction on the size of the set S. The base case of S containing only one card is trivial. For the induction step, if both S1 and S2 have the same majority card, then the algorithm correctly returns that card as the majority card of S. If not, then the majority card of S must be either M1 or M2, since any other card cannot be the majority card of both S1 and S2. We can determine which of the two majority cards is the majority card of S by counting the number of cards equivalent to each of them, and this requires only O(n) invocations of the equivalence tester in total.

The running time of this algorithm is O(n log n), since at each recursive level, we divide the set of cards in half and recursively solve two subproblems of size n/2, and the total number of recursive levels is O(log n). Each level requires only O(n) invocations of the equivalence tester, so the total number of invocations is O(n log n).