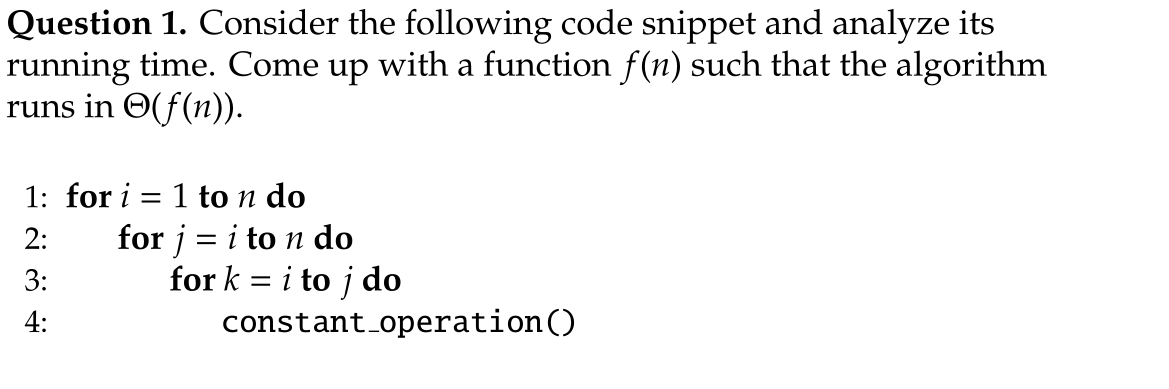
Algorithms and Data Structures  
Homework 1

This homework is divided into two parts:

# Part I Running Time Analysis



**Answer: Theta (n^3)**

**Explanations:**

The code consists of three nested loops:

1. The outer loop runs with variable i from 1 to n .

2. The middle loop runs with variable j from i to n .

3. The inner loop runs with variable k from i to j .

Our goal is to compute the total number T(n) of times constant\_operation() is executed.

T(n) = \sum\_{i=1}^{n} \sum\_{j=i}^{n} \sum\_{k=i}^{j} 1

= \sum\_{i=1}^{n} \sum\_{j=i}^{n} (j - i + 1)

= \sum\_{i=1}^{n} ((n – i)(n - i + 1)/2 + n - i + 1)

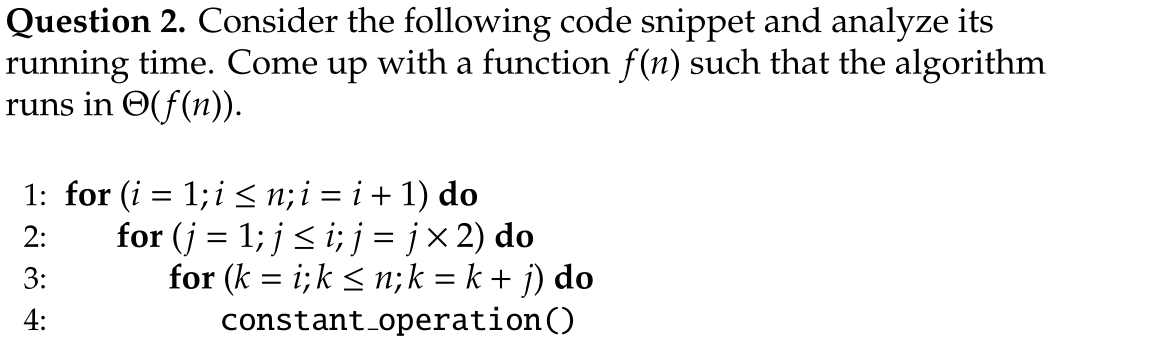
= (\sum\_{i=1}^{n} (n - i)(n - i + 1))/2 + \sum\_{i=1}^{n} (n - i + 1)

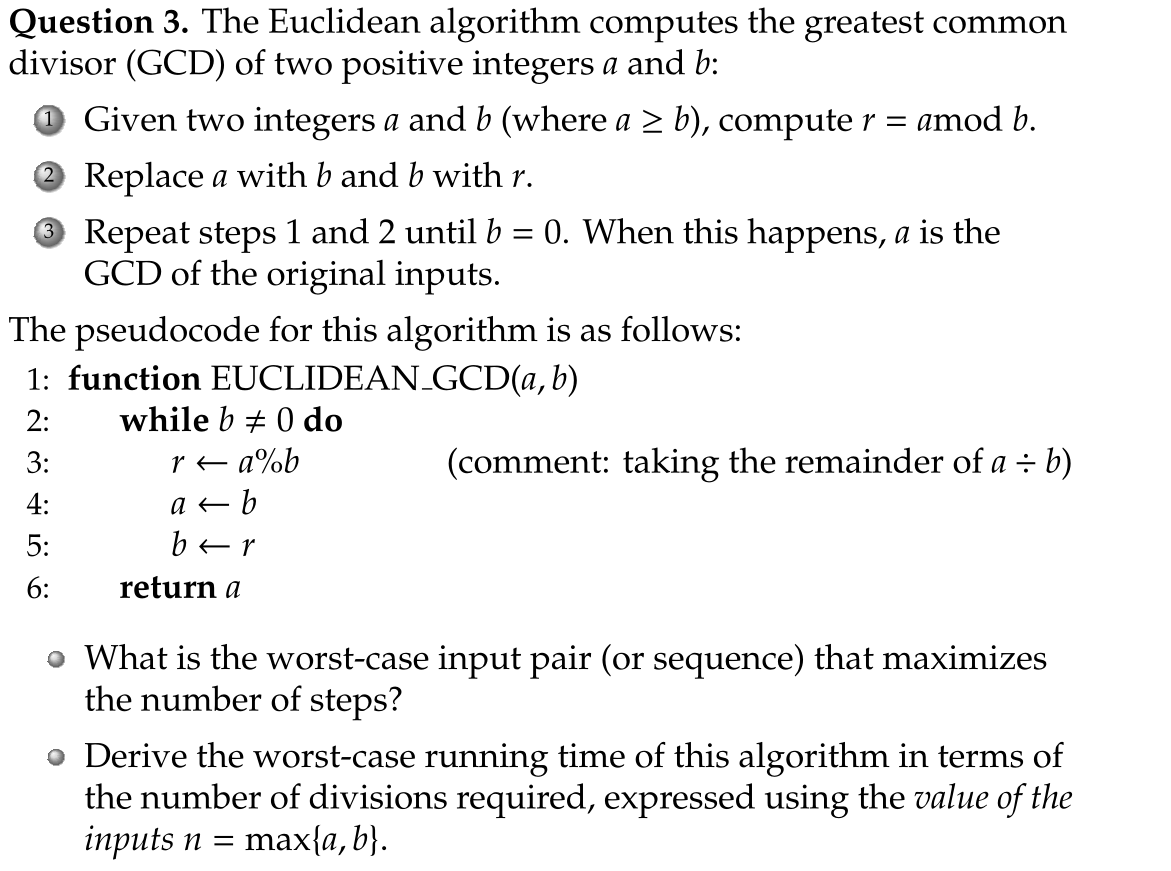
**=** (\sum\_{s=0}^{n - 1} s(s + 1))/2 + \sum\_{s=0}^{n - 1} (s + 1)

= (n - 1)n(n + 1)/6 + (n^2 + n)/2

= ((n - 1)n(n + 1) + 3(n^2 + n))/6

**Therefore T(n) is Theta (n^3)**





**Answer: Theta(n^2)**  
Analysis:  
1. Outer Loop (i):

- Runs from 1 to n.

- Total iterations: n.

2. Middle Loop (j):

- For each i, j starts at 1 and multiplies by 2 each time until j exceeds i.

- j takes on all powers of 2 less than or equal to i.

- Number of iterations for each i: \lfloor \log\_2(i) \rfloor + 1.

3. Inner Loop (k):

- For each i and j, k starts at i and increments by j until k exceeds n.

- Number of iterations: \lfloor (n - i)/j \rfloor + 1.

The total number of times `constant\_operation()` is called is:

T(n) = \sum\_{i = 1}^{n} \sum\_{j = 1; \, j \leq i; \, j \times = 2} ( \lfloor \frac{n - i}{j} \rfloor + 1 )

= \sum\_{j = 1; \, j \leq n; \, j \times = 2} \sum\_{i = j}^{n} ( \frac{n - i}{j} + 1 )

= \sum\_{k = 0}^{\lfloor \log\_2 n \rfloor} [(n - 2^k)(n - 2^k + 1)/2^{k+1} + n - 2^k + 1]

= \sum\_{k = 0}^{\lfloor \log\_2 n \rfloor} (n(n + 1)/2^{k+1} - 2^{k - 1} + 1/2)

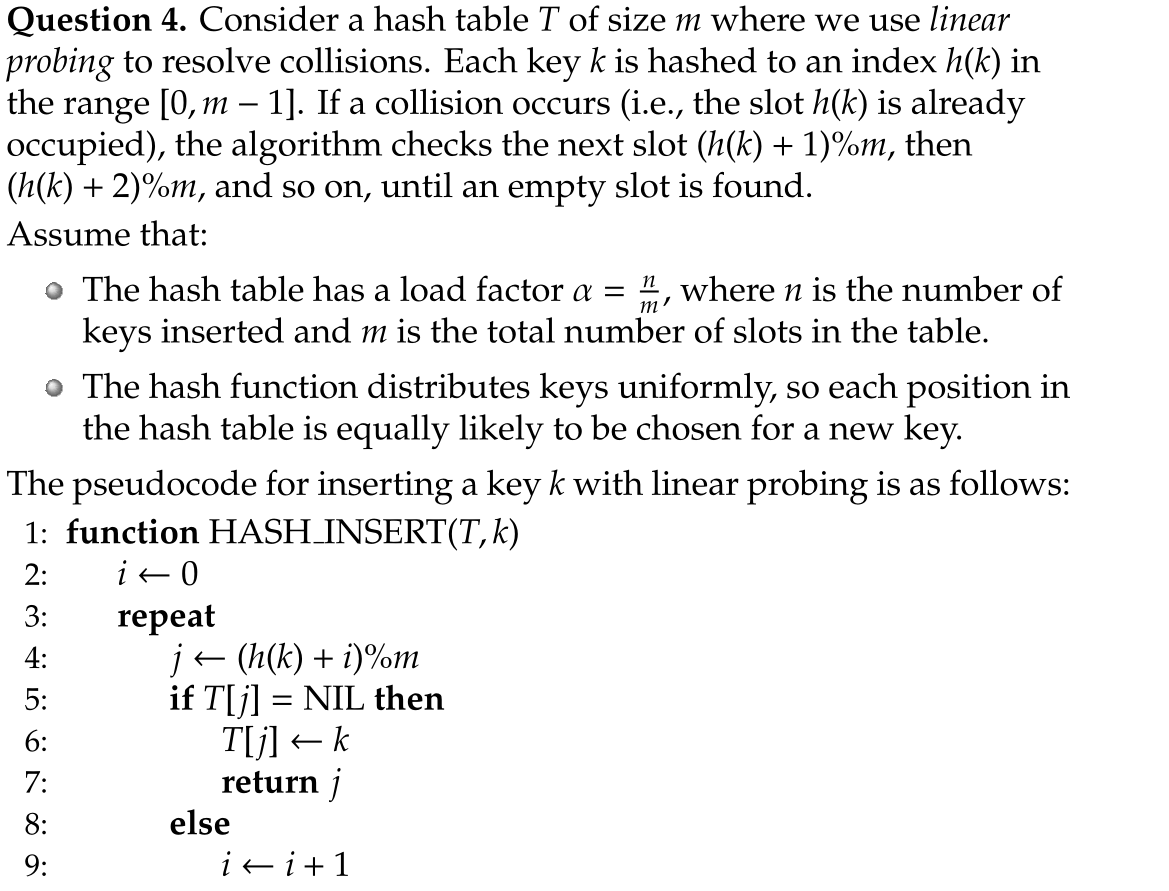
The dominant term in T(n) is \sum\_{k = 0}^{\lfloor \log\_2 n \rfloor} n^2/2^{k+1} which is **Theta(n^2)**

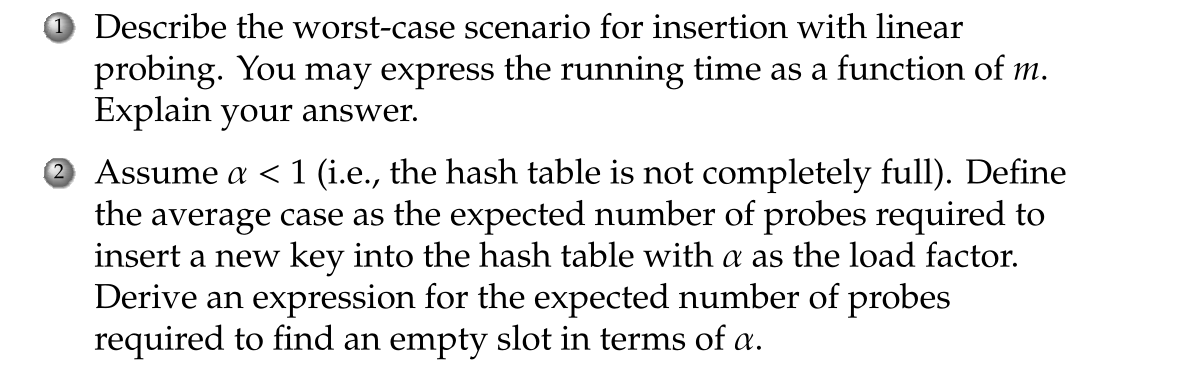
The worst-case input pair for the Euclidean algorithm occurs when the two integers a and b are **consecutive Fibonacci numbers**. Specifically, when a=F\_{k+1}​ and b=F\_k ​, where F\_k ​ is the k-th Fibonacci number.

**Running time analysis:**

When a and b are consecutive Fibonacci numbers, the number of division steps k needed to reduce b to 0 is the index of the smaller Fibonacci number F\_k=b.

Therefore the number of division steps k required by the Euclidean algorithm is proportional to the logarithm of the k-th Fibonacci. Since the Fibonacci numbers grow exponentially with a base equal to the golden ratio (roughly 1.618), k = **O(log n)**.





**Answer:** Under the assumption of uniform hashing and a load factor alpha , the expected number of probes E[P] required to insert a new key into the hash table using linear probing is:

**E[P] = 1/(1 - \alpha)**

Working:

1. Probability of a Slot Being Occupied

- Each slot is equally likely to be the home position for any key.

- The probability that a particular slot is occupied is \alpha.

- The probability that a slot is empty is 1-\alpha.

2. Probing Sequence and Geometric Distribution

- The number of probes needed follows a geometric distribution adjusted for linear probing.

- The probability that the first k - 1 slots are occupied and the k-th slot is empty is:

P\_k = \alpha^{k - 1} (1 - \alpha)

3. Calculating the Expected Number of Probes

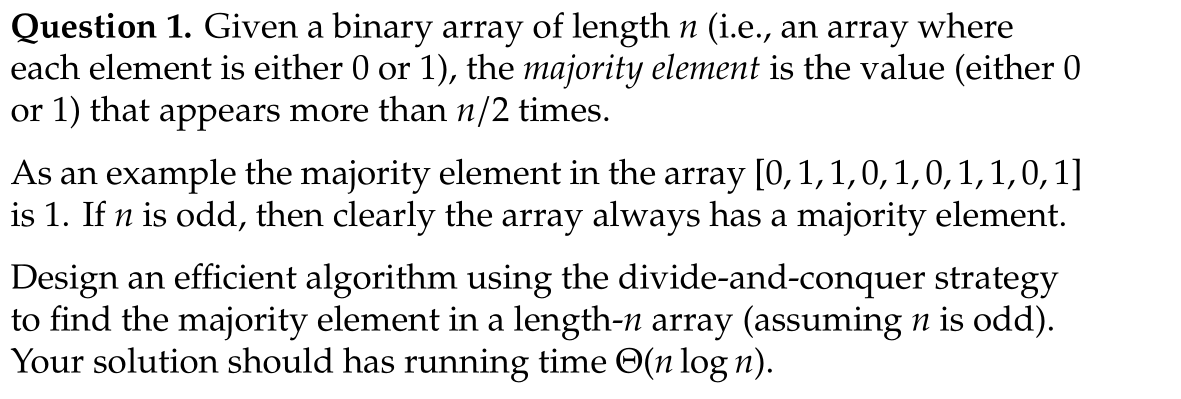
- The expected number of probes is:

E[P] = \sum\_{k=1}^{\infty} k P\_k = \sum\_{k=1}^{\infty} k \alpha^{k - 1} (1 - \alpha)

- This is a standard expected value for a geometric distribution:

E[P] = 1/(1 - \alpha)

# Part 2 Divide and Conquer



We will use a recursive divide-and-conquer algorithm called **MajorityElement(A, low, high)**

function MajorityElement(A, low, high):

if low == high:

return A[low]

mid = floor((low + high) / 2)

leftMajority = MajorityElement(A, low, mid)

rightMajority = MajorityElement(A, mid + 1, high)

if leftMajority == rightMajority:

return leftMajority

else:

leftCount = countOccurrences(A, leftMajority, low, high)

rightCount = countOccurrences(A, rightMajority, low, high)

if leftCount > (high - low + 1) / 2:

return leftMajority

else:

return rightMajority

function countOccurrences(A, x, low, high):

count = 0

for i from low to high:

if A[i] == x:

count += 1

return count

**Running time:**

Let T(n) be the running time for an array of size n.

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

This is a standard divide-and-conquer recurrence, which solves to:

T(n) = Theta(n log n)



We will use a modified binary search algorithm to find the k-th smallest element in the union of two sorted arrays. The key idea is to eliminate parts of the arrays that cannot contain the k-th smallest element at each step, thus reducing the problem size logarithmically.

Define function **find\_kth\_smallest(A, B, k)** where A and B are the input arrays and k is the position of the desired smallest element

**Step 1. Base Cases:**

- If one array is empty: Return the k-th element from the other array.

- If k = 1: Return the minimum of the first elements of A and B.

**Step 2. Recursive Step:**

- Let i = min(\lfloor k/2 \rfloor, size(A))

- Let j = k - i

- If A[i - 1] < B[j - 1]:

- The first i elements of A cannot be the k-th smallest.

- **Recursively call find\_kth\_smallest(A[i:], B, k - i)**

- Else:

- The first j elements of B cannot be the k-th smallest.

- **Recursively call find\_kth\_smallest(A, B[j:], k - j)**

**Step 3. Compute the Median:**

- Find the n-th smallest element m1 using find\_kth\_smallest(A, B, n).

- Find the (n + 1)-th smallest element m2 using find\_kth\_smallest(A, B, n + 1).

- median = (m1 + m2)/2

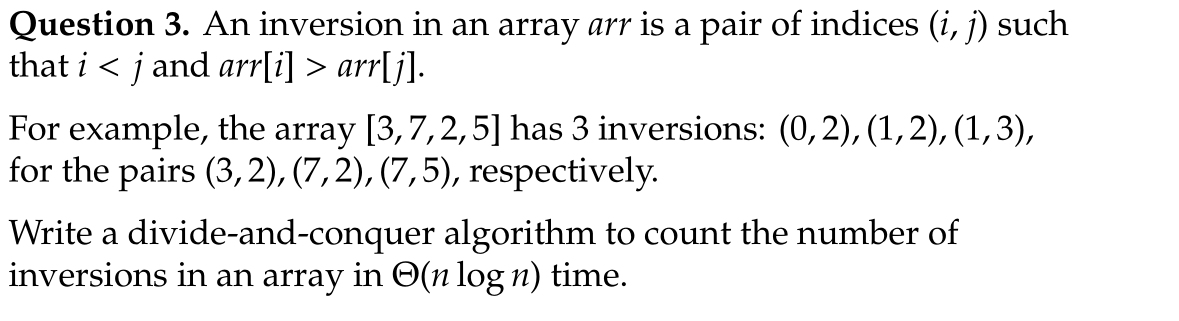
**Running time:**

- At each recursive call, we reduce the size of the problem by at least half.

- The number of recursive calls is proportional to log n.

- No merging or linear scans are performed.

- Therefore, the overall time complexity is **Theta(log n).**



We can modify the Merge Sort algorithm to count inversions while sorting the array.

-Divide: Split the array into two halves.

-Conquer: Recursively count inversions in each half.

-Combine: Merge the two sorted halves. While merging, when an element from the right half is placed before an element from the left half, it indicates inversions equal to the number of remaining elements in the left half.

**function CountInversions(arr, left, right):**

if left >= right:

return 0 // Base case: single element has no inversions

mid = (left + right) / 2

inversions = 0

inversions += CountInversions(arr, left, mid)

inversions += CountInversions(arr, mid + 1, right)

inversions += MergeAndCount(arr, left, mid, right)

return inversions

**function MergeAndCount(arr, left, mid, right):**

LeftArray = arr[left..mid]

RightArray = arr[mid+1..right]

i = 0; j = 0; k = left

inversions = 0

while i < length(LeftArray) and j < length(RightArray):

if LeftArray[i] <= RightArray[j]:

arr[k] = LeftArray[i]; i += 1

else:

arr[k] = RightArray[j]

inversions += (length(LeftArray) - i) // Count inversions

j += 1

k += 1

while i < length(LeftArray):

arr[k] = LeftArray[i]

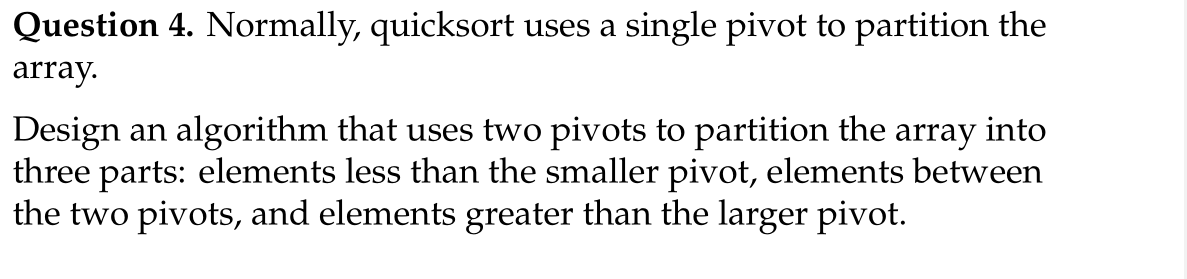
i += 1; k += 1

while j < length(RightArray):

arr[k] = RightArray[j]

j += 1; k += 1

return inversions



This algorithm is known as **Dual-Pivot Quicksort**.

1. Choosing the Pivots

- For simplicity, select the first and last elements of the array as pivots.

- Let p = A[low] and q = A[high].

- If p > q , swap them.

2. Partitioning Process

- i starts from low + 1.

- k starts from low + 1.

- j starts from high - 1.

- While k <= j , do:

- Case 1: If A[k] < p:

- Swap A[k] with A[i].

- Increment i and k.

- Case 2: If p <= A[k] <= q:

- Increment k.

- Case 3: If A[k] > q:

- Swap A[k] with A[j].

- Decrement j.

- Final Swaps:

- Decrement i and increment j.

- Swap A[low] with A[i] (placing p in its correct position).

- Swap A[high] with A[j] (placing q in its correct position).

3. Recursive Sorting

- Subarrays to Sort:

- Left Subarray: A[low.. i - 1] (elements less than p ).

- Middle Subarray: A[i + 1.. j - 1] (elements between p and q ).

- Right Subarray: A[j + 1.. high] (elements greater than q ).

- Recursive Calls:

- Recursively apply Dual-Pivot Quicksort to the left, middle, and right subarrays.