ps4

Ji Qiankun 221300066

June 12, 2024

1

(a) Algorithm to Sort an Input Array using SqrtSort Subroutine:
1. Define a function SqrtSortArray(A, n) that takes an input array A of size n as input.
2. If $n \leq 1$, return the array as it is (base case).
3. Calculate the square root of n and store it in a variable sqrt_n.
4. Initialize a variable k as 0.
5. While $k + sqrt\_n < n$:
- Call the SqrtSort(k) subroutine to sort the subarray A[(k + 1) $\cdots$ (k + sqrt_n)] in place.
- Increment k by sqrt_n.
6. Call the SqrtSort(k) subroutine to sort the remaining subarray A[(k + 1) $\cdots$ n] in place.
7. Return the sorted array A. Pseudocode:

Algorithm 1 SqrtSortArray(A, n)

```
function SqrtSortArray(A, n)
    if n ≤ 1 then
        return A
    end if
    sqrt_n ← √n
    k ← 0
    while k + sqrt_n < n do
        SqrtSort(k)
        k ← k + sqrt_n
    end while
    SqrtSort(k)
    return A
end function
```

Correctness:
The algorithm divides the input array into subarrays of size sqrt_n (except for

the last subarray which may have a smaller size). It then calls the SqrtSort subroutine to sort each subarray in place. Since sqrt_n is an integer, the subarrays are disjoint and cover the entire input array. Therefore, sorting each subarray individually will result in the entire array being sorted correctly.

Worst-case Analysis:

In the worst case, the algorithm will call the SqrtSort subroutine $\sqrt{n}$ times. This is because the algorithm divides the array into $\sqrt{n}$ subarrays, each of size $\sqrt{n}$, and calls SqrtSort on each subarray. Therefore, the worst-case number of calls to SqrtSort is $\sqrt{n}$.

(b) Worst-case Running Time of the Resulting Sorting Algorithm:
Analyzed as follows:
At each level of recursion, the size of the subarray being sorted is roughly $n^{1/4}$. The recursion continues until the subarray size becomes 1 or smaller, which happens when $n^{1/4}$ reaches 1. In other words, the recursion depth is approximately 4.
At each level of recursion, the SqrtSort subroutine is called $\sqrt{n}$ times,where n is the subarray size. Therefore, the total number of calls to SqrtSort can be calculated as the sum of $\sqrt{n}$ at each recursion level, which can be expressed as:

$\sqrt{n} + \sqrt{\sqrt{n}} + \sqrt{\sqrt{\sqrt{n}}} + ...$

This is a geometric series with a common ratio of sqrt(n). So the sum of a geometric series can be calculated using the formula:

$sum = a * (1 - r^k)/(1 - r)$

where a is the first term, r is the common ratio, and k is the number of terms.In this case, a = $\sqrt{(n)}$, r = $\sqrt{(n)}$, and k = 4 (recursion depth). Plugging these values into the formula, we get:

$sum = \sqrt{(n)} \times (1 - \sqrt{(n)^4})/(1 - \sqrt{(n)})$

Therefore, the worst-case running time of the resulting sorting algorithm is simplified to $\sqrt{(n)} \times (1 - n^2)/(1 - \sqrt{(n)})$.

2

(a) To use a recursive approach:
Let $P(k)$ be the probability of falling off the right end of the path, starting from vertex k. We want to show that $P(k) = k/(n + 1)$.
Base Case:
When $k = 1$, we are already at the leftmost vertex. Therefore, the probability of falling off the right end is 0, i.e., $P(1) = 0$.
Recursive Step:
For $k > 1$, we can consider the two possible moves: moving left to vertex k - 1 or moving right to vertex k + 1.
If we move left to vertex k - 1, the probability of falling off the right end is P(k - 1). This occurs with a probability of 1/2, as we have an equal chance of moving left or right at each step.

If we move right to vertex k + 1, the probability of falling off the right end is 1 - P(k + 1). This occurs with a probability of 1/2, as we have an equal chance of moving left or right at each step.
Therefore, we can express the probability of falling off the right end from vertex k as:
$P(k) = (1/2)P(k-1) + (1/2)(1 - P(k+1))$

To prove that P(k) = k/(n + 1), we can use mathematical induction.
Base Case:
For $k = 1$, we have already shown that $P(1) = 0$, which is equal to $1/(n+1)$ when $k = 1$.
Inductive Step:
Assume that $P(k) = k/(n+1)$ holds for some $k > 1$. We will show that $P(k+1) = (k+1)/(n+1)$.
Using the recursive formula, we have:
$P(k+1) = (1/2)P(k) + (1/2)(1 - P(k+2))$

Substituting $P(k) = k/(n+1)$, we get:

$P(k+1) = (1/2)(k/(n+1)) + (1/2)(1 - P(k+2n+1)) + (1/2)(1 - P(k+2))$

Simplifying the expression, we have:
$P(k+1) = (k/(2(n+1))) + (1/2) - (1/2)P(k+2)$

Since we assumed that $P(k) = k/(n+1)$, we can substitute this into the equation and simplifying further, we get:
$P(k+1) = (k+1)/(n+1)$

Thus, we have shown that $P(k) = k/(n+1)$ for all $k > 1$.
Therefore, the probability of falling off the right end of the path, starting from vertex k, is exactly $k/(n+1)$.

(b) Algorithm FairDie:
1. Initialize a variable called "result" to store the result of the FairDie algorithm.
2. Repeat the following steps until "result" is not equal to zero:
a. Call the UnfairDie subroutine and store the result in "result".
b. If "result" is in the set $1, 2, 3, 4, 5, 6$, return "result".
3. If the algorithm reaches this point, repeat step 2 until a valid result is obtained.
Pseudocode:

Algorithm 2 FairDie Algorithm

InputInput OutputOutput result ← 0

while result = 0 **do** result ← UnfairDie()

if result ∈ {1, 2, 3, 4, 5, 6} **then return** result

The expected number of times the FairDie algorithm calls the UnfairDie subroutine can be calculated as follows:
Let X be the random variable representing the number of times thebUnfairDie subroutine is called until a valid result is obtained.
The probability of obtaining a valid result on each call is 6/6 = 1. Therefore,

the probability of not obtaining a valid result on each call is $1 - 1/6 = 5/6$.
The expected value of X can be calculated using the formula for the expected value of a geometric distribution:
$E(X) = 1/p = 1/(5/6) = 6/5$

Hence, the expected number of times the FairDie algorithm calls the UnfairDie subroutine is 6/5.

3

(a) No, I don't. It is not possible to develop a new data structure for priority queues that supports the operations Insert, GetMax, and ExtractMax all in O(1) worst-case time.
The reason is that maintaining the maximal elements in a priority queue inherently requires at least O(log n) time complexity for insertion and removal of elements.
Priority queues are typically implemented using either a binary heap or a Fibonacci heap. In a binary heap, insertion and extraction of elements take O(log n) time because each operation may require traversing the height of the tree, which can be up to log n levels deep. Similarly, finding the maximum element also takes O(log n) time because it may need to traverse the height of the tree. Fibonacci heaps have better average-case performance for insertion and extraction compared to binary heaps, but they still do not support these operations in O(1) time complexity. In fact, finding the maximum element in a Fibonacci heap typically requires O(log n) time as well.
Therefore, it is not possible with current known data structures and algorithms.

(b) Using a decision tree model:
In a comparison-based sorting algorithm, each comparison between two elements in the input sequence corresponds to a decision made by the algorithm, so the decision tree model represents all possible outcomes of these comparisons.
In the given scenario, we have n elements, and for each element initially in position i such that $imod4 = 0$, it is either already in its correct position or one place away from its correct position. This means that for these elements, there are at most 2 possible outcomes for each comparison.
Considering the worst-case scenario, where all elements in positions i such that $imod4 = 0$ are one place away from their correct positions, we can construct a decision tree with $2^{(n/4)}$ leaf nodes. Each leaf node represents a unique permutation of the elements.
Since there are $n!$ possible permutations of n elements, and each leaf node in the decision tree represents a unique permutation, we have $2^{(n/4)} \leq n!$. Taking the logarithm of both sides, we get $(n/4)log2 \leq \log_2(n!)$.
Using Stirling's approximation, we can approximate $\log_2(n!)$ as $n\log_2 n - n\log_2 e$, where e is the base of the natural logarithm. Substituting this approximation into the inequality, we have $(n/4)log2 \leq n\log_2 n - n\log_2 e$.
Simplifying the inequality, we have $(1/4)log2 \leq 1 - (\log_2 e/\log_2 n)$.

As n approaches infinity, the right-hand side of the inequality approaches 1, while the left-hand side remains constant. Therefore, for sufficiently large values of n, we have $(1/4)log2 \leq 1 - (\log_2 e / \log_2 n) < 1$.
This implies that the decision tree must have at least $\log_2 n/4$ levels. Since the height of the decision tree represents the number of comparisons required by the sorting algorithm, we can conclude that any comparison-based sorting algorithm for this scenario requires (nlogn) comparisons in the worst case.

Hence, the $\Omega(nlogn)$ lower bound on comparison-based sorting still holds in this case.

4

(a) Using the counting sort algorithm:
Here is the algorithm:
1. Create an array *count*[] of size n^6, initialized with all zeros.
2. Traverse the input array and for each element x, increment $count[x]$ by 1.
3. Create an output array *sorted*[] of size n.
4. Initialize a variable index to 0.
5. Traverse the *count*[] array from index 0 to $n^6 - 1$: - If $count[i]$ is greater than 0, assign i to $sorted[index]$ and increment index by 1.
6. Return the sorted array.
Pseudocode: Correctness:

Algorithm 3 CountingSort(inputArray, n)

```
function CountingSort(inputArray, n)
    count ← CreateArray(n^6, 0)
    for i ← 0 to length(inputArray) − 1 do
        count[inputArray[i]] ← count[inputArray[i]] + 1
    end for
    sorted ← CreateArray(n, 0)
    index ← 0
    for i ← 0 to n^6 − 1 do
        if count[i] > 0 then
            sorted[index] ← i
            index ← index + 1
        end if
    end for
    return sorted
end function=0
```

- The counting sort algorithm works by counting the occurrences of each element in the input array and then reconstructing the sorted array based on the counts. Since the range of the input integers is $[0, n^6 - 1]$, the count array has a size of n^6, which is sufficient to store the counts for all possible values.
- By traversing the count array and reconstructing the sorted array, we ensure

that the output array contains all the input elements in sorted order.
The time complexity of the algorithm is O(n):
- The first step of creating the count array takes O(n) time, as we traverse the input array once.
- The second step of traversing the count array takes a constant time($O(1)$). - The third step of creating the sorted array takes O(n) time, as we traverse the count array once.
- Therefore, the overall time complexity of the algorithm is O(n + 1 + n), which simplifies to O(n).

(b) Using the radix sort algorithm.
Here is the algorithm:
1. Find the maximum length of strings in the array and store it in a variable maxLen.
2. Initialize a variable numChars to 26, representing the number of possible characters in the alphabet (assuming lowercase English letters).
3. Create numChars buckets, each representing a character in the alphabet.
4. Starting from the last character (rightmost) to the first character (leftmost), perform the following steps for each character position:
- Initialize an empty array of strings for each bucket.
- Traverse the input array of strings and for each string, place it in the corresponding bucket based on the character at the current position.
- Concatenate the strings in each bucket to form a new sorted array of strings.
5. Return the sorted array of strings.
Pseudocode:

Algorithm 4 RadixSort(strings)

```
function RadixSort(strings)
    maxLen ← FindMaxLength(strings)
    numChars ← 26                          ▷ Assuming lowercase English letters
    for pos ← maxLen − 1 to 0 do
        buckets ← CreateEmptyBuckets(numChars)
        for i ← 0 to length(strings) − 1 do
            char ← GetCharacter(strings[i], pos)
            bucketIndex ← GetBucketIndex(char)
            AddToStringArray(buckets[bucketIndex], strings[i])
        end for
        strings ← ConcatenateBuckets(buckets)
    end for
    return strings
end function
```

Correctness:
- The radix sort algorithm works by sorting the strings based on each character position, starting from the least significant digit to the most significant digit.

By repeatedly sorting the strings based on each character position, we ensure that the strings are sorted in the standard alphabetical order.
- Since we are considering lowercase English letters, which have a constant number of possible characters (26), the number of buckets is fixed. This allows us to perform the sorting operation in a stable manner, preserving the relative order of strings with the same character at the current position.
The time complexity of the algorithm is O(n):
- Finding the maximum length of strings in the array takes O(n) time, as we need to traverse the input array once.
- The radix sort operation iterates over each character position, which takes O(maxLen) time.
- Within each iteration, placing the strings in the corresponding buckets and concatenating the strings take O(n) time, as we need to traverse the input array once.
- Therefore, the overall time complexity of the algorithm is O(n + maxLen n), which simplifies to O(n).

5

Using the following method:
1. Sort the wells based on their y-coordinates in ascending order.
2. Calculate the total length of the spurs for each possible location of the main pipeline along the y-axis.
3. Choose the location with the minimum total length of the spurs.
Algorithm to Minimize Total Length of Spurs:
1. Define a function MinimizeSpurLength(wells) that takes the coordinates of the wells as input.
2. Sort the wells based on their y-coordinates in ascending order.
3. Initialize variables min_length and optimal_location to store the minimum total length of the spurs and the optimal location of the main pipeline, respectively.
4. Iterate over each well from the first to the second-to-last well:
- Calculate the total length of the spurs for the current location of the main pipeline along the y-axis.
- If the total length is less than min_length, update min_length and optimal_location.
5. Return optimal_location as the optimal location of the main pipeline.
Pseudocode:
Correctness:
The algorithm considers each possible location of the main pipeline along the y-axis. For each location, it calculates the total length of the spurs by summing the distances from each well to the main pipeline. By iterating over all possible locations and selecting the one with the minimum total length, the algorithm guarantees that the chosen location minimizes the total length of the spurs.
Runtime:

Algorithm 5 MinimizeSpurLength(wells)

```
function MinimizeSpurLength(wells)
    Sort wells based on y-coordinates in ascending order
    min_length = infinity
    optimal_location = null
    for i = 0 to length(wells) − 2 do
        total_length = 0
        for j = 0 to i do
            total_length += distance(wells[j], main_pipeline_location)
        end for
        for j = i + 1 to length(wells) − 1 do
            total_length += distance(wells[j], main_pipeline_location)
        end for
        if total_length ¡ min_length then
            min_length = total_length
            optimal_location = wells[i].y
        end if
    end for
    return optimal_location
end function
```

The algorithm has a time complexity of $O(n^2)$ because it iterates over each well and calculates the total length of the spurs for each possible location of the main pipeline. Sorting the wells based on their y-coordinates takes $O(nlogn)$ time, and the nested loops for calculating the total length of the spurs contribute to an additional $O(n^2)$ time complexity.Therefore, the overall runtime of the algorithm is $O(n^2)$.
In terms of space complexity, the algorithm requires $O(n)$ space to store the sorted wells and the variables for the minimum length and optimal location.

6

(a) The algorithm described can be summarized as follows:
1. If i is greater than or equal to n/2, use the Select algorithm to find the i-th order statistic directly. This will require S(n) comparisons.
2. If i is less than n/2, perform the following steps:
a. Find the median of the n elements using the Select algorithm. This will require S(n) comparisons.
b. Partition the elements into three groups: those smaller than the median, those equal to the median, and those larger than the median.
c. If the number of elements smaller than the median is greater than i, recursively apply the algorithm to the smaller group of elements. Specifically, apply Ui([n/2]) to find the i-th smallest element in the smaller group. This will require Ui([n/2]) comparisons.

d. If the number of elements smaller than the median is equal to i, return the median as the i-th smallest element.
e. If the number of elements smaller than the median is less than i, recursively apply the algorithm to the larger group of elements. Specifically, apply Ui(n - [n/2] - 1) to find the (i - (number of elements smaller than the median) - 1)-th smallest element in the larger group. This will require Ui(n - [n/2] - 1) comparisons.
The total number of comparisons made by this algorithm, denoted as Ui(n), can be calculated using the provided recurrence relation:
- If $i \geq n/2$, Ui(n) = S(n).
- If $i < n/2$, Ui(n) = [n/2] + Ui([n/2]) + S(2i).

By using this algorithm, we can find the i-th smallest element with fewer comparisons in the worst case compared to using the Select algorithm directly.

(b) Using mathematical induction:

Base case: When n = 1, the algorithm only needs to compare the single element with itself, so $Ui(1) = 1 = n + O(S(2i)lg(n/i))$.
Inductive step: Assume that the statement holds for all values up to n - 1. We need to prove that it holds for n.

If $i < n/2$, according to the algorithm:
1. We use the Select algorithm to find the median of the n elements, which requires S(n) comparisons.
2. We partition the elements into three groups: smaller, equal, and larger than the median.
3. If the number of elements smaller than the median is greater than i, we recursively apply the algorithm to the smaller group of elements.
4. If the number of elements smaller than the median is equal to i, we return the median as the i-th smallest element.
5. If the number of elements smaller than the median is less than i, we recursively apply the algorithm to the larger group of elements.
Now analyze the number of comparisons made in each step:
1. Finding the median using the Select algorithm requires S(n) comparisons.
2. Partitioning the elements can be done in O(n) time and does not involve any additional comparisons.
3. If we recursively apply the algorithm to the smaller group, the number of elements becomes at most n/2. According to the induction hypothesis, the number of comparisons made in this step is Ui([n/2]).
4. If the number of elements smaller than the median is equal to i, we return the median without any additional comparisons.
5. If we recursively apply the algorithm to the larger group, the number of elements becomes at most n - [n/2] - 1. According to the induction hypothesis, the number of comparisons made in this step is Ui(n - [n/2] - 1).
Therefore, the total number of comparisons made by the algorithm can be expressed as:
$Ui(n) = S(n) + O(n) + Ui([n/2]) + Ui(n - [n/2] - 1)$.

(c) Also using mathematical induction.

Base case: When n = 1, the algorithm only needs to compare the single element with itself, so $Ui(1) = 1 = n + O(lgn)$.

Inductive step: Assume that the statement holds for all values up to n - 1, prove that it holds for n.

If i is a constant less than n/2, according to the algorithm:

1. We use the Select algorithm to find the median of the n elements, which requires S(n) comparisons.
2. We partition the elements into three groups: smaller, equal, and larger than the median.
3. If the number of elements smaller than the median is greater than i, we recursively apply the algorithm to the smaller group of elements.
4. If the number of elements smaller than the median is equal to i, we return the median as the i-th smallest element.
5. If the number of elements smaller than the median is less than i, we recursively apply the algorithm to the larger group of elements.

Now analyze the number of comparisons made in each step:

1. Finding the median using the Select algorithm requires S(n) comparisons.
2. Partitioning the elements can be done in O(n) time and does not involve any additional comparisons.
3. If we recursively apply the algorithm to the smaller group, the number of elements becomes at most [n/2]. According to the induction hypothesis, the number of comparisons made in this step is Ui([n/2]) = [n/2] + O(lg([n/2])).
4. If the number of elements smaller than the median is equal to i, we return the median without any additional comparisons.
5. If we recursively apply the algorithm to the larger group, the number of elements becomes at most n - [n/2] - 1. According to the induction hypothesis, the number of comparisons made in this step is $Ui(n - [n/2] - 1) = (n - [n/2] - 1) + O(lg(n - [n/2] - 1))$.

So the total number of comparisons made by the algorithm can be expressed as follows:

$Ui(n) = S(n) + O(n) + [n/2] + O(lg([n/2])) + (n - [n/2] - 1) + O(lg(n - [n/2] - 1))$.

Since i is a constant less than n/2, we can treat it as a constant k. Therefore, [n/2] can be simplified to k.

$Ui(n) = S(n) + O(n) + k + O(lg(n/2)) + (n - k - 1) + O(lg(n - k - 1))$.

Simplifying further, we get:

$Ui(n) = n + O(n) + O(lgn) = n + O(lgn)$.

Therefore, if i is a constant less than n/2, then $Ui(n) = n + O(lgn)$.
Proof completed.

(d) Proof:
First consider the algorithm that uses Select as a subroutine to find the i-th order statistic from n numbers, where i = n/k for $k \geq 2$.In this algorithm, we

divide the input array into k equal-sized subarrays, each of size n/k.We then use Select to find the median of each subarray. This takes O(S(n/k)) comparisons in the worst case.
Then denote the medians of the subarrays as M1, M2, ..., Mk.
We use Select to find the median of the medians, which we'll call x. This takes O(S(k)) comparisons in the worst case. Now, we partition the input array around x, creating two subarrays: one with elements less than x and one with elements greater than x.
If i is less than the number of elements in the subarray less than x, we recursively apply the algorithm to the subarray less than x.
If i is greater than the number of elements in the subarray less than x, we recursively apply the algorithm to the subarray greater than x, adjusting i accordingly.
The recursion stops when we find the i-th order statistic in one of the subarrays, which takes O(n) comparisons in the worst case.
Therefore, the total number of comparisons made by the algorithm can be expressed as follows:
Ui(n) = O(S(n/k)) + O(S(k)) + Ui(n/k) + O(n)

Since i = n/k, we can substitute n/k for i in the above equation:
Ui(n) = O(S(n/k)) + O(S(k)) + Ui(n/k) + O(n)

Now, analyze the complexity of the algorithm. The first two terms, O(S(n/k)) and O(S(k)), represent the comparisons made to find the medians of the subarrays and the median of the medians, respectively.The third term, Ui(n/k), represents the number of comparisons made in the recursive calls to find the i-th order statistic in the subarrays. The last term, O(n), represents the comparisons made to partition the input array and find the i-th order statistic in one of the subarrays.
By analyzing the algorithm, we can see that the number of elements in the subarrays decreases by a factor of k in each recursive call.

Therefore, the recursion stops when the size of the subarrays is 2n/k, which corresponds to the base case of the Select algorithm.

Thus, we can express Ui(n) in terms of S(2n/k) as follows:
$Ui(n) = O(S(n/k)) + O(S(k)) + Ui(2n/k) + O(n)$
$= n + O(S(2n/k)) + O(S(k)) + Ui(2n/k) + O(n)$

Since i = n/k, we can substitute n/k for i in the above equation and simplifying the equation, we get:
Ui(n) = n + O(S(2n/k) · lg k)

Proof completed.