# Algorithm Homework

Tai Jiang

October 2023

## Contents

## 1 Show that the solution of $T(n) = T(\lceil n/2 \rceil) + 1$ is $O(\lg n)$ .

**Origin** :

**Exercise** 4.3-2

**Page** 87

**Answer** :

That can build a recursion tree to visualize how this recurrence works:

1. Start with $T(n)$ at the top.
2. At each level of the tree, we have $T(\lceil n/2 \rceil) + 1$.
3. The tree branches into two subproblems, one with size $\lceil n/2 \rceil$ and another with size $\lceil n/2 \rceil$.

The tree might look like Figure 1.

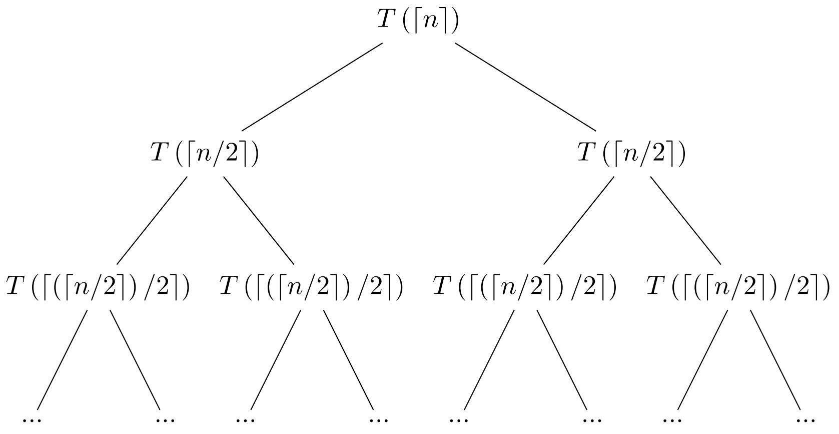
$T(\lceil n \rceil)$
$T(\lceil n/2 \rceil)$
$T(\lceil n/2 \rceil)$
$T(\lceil (\lceil n/2 \rceil)/2 \rceil)$
$T(\lceil (\lceil n/2 \rceil)/2 \rceil)$
$T(\lceil (\lceil n/2 \rceil)/2 \rceil)$
$T(\lceil (\lceil n/2 \rceil)/2 \rceil)$
...

Figure 1: Recursion Tree.

At each level, the size of the problem is divided by 2, and we keep going until the size becomes 1 (or less, in which case we stop). The depth of the tree will be the number of times we can halve n until it becomes 1. So it's the number of times we can take $\lceil n/2 \rceil$ until $\lceil n/2 \rceil \leq 1$.

At each step, we're taking the ceiling of half of the previous value, which is effectively dividing it by 2. We continue this process until the value is less than or equal to 1. So, let k be the number of steps it takes for $\lceil n/2 \rceil$ to become 1, look like equation (1).

$$\frac{\lceil n/2 \rceil}{2^k} \leq 1 \tag{1}$$

Then sovle for k, look like equation (2):

$$\begin{gathered} \frac{n}{2^k} \leq 1 \\ n \leq 2^k \\ \lg n \leq k \end{gathered} \tag{2}$$

The depth of the recursion tree is $O(\lg n)$, which means the algorithm has a time complexity of $O(\lg n)$. So the solution of $T(n) = T(\lceil n/2 \rceil) + 1$ is indeed $O(\lg n)$.

## 2 Use a recursion tree to determine a good asymptotic upper bound on the recurrence $T(n) = 4T(n/2 + 2) + n$. Use the substitution method to verify your answer.

**Origin** :

**Exercise** 4.4-3

**Page** 93

**Answer** :

That can build a recursion tree to visualize how this recurrence works:

1. Start with $T(n)$ at the top.
2. At each level of the tree, we have 4 subproblems of size $T(n/2+2)+n$.
3. The cost of each level is n.

The tree might look like Figure 2.

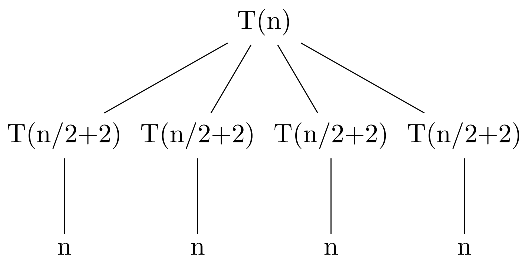

Figure 2: Recursion Tree.

At each level, we have 4 subproblems of size T(n/2 + 2), and each subproblem incurs a cost of n. The number of levels in the tree will depend on how quickly the subproblem size decreases.

The subproblem size is equation (3):

$$T(n) = 4T(n/2+2)+n \tag{3}$$

The subproblem size is n/2 + 2, so we calculate the subproblem size for the next level by equation (4):

$$\begin{aligned} T(n/2+2) &= 4T((n/2+2)/2+2)+n/2+2 \\ &= 4T(n/4+1+2)+n/2+2 \\ &= 4T(n/4+3)+n/2+2 \end{aligned} \tag{4}$$

At each level, we are adding 2 to the subproblem size. Therefore, at level k, the subproblem size will be $n/(2^k)+2k$.

Now, we want to find the level where the subproblem size becomes a constant, $n/(2^k)+2k = CforsomeconstantC$, calculate the k look like equation (5).

$$\begin{aligned} n/(2^k)+2k &= C \\ n/(2^k) &= C-2k \\ 2^k &= n/(C-2k) \\ k &= \lg(n/(C-2k)) \end{aligned} \tag{5}$$

C is a constant. The number of levels in the recursion tree is $O(\lg n)$.

The cost at each level in the recursion tree is equation (6):

$$\begin{aligned} &\text{At level 0:} && n \\ &\text{At level 1:} && 4*n/2 = 2n \\ &\text{At level 2:} && 4*(n/4) = n \\ &\quad ... && \\ &\text{At level k:} && (4^k)*(n/(2^k)) = (n/2^k)*(4^k) = C*4^k \end{aligned} \tag{6}$$

Summing up the costs of all levels(Equation (7)):

$$T(n) = n+2n+4n+...+C*4^k \tag{7}$$

This is a geometric series, and its sum can be bounded by(Equation (8)):

$$T(n) \leq n * (1 - 4^(k+1))/(1-4) \tag{8}$$

Since $k = O(\lg(n))$, $4^(k+1)$ is polynomial in n. Therefore, T(n) is O(n).
verify this result using the substitution method:
Assume that $T(m) \leq km - p$ for some positive constants k and p, where m ¡ n(Equation (9)).

$$\begin{aligned} T(n) &= 4T(n/2+2) + n \\ &\leq 4(k(n/2+2) - p) + n \\ &= 2kn - 4k + 4n - 4p + n \\ &= (2k+5)n - 4k - 4p \end{aligned} \tag{9}$$

Find k and p such that $(2k+5)n - 4k - 4p \leq kn - p.$
This holds if $2k + 5 \leq kand - 4k - 4p \leq -p.$
Solving these inequalities(Equation (10)):

$$\begin{aligned} 2k + 5 &\leq k \\ k &\leq -5 \\ -4k - 4p &\leq -p \\ -4k &\leq 0 \\ k &\geq 0 \end{aligned} \tag{10}$$

Since we can't find k and p that satisfy these inequalities, our initial assumption that $T(m) \leq km - p$ for m ¡ n is incorrect. Therefore, T(n) is not $O(n^k)$ for any positive constant k. Instead, as shown earlier, T(n) is O(n).

## 3 Can the master method be applied to the recurrence $T(n) = 4T(n/2) + n^2 \lg n$? Why or why not? Give an asymptotic upper bound for this recurrence.

**Origin** :

**Exercise** 4.5-4

**Page** 97

**Answer** :

The master theorem can be applied to recurrence
To apply the master theorem, we need to check whether f(n) satisfies the following conditions for some constant $\epsilon > 0$:

1. If $f(n) = O(n^{log_b a - \epsilon})$, for some $\epsilon > 0$, then $T(n) = \Theta(n^{log_b a})$.
2. If $f(n) = \Theta(n^{log_b a})$, then $T(n) = \Theta(n^{log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{log_b a + \epsilon})$, for some $\epsilon > 0$, and if $af(n/b) \leq k * f(n)$ for some k ¡ 1 and sufficiently large n, then $T(n) = \Theta(f(n))$.

The asymptotic upper bound for the given recurrence relation $T(n) = 4T(n/2) + n^2 * \lg n = O(n^2)$.

## 4 Use indicator random variables to compute the expected value of the sum of n dice.

**Origin** :

**Exercise** 5.2-3

**Page** 122

**Answer** :

Let $X_i$ be the random variable that is 1 if the i-th die shows a particular face (1, 2, 3, 4, 5, or 6), and 0 otherwise.

The sum of n dice can then be expressed as the sum of these indicator variables:

$S = X_1 + X_2 + X_3 + ... + X_n$

Now, we can calculate the expected value of S:

$E(S) = E(X_1) + E(X_2) + E(X_3) + ... + E(X_n)$

Since each die is fair, the probability of each face (1, 2, 3, 4, 5, or 6) appearing on a single die is 1/6, and the expected value of each indicator variable is:

$E(X_i) = 1 * P(X_i = 1) + 0 * P(X_i = 0) = 1 * (1/6) + 0 * (5/6) = 1/6$

Now, you can sum up the expected values of the individual indicators:

$E(S) = E(X_1) + E(X_2) + E(X_3) + ... + E(X_n) = (1/6) + (1/6) + (1/6) + ... + (1/6) = (n/6)$

So, the expected value of the sum of n dice is (n/6).

## 5 Use indicator random variables to solve the following problem, which is known as the hat-check problem. Each of n customers gives a hat to a hat-check person at a restaurant. The hat-check person gives the hats back to the customers in a random order. What is the expected number of customers who get back their own hat?

**Origin** :

**Exercise** 5.2-4

**Page** 122

**Answer** :

Let $X_i$ be an indicator random variable for the i-th customer, where:

$X_i = 1$ if the i-th customer gets their own hat back. $X_i = 0$ if the i-th customer does not get their own hat back.

The probability that the i-th customer gets their own hat back is 1/n, as there are n hats and they are returned in a random order. Therefore, we have:

$E(X_i) = P(X_i = 1) = 1/n$

Now, let's define a random variable Y as the total number of customers who get their own hat back. Y is the sum of the indicator random variables for each customer:

$Y = X_1 + X_2 + X_3 + ... + X_n$

Now, we can find the expected value of Y using linearity of expectation:

$E(Y) = E(X_1 + X_2 + X_3 + ... + X_n)$

Using the linearity of expectation, we can write this as:

$E(Y) = E(X_1) + E(X_2) + E(X_3) + ... + E(X_n)$

Since each customer's indicator random variable has the same expected value (1/n), we can simplify further:

$E(Y) = (1/n) + (1/n) + (1/n) + ... + (1/n)(ntimes)$

$E(Y) = (1/n) * n = 1$

So, the expected number of customers who get back their own hat is 1. This result may be surprising, but it's a classic result of the hat-check problem. On average, one customer is expected to get their own hat back, while the others get hats belonging to other customers.

## 6 Show that the worst-case running time of HEAPSORT is $\Omega(n \lg n)$.

**Origin** :

**Exercise** 6.6-4

**Page** 160

**Answer** :

Basic heapsort:

1. Build a max-heap from the input array.
2. Repeatedly remove the maximum element from the heap and place it at the end of the array, shrinking the heap size.
3. Repeat step 2 until the heap is empty.

The worst-case occurs when the input array is specially ordered, i.e., the largest element is at the beginning, the second largest is at the second position, and so on.

Every time we extract the maximum element from the heap (which is always at the root of the heap), we have to perform the following operations:

1. Swap the maximum element with the last element of the array.
2. Restore the heap property, which involves down-heapifying the heap (sifting down the element that was originally at the end of the array, which is the smallest element in the heap)

After the first element is removed, the second-largest element becomes the new root, and the rest of the array is unsorted. We then need to restore the heap property again and again for each element, resulting in a series of down-heapify operations.

In the worst-case scenario, for each element, we perform a down-heapify operation on a heap of size n, and the number of operations increases as we progress through the array. The first element requires the most operations, the second element requires fewer operations, and so on.

The total number of operations can be shown to be $\Omega(n \lg n)$ because each down-heapify operation takes $\Omega(\lg n)$ time, and we do this for all n elements in the worst-case scenario.

Therefore, in the worst-case scenario, HEAPSORT has a lower bound of $\Omega(n \lg n)$, meaning that its worst-case running time is at least proportional to n lg n.

## 7 Give an $O(n \lg k)$-time algorithm to merge k sorted lists into one sorted list, where n is the total number of elements in all the input lists. (Hint: Use a minheap for k-way merging.)

**Origin** :

**Exercise** 6.5-9

**Page** 166

**Answer** :

1. Create a min-heap and initialize it with the first element from each of the k sorted lists, along with an index indicating the list it came from. The heap will keep track of the smallest element among these k elements. The index is important for knowing which list the element belongs to.
2. Initialize an empty result list to store the sorted elements.
3. Repeat the following steps until the min-heap is empty:
   (a) Extract the minimum element (the smallest among the k elements) from the min-heap. This element came from one of the input lists.
   (b) Add this element to the result list.
   (c) Retrieve the next element from the same input list (the one we just extracted from) and add it to the min-heap. If there are no more elements in that list, do nothing.
4. Continue these steps until the min-heap is empty. The result list will be the merged, sorted list.

## 8 Use the substitution method to prove that the recurrence $T(n) = T(n-1) + \Theta(n)$ has the solution $T(n) = \Theta(n^2)$, as claimed at the beginning of Section 7.2.

**Origin** :

**Exercise** 7.2-1

**Page** 178

**Answer** :

To prove that the recurrence $T(n) = T(n-1) + \Theta(n)$ has the solution $T(n) = \Theta(n^2)$ using the substitution method, we'll first make an educated guess and then use mathematical induction to prove it.

Guess: $T(n) = \Theta(n^2)$

Inductive Hypothesis: We assume that $T(k) = \Theta(k^2)$ for all k ¡ n, where n is a positive integer.

Now, we will prove that $T(n) = \Theta(n^2)$ based on this assumption.

$T(n) = T(n-1) + \Theta(n)$

By our inductive hypothesis, $T(n-1) = \Theta((n-1)^2)$, which we can write as $T(n-1) = \Theta(n^2 - 2n + 1)$.

Now, let's substitute this into the original recurrence:

$T(n) = \Theta(n^2 - 2n + 1) + \Theta(n)$

Since $\Theta(n^2)$ dominates $\Theta(-2n+1)$ and $\Theta(n)$ in terms of growth rates, we can drop the lower-order terms and constants, as they won't affect the asymptotic behavior:

$T(n) = \Theta(n^2)$

So, we've shown that for n, $T(n) = \Theta(n^2)$, assuming $T(k) = \Theta(k^2)$ for all k ¡ n. This completes the proof by induction.

Therefore, the solution to the recurrence $T(n) = T(n-1) + \Theta(n) is T(n) = \Theta(n^2)$.

## 9 Show that the running time of QUICKSORT is $\Theta(n^2)$ when the array A contains distinct elements and is sorted in decreasing order.

**Origin** :

**Exercise** 7.2-3

**Page** 178

**Answer** :

To show that the running time of QUICKSORT is $\Theta(n^2)$ when the array A contains distinct elements and is sorted in decreasing order, we can analyze the worst-case behavior of the algorithm. In this specific scenario, the worst-case behavior occurs when the pivot selection strategy consistently selects the smallest or largest element as the pivot, leading to unbalanced partitioning.

Here's a step-by-step analysis:

1. In QUICKSORT, a pivot element is chosen, and the array is partitioned into two subarrays: elements less than the pivot and elements greater than the pivot.
2. In the case where the array is already sorted in decreasing order, if the pivot selection strategy consistently chooses the largest element as the pivot, then one partition will contain n - 1 elements, and the other partition will contain only 1 element. This results in an unbalanced partition.
3. Consequently, the algorithm makes n - 1 recursive calls to sort the subarray with n - 1 elements and only 1 recursive call to sort the subarray with 1 element.
4. In the next level of recursion, the same situation occurs. The larger partition is further divided into two subarrays, one with n - 2 elements and the other with 1 element, and so on.
5. The recursive calls continue, each time decreasing the size of the larger partition by 1, until the entire array is sorted.

6. The total number of recursive calls made in this scenario is $1 + 2 + 3 + ... + (n - 1)$, which is a sum of the first n - 1 natural numbers. This sum is given by (n - 1)(n)/2.

The number of comparisons made during each level of recursion is proportional to the size of the partition. In this worst-case scenario, the partition sizes are unbalanced, with one partition containing only 1 element. Therefore, the total number of comparisons is also proportional to (n - 1)(n)/2.

Asymptotically, we can conclude that the running time of QUICKSORT in this worst-case scenario, where the array is sorted in decreasing order, is $\Theta(n^2)$.

This demonstrates that when the array contains distinct elements and is sorted in decreasing order, the worst-case running time of QUICKSORT is $\Theta(n^2)$.

## 10 Stack depth for quicksort

**Origin** :

**Problems** 7-4

**Page** 188

**Subject** : The QUICKSORT algorithm of Section 7.1 contains two recursive calls to itself. After QUICKSORT calls PARTITION, it recursively sorts the left subarray and then it recursively sorts the right subarray. The second recursive call in QUICKSORT is not really necessary; we can avoid it by using an iterative control structure. This technique, called *tail recursion*, is provided automatically by good compilers. Consider the following version of quicksort, which simulates tail recursion:

TAIL-RECURSIVE-QUICKSORT(A, p, r)

```
while p<r
 // Partition and sort left subarray.
 q = PARTITION(A,p,r)
 TAIL-RECURSIVE-QUICKSORT(A,p,q-1)
 p=q+1
```

a Argue that TAIL-RECURSIVE-QUICKSORT(A,1,A.length) correctly sorts the array A.

Compilers usually execute recursive procedures by using a *stack* that contains pertinent information, including the parameter values, for each recursive call. The information for the most recent call is at the top of the stack, and the information for the initial call is at the bottom. Upon calling a procedure, its information is *pushed* onto the stack; when it terminates, its information is *popped.*Sincewe assume that array parameters are represented by pointers, the information for each procedure call on the stack requires $O(1)$ stack space. The *stack depth* is the maximum amount of stack space used at any time during a computation.

b Describe a scenario in which TAIL-RECURSIVE-QUICKSORT's stack depth is $\Theta(n)$ on an n-element input array.

c Modify the code for TAIL-RECURSIVE-QUICKSORT so that the worst-case stack depth is $\Theta(\lg n)$. Maintain the $O(n \lg n)$ expected running time of the algorithm.

## 11 Describe an algorithm that, given n integers in the range 0 to k, preprocesses its input and then answers any query about how many of the n integers fall into a range [a...b] in O(1) time. Your algorithm should use $\Theta(n + k)$ preprocessing time.

**Origin** :

**Exercise** 8.2-4

**Page** 197

**Answer** :

To solve the problem of preprocessing n integers in the range 0 to k and answering queries about how many of the n integers fall into a given range [a...b] in O(1) time, you can use a technique called "prefix sum" or "cumulative sum" along with some preprocessing. Here's the algorithm:

1. Create an array **counts** of size k + 1, initially filled with all zeros. This array will be used for preprocessing.

2. Preprocessing ($\Theta(n)$ time):

   - For each integer x in the input list, increment **counts[x]** by 1. This step will count the occurrences of each integer in the range 0 to k.
   - Next, compute the cumulative sum array **cumulative** from the **counts** array. Initialize **cumulative[0]** with **counts[0]**, and for each i from 1 to k, set **cumulative[i] = cumulative[i-1] + counts[i]**. This step helps us obtain the cumulative count of integers less than or equal to i.

3. Query (O(1) time):

   - To answer a query about how many integers fall into the range [a...b], use the cumulative sum array: **result = cumulative[b] - cumulative[a-1]**. This is a constant-time operation as you only need to perform two array lookups and subtraction.

4. This algorithm has a preprocessing time of $\Theta(n + k)$ and can answer queries in O(1) time, making it efficient for this specific problem. It effectively utilizes the cumulative sum array to provide quick answers to range count queries without reevaluating the counts for each query.

## 12 Show that the second smallest of n elements can be found with $n+\lceil\lg n\rceil-2$ comparisons in the worst case. (Hint: Also find the smallest element.)

**Origin** :

**Exercise** 9.1-1

**Page** 215

**Answer** :

To find the second smallest of n elements and the smallest element using $(n + \lceil\lg n\rceil - 2)$ comparisons in the worst case, you can use a modified version of the "tournament" method. This approach requires a total of $n + \lceil\lg n\rceil - 2$ comparisons in the worst case.

Here's how it works:

1. Divide the n elements into pairs. Compare each element in each pair and keep track of the smaller element in each pair. This step requires n/2 comparisons.

2. Continue dividing the winners of each pair into pairs and comparing them until you have just one element left. This will take $\lceil\lg n\rceil - 1$ more rounds of comparisons.

3. At the end of this process, you will have the smallest element (found after n/2 comparisons) and $\lceil\lg n\rceil - 1$ other elements.

4. Compare these $\lceil\lg n\rceil - 1$ elements to find the second smallest element. This step requires $\lceil\lg n\rceil - 1$ comparisons.

The total number of comparisons required is $(n/2) + (\lceil\lg n\rceil - 1) = n/2 + \lceil\lg n\rceil - 1$, which is equivalent to $n + \lceil\lg n\rceil - 2$. Therefore, you can find the second smallest element with $(n + \lceil\lg n\rceil - 2)$ comparisons in the worst case.

## 13 In the algorithm SELECT, the input elements are divided into groups of 5. Will the algorithm work in linear time if they are divided into groups of 7? Argue that SELECT does not run in linear time if groups of 3 are used.

**Origin** :

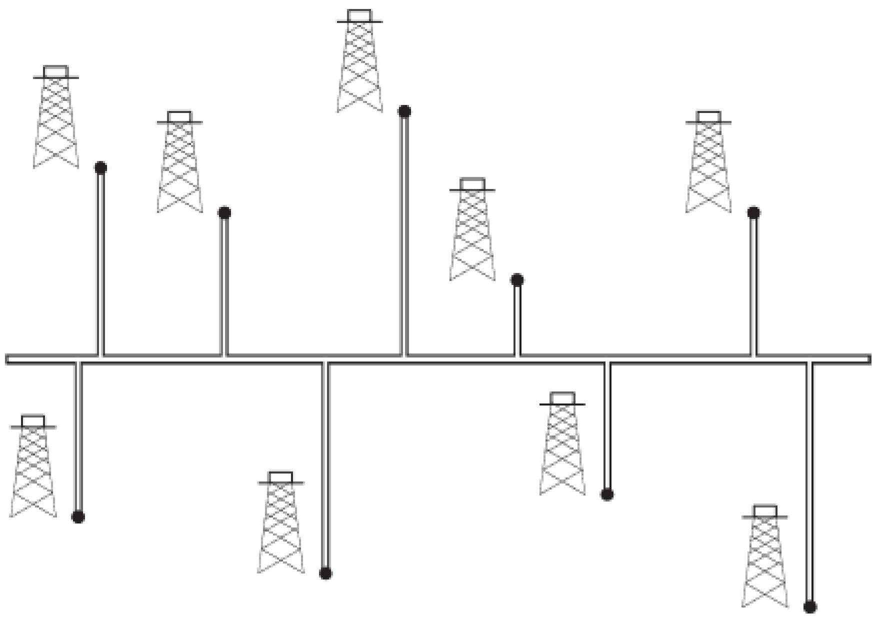

Figure 3: Professor Olay needs to determine the position of the east-west oil pipeline that minimizes the total length of the north-south spurs.

**Exercise** 9.3-1

**Page** 223

**Answer** : Answer

## 14 Professor Olay is consulting for an oil company, which is planning a large pipeline running east to west through an oil field of n wells. The company wants to connect a spur pipeline from each well directly to the main pipeline along a shortest route (either north or south), as shown in Figure 3. Given the x-andy-coordinates of the wells, how should the professor pick the optimal location of the main pipeline, which would be the one that minimizes the total length of the spurs? Show how to determine the optimal location in linear time.

**Origin** :

**Exercise** 9.3-9

**Page** 223

**Answer** :
Answer