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.

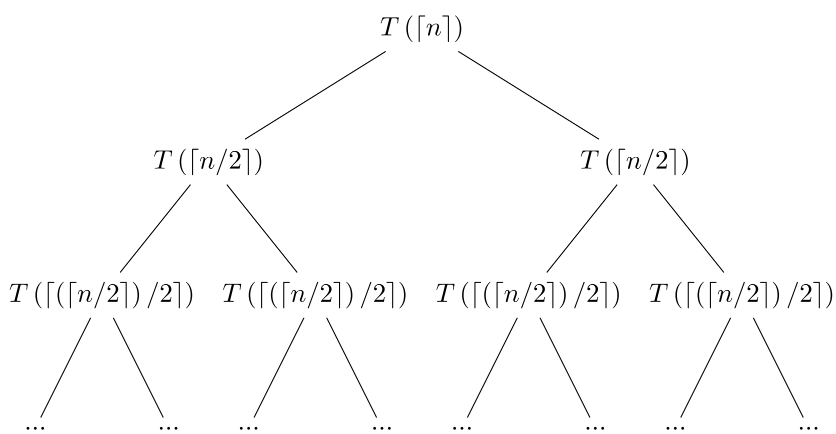

Figure 1: Recursion Tree.

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):

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

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.

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):

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

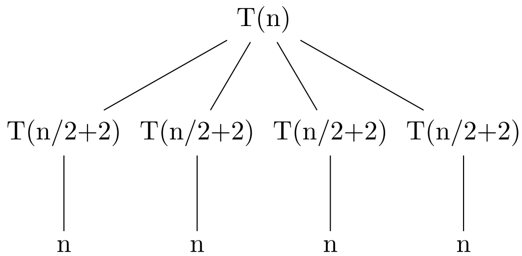

Figure 2: Recursion Tree.

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, $n/(2^k) + 2k = C for some constant C$, 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):

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

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 : Answer

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 : Answer

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 : Answer

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.