Problems 2

LeeJX

April 8, 2017

2-1 a. The problem T(n) is divided into (n/k) subquestions in T(k), each subquestion is in $O(k^2)$, so the total insertion sort cost $O(n/k*k^2) = O(nk)$

b. Such as the Figure 2.5, there are lg(n/k) levels, each level costs cn, the merge time is $O(nlg(n/k))$

c. The merge sort costs$O(nlgn)$, the given algorithm costs$O(nk+nlg(n/k)$, $O(nlg(n/k))$ is smaller than $O(nlgn)$ for any $k >= 1$, so the keynote is the relation between $nlgn$ and nk, thus $k = O(lgn)$

d. Consider the length of subquestions.

reference CSDN

2-2 a. Each element in array A is in A'.

b. this **for** loop is to make the A[j] the smallest element.

Initialization: $j = A.length$ begin at the last element, right

Maintenance: exchange if $A[j] < A[j-1]$ that makes A[j-1] the smallest in A[j...length].

Termination: the loop end at the time when $j < i+1$, that is $j = i$, and A[j] is smallest.

2-3 a. $O(n)$

b. The problem should be explained like this: The original problem is to compute the sum of a polynomial equation,

$$P(x) = \sum_{k=0}^{n} a_k x^k = a_0 + a_1 x + a_2 x^2 + ... + a_n x^n$$

the Horner's rule simplifies this problem using the equation:

$P(x) = a_0 + x(a_1 + x(a_2 + ... + x(a_{n-1} + xa_n)...))$

the naive polynomial-evaluation

sum = a_0

for i = 1 upto n

  sum = sum + a[i] * x

  x = x * x

the naive polynomial-evaluation takes one more computation.

c. prove the given equation is equal to the original problem.

Initialization: in the pseudocode, i is begin at n. If $i = n$, in the given equation, $y = \sum_{k=0}^{-1} = 0$, which is the same as pseudocode.

Maintenance: If $i = n - 1$, in the given equation, $y = \sum_{k=0}^{0} a_{k+n}x^k = a_n$. In the pseudocode, line 3 has been computed once, when $i = n - 1$, and $y = a_n + x * 0 = a_n$. They are the same.

Termination: The pseudocode end at the time when $i < 0$, that is $i = -1$. If $i = -1$, in the given equqtion, $y = \sum_{k=0}^{n} a_k x^k$ is the same with the original problem.

d. The Horner's rule is a deformation of the original problem in mathematics, it extracts the common factor.

2-4 a. (2,1) (3,1) (8,6) (8,1) (6,1)

b. the inverted sequence, has $\sum_{k=2}^{n} k - 1$

c. the inversions only influence the step in insertion sort choosing the exact location to insert in. The inversions increase the times of compare.

d. In the function merge, assume we merge array L[] and array R[], if $L[i] < R[j]$ then result array $A[k] = L[i]$, else $A[k] = R[j]$. At this step, $A[k] = R[j]$ means there is an inversion pair (R[j],L[i]), so remember all such situation, and we can get the number of inversions. This cost no more than $O(nlgn)$.

reference CSDN