

CSE 201

Homework Assignment 4 Solutions

1. Assume the correctness of the algorithm $\text{Partition}(A, p, r)$ on page 171 of the text. Use induction to prove the correctness of $\text{QuickSort}(A, p, r)$ on page 171.

Proof:

We reproduce $\text{QuickSort}()$ here for reference.

$\text{QuickSort}(A, p, r)$

1. if $p < r$
2. $q = \text{Partition}(A, p, r)$
3. $\text{QuickSort}(A, p, q - 1)$
4. $\text{QuickSort}(A, q + 1, r)$

The call to $\text{Partition}(A, p, r)$ is assumed to re-arrange $A[p \cdots r]$ and return an index q in the range $p \leq q \leq r$ such that $A[p \cdots (q - 1)] \leq A[q] < A[(q + 1) \cdots r]$. We proceed by induction on the length $m = r - p + 1$ of $A[p \cdots r]$.

- I. If $m = 1$, then $r = p$ and the test on line (1) is false, so $\text{QuickSort}()$ returns with no change to A . Indeed, an array of length 1 is already sorted, so no changes are needed.
- II. Let $m > 1$ and assume that $\text{QuickSort}()$ correctly sorts any subarray of length less than m . We must show that $\text{QuickSort}()$ correctly sorts any subarray of length $m = r - p + 1$. Now $m > 1$ implies $p < r$, so the test on line (1) is true and lines (2) through (4) are executed. Our assumption on Partition says that $A[p \cdots (q - 1)] \leq A[q] < A[(q + 1) \cdots r]$ is true after line (2), where $p \leq q \leq r$. Observe

$$\text{length}(A[p \cdots (q - 1)]) = (q - 1) - p + 1 < r - p + 1 = m$$

and

$$\text{length}(A[(q + 1) \cdots r]) = r - (q + 1) + 1 < r - p + 1 = m.$$

The induction hypothesis implies that lines (3) and (4) correctly sort $A[p \cdots (q - 1)]$ and $A[(q + 1) \cdots r]$. The inequality $A[p \cdots (q - 1)] \leq A[q] < A[(q + 1) \cdots r]$ now implies that the subarray $A[p \cdots r]$ is sorted, as required. ■

2. Recall the n^{th} harmonic number was defined to be $H_n = \sum_{k=1}^n \left(\frac{1}{k}\right) = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n-1} + \frac{1}{n}$. Use induction to prove that

$$\sum_{k=1}^n kH_k = \frac{1}{2}n(n+1)H_n - \frac{1}{4}n(n-1)$$

for all $n \geq 1$. (Hint: Use the fact that $H_n = H_{n-1} + \frac{1}{n}$.)

Proof:

- I. If $n = 1$ we have $\sum_{k=1}^1 kH_k = 1 \cdot H_1 = 1$ and $\frac{1}{2} \cdot 1 \cdot (1+1)H_1 - \frac{1}{4} \cdot 1 \cdot (1-1) = 1$, so the base case is satisfied.
- II. Let $n \geq 1$ and assume that $\sum_{k=1}^n kH_k = \frac{1}{2}n(n+1)H_n - \frac{1}{4}n(n-1)$ holds. We must show that $\sum_{k=1}^{n+1} kH_k = \frac{1}{2}(n+1)(n+2)H_{n+1} - \frac{1}{4}(n+1)n$ is also true.

$$\begin{aligned} \sum_{k=1}^{n+1} kH_k &= \sum_{k=1}^n kH_k + (n+1)H_{n+1} \\ &= \frac{1}{2}n(n+1)H_n - \frac{1}{4}n(n-1) + (n+1)H_{n+1} \quad (\text{by the induction hypothesis}) \\ &= \frac{1}{2}n(n+1) \left\{ H_{n+1} - \frac{1}{n+1} \right\} - \frac{1}{4}n(n-1) + (n+1)H_{n+1} \quad (\text{using the hint}) \\ &= \left(\frac{n}{2} + 1 \right) (n+1)H_{n+1} - \frac{1}{2} \cdot n - \frac{1}{4} \cdot n(n-1) \\ &= \left(\frac{n+2}{2} \right) (n+1)H_{n+1} - \left(\frac{n}{2} + \frac{n^2}{4} - \frac{n}{4} \right) \\ &= \frac{1}{2} \cdot (n+1)(n+2)H_{n+1} - \frac{n^2+n}{4} \\ &= \frac{1}{2} \cdot (n+1)(n+2)H_{n+1} - \frac{1}{4} \cdot (n+1)n, \end{aligned}$$

as required. ■

3. Use the results of problem #2 above, and problem #3 on the Midterm 1 solutions to show, by direct substitution, that the solution to the recurrence

$$(*) \quad t(n) = (n-1) + \frac{2}{n} \cdot \sum_{k=1}^{n-1} t(k)$$

is given by: $t(n) = 2(n+1)H_n - 4n$.

Proof:

We substitute $t(n) = 2(n+1)H_n - 4n$ into the left and right hand sides of the recurrence relation (*) to obtain an identity for all $n \geq 1$. Along the way we use the identities

$$(**) \quad \sum_{k=1}^n kH_k = \frac{1}{2}n(n+1)H_n - \frac{1}{4}n(n-1)$$

and

$$(***) \quad \sum_{k=1}^n H_k = (n+1)H_n - n$$

$$\begin{aligned}
\text{RHS} &= (n-1) + \frac{2}{n} \cdot \sum_{k=1}^{n-1} t(k) \\
&= (n-1) + \frac{2}{n} \cdot \sum_{k=1}^{n-1} [2(k+1)H_k - 4k] \\
&= (n-1) + \frac{4}{n} \cdot \sum_{k=1}^{n-1} kH_k + \frac{4}{n} \cdot \sum_{k=1}^{n-1} H_k - \frac{8}{n} \cdot \sum_{k=1}^{n-1} k \\
&= (n-1) + \frac{4}{n} \cdot \left(\frac{1}{2}n(n-1)H_{n-1} - \frac{1}{4}(n-1)(n-2) \right) \\
&\quad + \frac{4}{n} \cdot (nH_{n-1} - (n-1)) - \frac{8}{n} \cdot \left(\frac{n(n-1)}{2} \right) \quad \text{by ** and ***} \\
&= (n-1) + 2(n-1)H_{n-1} - \frac{(n-1)(n-2)}{n} + 4H_{n-1} - \frac{4(n-1)}{n} - 4(n-1) \\
&= -3(n-1) + (2n-2+4)H_{n-1} - \frac{(n-2+4)(n-1)}{n} \\
&= -3(n-1) + 2(n+1)H_{n-1} - \frac{(n+2)(n-1)}{n} \\
&= 2(n+1) \left(H_n - \frac{1}{n} \right) - \frac{3n(n-1) + (n+2)(n-1)}{n} \\
&= 2(n+1)H_n - \frac{2(n+1) + 3n(n-1) + (n+2)(n-1)}{n} \\
&= 2(n+1)H_n - \frac{2n+2+3n^2-3n+n^2+2n-n-2}{n} \\
&= 2(n+1)H_n - 4n = t(n) = \text{LHS} \quad \blacksquare
\end{aligned}$$

4. Design a recursive algorithm called $\text{Extrema}(A, p, r)$ that, given an array $A[1 \cdots n]$ finds and returns both the min and max of the subarray $A[p \cdots r]$ as an ordered pair: $(\min(A[p \cdots r]), \max(A[p \cdots r]))$. Your algorithm should perform exactly $\lceil 3n/2 \rceil - 2$ array comparisons on an input array of length n . (Hint: Section 9.1 of the text describes an iterative algorithm that does this.)

a. Write your algorithm in pseudo-code.

Solution:

$\text{Extrema}()$ will call the following 3 subroutines, whose correctness is taken as obvious.

$\min(a, b)$ (returns the smaller of a and b)

1. return $(a < b) ? a : b$

$\max(a, b)$ (returns the larger of a and b)

1. return $(a < b) ? b : a$

$\text{order}(a, b)$ (returns the pair (a, b) in increasing order)

1. return $(a < b) ? (a, b) : (b, a)$

Note each of the above functions performs exactly one comparison.

$\text{Extrema}(A, p, r)$ (pre: $p \leq r$)

```

1. if  $p = r$ 
2.   return  $(A[p], A[p])$ 
3. else if  $p + 1 = r$ 
4.   return  $\text{order}(A[p], A[p + 1])$ 
5. else
6.    $(m_1, M_1) = \text{order}(A[p], A[p + 1])$ 
7.    $(m_2, M_2) = \text{Extrema}(A, p + 2, r)$ 
8.   return  $(\min(m_1, m_2), \max(M_1, M_2))$ 

```

■

- b. Prove the correctness of your algorithm by induction on $m = r - p + 1$, the length of the subarray $A[p \cdots r]$.

Proof:

- I. If $m = 1$, then $p = r$ and the test on line (1) is true, so line (2) is executed. Indeed, in this case both the maximum and minimum of this one element array is $A[p]$. If $m = 2$, we have $p + 1 = r$, so the test on line (1) is false and that on line (3) is true. The subroutine call $\text{order}(A[p], A[p + 1])$ returns an ordered pair consisting of the minimum and maximum (in that order) of the subarray $A[p, p + 1]$. The two base cases $m = 1$ and $m = 2$ are therefore satisfied.
- II. Let $m > 2$ and assume that $\text{Extrema}()$, when called on any subarray of length less than m , returns an ordered pair consisting of the minimum and maximum of the subarray, in that order. We must show that if $m = \text{length}(A[p \cdots r])$, then $\text{Extrema}(A, p, r)$ correctly finds and returns the ordered pair $(\min(A[p \cdots r]), \max(A[p \cdots r]))$. Since $m > 2$, the tests on lines (1) and (3) are false, and lines (6) through (8) are executed. Line (6) assigns the minimum and maximum of $A[p, p + 1]$ to m_1 and M_1 , respectively. Also, since

$$\text{length}(A[(p+2) \cdots r]) = r - (p+2) + 1 = (r - p + 1) - 2 = m - 2 < m,$$

the induction hypothesis guarantees that line (7) assigns the minimum and maximum of the subarray $A[(p+2) \cdots r]$ to m_2 and M_2 , respectively. The minimum and maximum of $A[p \cdots r]$ are therefore $\min(m_1, m_2)$ and $\max(M_1, M_2)$, respectively, which is exactly the ordered pair returned on line (8), as required. ■

- c. Write a recurrence for the number of comparisons performed on $A[1 \cdots n]$, and show that $T(n) = \lceil 3n/2 \rceil - 2$ is the solution.

Solution:

We have the recurrence

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 1 & \text{if } n = 2 \\ T(n-2) + 3 & \text{if } n > 2 \end{cases}$$

Observe that the function $T(n) = \lceil 3n/2 \rceil - 2$ satisfies $T(1) = 0$ and $T(2) = 1$. When $n > 2$ we have

$$\begin{aligned} \text{RHS} &= T(n-2) + 3 \\ &= \left(\left\lceil \frac{3(n-2)}{2} \right\rceil - 2 \right) + 3 \\ &= \left\lceil \frac{3n}{2} - 3 \right\rceil + 1 \\ &= \left\lceil \frac{3n}{2} \right\rceil - 3 + 1 \\ &= \left\lceil \frac{3n}{2} \right\rceil - 2 \\ &= T(n) \\ &= \text{LHS} \end{aligned}$$

so that $T(n) = \lceil 3n/2 \rceil - 2$ satisfies the above recurrence. ■

5. (This is a re-wording of problem 4.2-4 on page 82 of CLRS 3rd edition.) Determine the largest integer k such that if there is a way to multiply 3×3 matrices using k multiplications (not assuming commutativity of multiplication of matrix elements), then there is an algorithm to multiply $n \times n$ matrices (where n is an exact power of 3) in time $O(n^{\lg(7)})$. What would the run time of this algorithm be? (Hint: Proceed as in the analysis of Strassen's algorithm, but recur on submatrices of size $\frac{n}{3} \times \frac{n}{3}$.)

Solution:

We regard an $n \times n$ square matrix (where n is a power of 3) as a 3×3 matrix, each of whose 9 elements is a square submatrix of size $\frac{n}{3} \times \frac{n}{3}$. Thus, to multiply two $n \times n$ square matrices, we need to multiply two 3×3 matrices of matrices. We are to assume that this multiplication can be done by performing only k multiplications of the underlying $\frac{n}{3} \times \frac{n}{3}$ submatrices (which are non-commutative operations). Presumably k is less than the 27 multiplications ordinarily needed for multiplying two 3×3 matrices. However, k must also be at least 9 since that is the number of $\frac{n}{3} \times \frac{n}{3}$ submatrices in the answer, and each requires at least 1 multiplication to compute. Since n is a power of 3, we can recur on this process down to matrices of size 1×1 , where the recursion halts and the product of real numbers is returned. At each level then, there are k recursive multiplications of 9 matrices whose size is $1/3^{\text{rd}}$ the size of the current level.

Let $T(n)$ denote the running time of the algorithm described above. Then, in analogy with Strassen's algorithm, we obtain the recurrence relation

$$T(n) = kT(n/3) + \Theta(1),$$

where the first term is the cost of the k recursive calls, and the second term represents the overhead of the current invocation of the algorithm. Case 1 of the Master Theorem gives the asymptotic solution to this recurrence as

$$T(n) = \Theta(n^{\log_3(k)}).$$

Alternatively, if we included the cost of the matrix additions necessary to compute the product, we would have the recurrence $T(n) = kT(n/3) + \Theta(n^2)$, which has the same asymptotic solution as above (since $k > 9$).

We therefore seek the largest integer k satisfying $n^{\log_3(k)} = O(n^{\lg(7)})$. Equivalently $\log_3(k) < \log_2(7)$, and hence

$$k < 3^{\log_2(7)} = 21.849 \dots$$

Therefore $k = 21$, and the run time of the above algorithm is $T(n) = \Theta(n^{\log_3(21)})$. ■

Remark:

Observe that nothing in this solution proves, or even suggests, that such an algorithm exists, since we know of no way to multiply two 3×3 matrices of matrices by computing only $k = 21$ submatrix products (and likewise for any other $k < 27$.) All it shows is that if such a method were to exist, we would need $k \leq 21$ to yield an algorithm better than Strassen's. ■

6. (This is a slight re-wording of problem 8-5 on page 207 of CLRS 3rd edition.)

Suppose that, instead of sorting an array, we just require that the elements increase on average. More precisely, we call an n -element array $A[1 \cdots n]$ **k -sorted** if for all i in the range $1 \leq i \leq n - k$, the following holds:

$$\frac{\sum_{j=i}^{i+k-1} A[j]}{k} \leq \frac{\sum_{j=i+1}^{i+k} A[j]}{k}$$

a. What does it mean for an array to be 1-sorted?

Solution:

A 1-sorted array is simply sorted in the usual sense. Indeed, substituting $k = 1$ into the definition yields

$$\frac{\sum_{j=i}^i A[j]}{1} \leq \frac{\sum_{j=i+1}^{i+1} A[j]}{1} \quad \text{for } 1 \leq i \leq n - 1$$

which says $A[i] \leq A[i + 1]$ for all i in the range $1 \leq i \leq n - 1$. ■

b. Give a permutation of the numbers $\{1, 2, 3, \dots, 10\}$ that is 2-sorted, but not sorted.

Solution:

Let $A = (1, 6, 2, 7, 3, 8, 4, 9, 5, 10)$. Then the consecutive 2-element averages are:

$$\begin{aligned} (1 + 6)/2 &= 3.5 \\ (6 + 2)/2 &= 4 \\ (2 + 7)/2 &= 4.5 \\ (7 + 3)/2 &= 5 \\ (3 + 8)/2 &= 5.5 \\ (8 + 4)/2 &= 6 \\ (4 + 9)/2 &= 6.5 \\ (9 + 5)/2 &= 7 \\ (5 + 10)/2 &= 7.5 \end{aligned}$$

Therefore A is 2-sorted. ■

c. Prove that an n -element array is k -sorted if and only if $A[i] \leq A[i + k]$ holds for all i in the range $1 \leq i \leq n - k$.

Proof:

By definition, $A[1 \cdots n]$ is k -sorted if and only if

$$\frac{\sum_{j=i}^{i+k-1} A[j]}{k} \leq \frac{\sum_{j=i+1}^{i+k} A[j]}{k} \quad \text{for all } 1 \leq i \leq n - k,$$

$$\text{iff } \sum_{j=i}^{i+k-1} A[j] \leq \sum_{j=i+1}^{i+k} A[j] \quad \text{for all } 1 \leq i \leq n - k,$$

$$\text{iff } A[i] + \sum_{j=i+1}^{i+k-1} A[j] \leq \sum_{j=i+1}^{i+k-1} A[j] + A[i + k] \quad \text{for all } 1 \leq i \leq n - k,$$

$$\text{iff } A[i] \leq A[i + k] \quad \text{for all } 1 \leq i \leq n - k,$$

as claimed. ■

- d. Describe an algorithm that k -sorts an n -element array in time $\Theta(n \log n)$.

Solution:

Partition $A[1 \cdots n]$ into k (non-contiguous) sub-arrays, each of length at most $\lceil n/k \rceil$, as follows. Select every k^{th} element in A , starting at index i , for $i = 1, 2, \dots, k$, and call that subarray A_i . Thus

$$\begin{aligned} A_1 &= (A[1], A[1+k], A[1+2k], A[1+3k], \dots) \\ A_2 &= (A[2], A[2+k], A[2+2k], A[2+3k], \dots) \\ A_3 &= (A[3], A[3+k], A[3+2k], A[3+3k], \dots) \\ &\vdots \\ A_k &= (A[k], A[2k], A[3k], A[4k], \dots) \end{aligned}$$

Sort each of these sub-arrays using a $\Theta(n \log(n))$ sorting algorithm (such as merge sort or heap sort). Finally, reassemble the elements of the arrays $A_1, A_2, A_3, \dots, A_k$ into the original array object A by placing one element from each A_i into A , in order, until all elements in all A_i are exhausted. Thus $A[1 \cdots n]$ now consists of

$$A = (A_1[1], A_2[1], \dots, A_k[1], A_1[2], A_2[2], \dots, A_k[2], A_1[3], A_2[3], \dots, A_k[3], \dots)$$

Since each A_i is sorted, we have $A[i] \leq A[i+k]$ for all i in the range $1 \leq i \leq n-k$. By part (c) above, the full array is k -sorted.

To accomplish the partitioning and reassembly, we can avoid the costly operation of allocating new array objects by sorting the (non-contiguous) subarrays of A in-place, relying on index calculations to step through each subarray. Even if we do create new array objects $A_1, A_2, \dots$, etc., the partition and reassemble steps involve no basic operations (array comparisons), and so can be ignored in the run time analysis. The total cost $T(n)$ of this algorithm is therefore the aggregate cost of the individual sorts, and hence

$$\begin{aligned} T(n) &= k \cdot \Theta\left(\left\lceil \frac{n}{k} \right\rceil \log \left\lceil \frac{n}{k} \right\rceil\right) \\ &= \Theta\left(k \left(\frac{n}{k}\right) \log \left(\frac{n}{k}\right)\right) \\ &= \Theta(n(\log n - \log k)) \\ &= \Theta(n \log n). \end{aligned}$$

■

Remarks

We illustrate the above algorithm by performing several k -sorts ($k = 2, 3, 4$) on the following array of length $n = 10$: $A = (5, 3, 8, 10, 1, 6, 2, 9, 4, 7)$

$$\begin{array}{ll} k = 2: & \text{sort} \\ A_1 = (5, 8, 1, 2, 4) & \rightarrow (1, 2, 4, 5, 8) \\ A_2 = (3, 10, 6, 9, 7) & \rightarrow (3, 6, 7, 9, 10) \end{array}$$

$$2\text{-sorted: } A = (1, 3, 2, 6, 4, 7, 5, 9, 8, 10)$$

$k = 3$: sort
 $A_1 = (5, 10, 2, 7) \rightarrow (2, 5, 7, 10)$
 $A_2 = (3, 1, 9) \rightarrow (1, 3, 9)$
 $A_3 = (8, 6, 4) \rightarrow (4, 6, 8)$

3-sorted: $A = (2, 1, 4, 5, 3, 6, 7, 9, 8, 10)$

$k = 4$: sort
 $A_1 = (5, 1, 4) \rightarrow (1, 4, 5)$
 $A_2 = (3, 6, 7) \rightarrow (3, 6, 7)$
 $A_3 = (8, 2) \rightarrow (2, 8)$
 $A_4 = (10, 9) \rightarrow (9, 10)$

4-sorted: $A = (1, 3, 2, 9, 4, 6, 8, 10, 5, 7)$

Observe that for each k , and all i in the range $1 \leq i \leq k$, we have $\text{length}(A_i) \leq \lceil 10/k \rceil$.

There is also a very simple and clever way to alter the Quicksort algorithm so as to perform a k -sort. We leave this alteration as an exercise. ■