Homework No. 2 (Solutions)

CSCI 4470/6470 Algorithms, CS@UGA, Fall 2017

Due Monday September 25, 2017

There are 6 questions, in which Q5(2) and Q6(2) are for graduate students only. Each question is worth 20 points. There are 120 points in total for graduate students and 100 for undergraduate students.

1. Consider the following problem to search a key from a sorted \( n \times n \) matrix, where all rows and columns of elements in the matrix are sorted. Design an algorithm for this problem using a divide-and-conquer strategy to achieve a non-trivial upper bound (for example, \( O(n^2) \) is a trivial upper bound).

**Answers:**

The following trivial methods, which would deserve no credit:

(1) sequential search, which basically search through all \( n^2 \) elements;
(2) binary search is applied on all columns (or all rows), resulting in time \( O(n \log_2 n) \).

In addition, some students may show the following strategy:

**Searching from the top-right corner cell in the matrix.** If the cell element < key, then move to the left-ward; if the cell element > key, then move down-ward; then repeat until hitting the left wall or bottom wall.

Though this strategy may result in a good upper bound \( O(n) \) but is not exactly a divide-and-conquer method. The student only gets one half of the credit, provided that all details are correct.

The following are two division-and-conquer methods.
**Method 1.** Pick the “center” element to compare with the key. Terminate if they match. Otherwise, if the key is smaller than the center element, the bottom-right quarter sub-matrix can be discarded; else the top-left quarter sub-matrix can be discarded, in the next round of recursion. Therefore, in the worst case, the process recursively searches for three $\frac{n}{2} \times \frac{n}{2}$ sub-squares, leading the complexity recurrence

$$T(n) = 3T\left(\frac{n}{2}\right) + c$$

The analysis should lead to $O(n^{\log_2 3})$ upper bound. The student will get the full credit if both the algorithm description and a upper bound proof is correct.

**Method 2.** Use binary search on the main diagonal (from top-left to bottom-right) to identify either the element = key, or two elements $x$ and $y$ neighboring on the diagonal such that $x < \text{key}$ and $y > \text{key}$. In the latter case, the matrix is partitioned into 4 sub-matrices, defined by the position of $x$. The next phase of recursion only need to search in the two sub-matrices, excluding the upper-left and lower-right sub-matrices.

The critical part is to estimate the size of the two sub-matrices to be recursively search to get a non-trivial upper bound, which can be done as follows:

Assume the upper-left sub-matrix has dimensions $r \times r$. Then the two sub-matrices to be recursively searched both have the dimensions $r \times (n - r)$ (not necessarily a square matrix). Define the complexity upper bound for such searching on a matrix of dimensions $m \times n$ to be $T(m, n)$. Then

$$T(n, n) \leq 2 \times \max_{1 \leq r < n} T(r, n - r) + \log_2 n = 2T\left(\frac{n}{2}, \frac{n}{2}\right) + \log_2 n$$

because the size $r(n - r)$ is the maximum when $r = \frac{n}{2}$.

One can prove that $T(n, n) = O(n)$. To do that, one can guess

$$T(n, n) \leq cn - b \log_2 n$$

for some constant $c$

which can be proved by the substitution method as follows.

Assume $T\left(\frac{n}{2}, \frac{n}{2}\right) \leq c\frac{n}{2} - b \log_2 \frac{n}{2}$. Then

$$T(n, n) \leq 2T\left(\frac{n}{2}, \frac{n}{2}\right) + \log_2 n \leq 2c\frac{n}{2} - 2b \log_2 \frac{n}{2} + \log_2 n = cn - (2b \log_2 n - \log_2 n - 2b)$$

By choosing $b = 1$, we have $(2b \log_2 n - \log_2 n - 2b) = \log_2 n - 2$, and

$$T(n, n) \leq cn - (\log_2 n - 2) \leq cn$$
where the last inequality holds when $\log_2 n \geq 2$ or $n \geq 4$.

Base case: when $n = 4$, we can search for a key in a $4 \times 4$ matrix using 16 comparisons (simply brute force). For inequality $T(4, 4) = 16 \leq c \times 4$ to hold, we choose $c = 4$.

As the idea is not trivial, the student deserves 75% points for the divide-and-conquer method and 25% points for the proof of the upper bound.
2. Using a decision tree method to prove that the problem of searching for a key in a given $n \times n$ matrix has time complexity lower bound $\Omega(\log_2 n)$ for the comparison-based model, regardless the given matrix is sorted or not. Hint: consider the number of distinct outcomes that any search algorithm would have to accommodate.

**Answers:**

The basic idea is to consider the number of distinct outcomes of the search that a search algorithm would have to accommodate. There are $n^2 + 1$ possible outcomes. So the corresponding decision tree would have at least $n^2 + 1$ leaves, and thus the longest path on such a tree would have length $\geq \log_2(n^2 + 1) \geq \log_2 n^2 = 2 \log_2 n$. So the running time is at least $\Omega(\log_2 n)$.

Points are broken down as:
- 10 points for the $n^2 + 1$ outcomes;
- 5 points for the connection to number of leaves;
- 5 points for the relationship between the leaves and height.
3. Consider to sort a list of binary bits (i.e., each element in the input list is a binary bit, either 0 or 1). What is your fastest comparison-based algorithm for this problem? What is a lower bound for the problem that you can get? Justify your answers.

Answers:

A comparison-based search algorithm cannot directly compare an element $x_i$ to 0 or 1. There are two parts in this problem.

(1) [10 points] Upper bound: the student cannot resort to an existing sorting algorithm. But any new idea to sort such lists would be okay. For example, the partition function used by the quick sort algorithm would be okay. Ideally, the student would get an algorithm of upper bound $O(n)$.

(2) [10 points] Lower bound: Since a sorted list would look like

$$0, \ldots, 0, 1, \ldots, 1$$

where the “boundary position” between 0’s and 1’s could be in one of the $n$ positions, any algorithm would have to accommodate $n$ possible outcomes. So an immediate lower bound would be $\log_2 n$ using the decision tree method.

If a student proves a lower bound based on the assumption that there are $2^n$ types of lists of length $n$, and concludes that $2^n$ should be the number of outcomes, he/she should not get the full credit (10 points).

Note: some students may have mistaken part (2) as finding the lower bound for his/her algorithm. Should that be the case, it deserves the full credit (10 points) if analysis is correct.
4. Consider the Select that finds the $k$th smallest element in a given set. If, instead of making groups of 5 elements, we make groups of 3 elements, would the algorithm still run in linear time? How about groups of 7 elements? Justify your answers.

**Answers:**

This exercise evaluates if a student understands the deterministic Select algorithm. He/she needs to justify his/her answers by showing the time function recurrence situations.

(1) [10 points] Group of 3 elements: the answer is 'NO'. Time function has the recurrence

$$T(n) \leq T\left(\frac{n}{3}\right) + T\left(n - 2 \times \frac{n}{6}\right) + cn = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + cn$$

which does not have the advantage to remove a fraction of the list before the next phase of recursion, thus not resulting in a linear time.

(2) [10 points] Group of 7 elements: the answer is 'YES'. This is because

$$T(n) = T\left(\frac{n}{7}\right) + T\left(n - 4 \times \frac{n}{14}\right) + cn = T\left(\frac{n}{7}\right) + T\left(\frac{5n}{7}\right) + cn$$

which has the advantage to remove a fraction $\frac{1}{7}$ of the list before the next phase of recursion, resulting in a linear time.
5. How to make a quick sort algorithm to run in the worst case time $O(n \log_2 n)$? The question is actually to ask you to find a good partition function for the quick sort.

(1) Give such an algorithm;

(2) \textbf{(Graduates only, bonus for undergraduates)} If the elements in the input list always have only $k$ distinct values, for $k < n$, where $n$ is the number of elements in the list, prove that the quick sort problem can be solved in the worst case time $O(n \log_2 k)$. \textit{Hint:} you may have to formulate time complexity in terms of $T(n, k)$ and find a recurrence before proving that $T(n, k) \leq cn \log_2 k$.

\textbf{Answers:}

(1) [10 points] The student will need to use the Select algorithm that can deterministically find the median of the list of elements in linear time. Using the median as the pivot to partition the list for the quick sort allows one to obtain the following recurrence for the worst case time:

$$T(n) = 2T\left(\frac{n}{2}\right) + cn$$

that has been proved to be $O(n \log_2 n)$.

(2) [10 points, for graduates only, bonus for undergraduates]

Still using the method described in (1) to identify a pivot that can always balance-partition the list into two sublist of equal length. Because there is another parameter $k$ involved, time complexity should be the function in both $n$ and $k$.

Define $T(n, k)$ to be the worst case time for such quick-sort on a list that contains $k$ distinct elements. Then

$$T(n, k) = \max_{1 \leq r < k} \{T\left(\frac{n}{2}, r\right) + T\left(\frac{n}{2}, k - r - 1\right)\} + an$$

where $an$ is the time of partition.

To prove $T(n, k) = O(n \log_2 k)$, one needs to prove that $T(n, k) \leq cn \log_2 k$ for some constant $c$, when $n$ is sufficiently large.

Using the \textit{substitution method}.
Assume that $T(n^2, p) \leq c \frac{n}{2} \log_2 p$. Then we have

$$T(n, k) = \max_{1 \leq r < k} \{T(\frac{n}{2}, r) + T(\frac{n}{2}, k-r-1)\} + an$$

$$\leq \max_{1 \leq r < k} \{c \frac{n}{2} \log_2 r + c \frac{n}{2} \log_2 (k-r-1)\} + an$$

$$= \max_{1 \leq r < k} \{\frac{cn}{2} (\log_2 r + \log_2 (k-r-1))\} + an$$

$$= \max_{1 \leq r < k} \{\frac{cn}{2} \log_2 r(k-r)\} + an$$

(1)

term $r(k-r)$ achieves the maximum when $r = \frac{k}{2}$,

$$T(n, k) \leq \frac{cn}{2} \log_2 (\frac{k}{2})^2 + an$$

$$= cn(\log_2 k - 1) + an$$

$$= cn \log_2 k - (cn - an)$$

by choosing $c \geq a$,

$$T(n, k) \leq cn \log_2 k$$

Points broken down as
- 5 points for the recurrence
- 5 points for the proof of $O(n \log_2 k)$. 

6. A typical dynamic programming algorithm computes the optimal solution. For example, the Assembly Line Scheduling problem is to compute the fastest path through the factory. If not only the fastest path but also the second fastest path are to be found, the dynamic programming algorithm AssemblyLine can be modified to accommodate this new goal. Discuss how you may revise the algorithm AssemblyLine for the purpose of finding both the fastest and second fastest paths through the factory.

(1) Show the details of how you would modify the algorithm to satisfy the new requirements.

(2) (Graduates only, bonus for undergraduates) Argue that the idea can be applied to finding top \( k \) fastest paths through the factory, for any \( k \geq 2 \). What will be the time complexity of such an algorithm to find top \( k \) fastest paths (in terms of both \( n \) and \( k \))?

**Answers:**

(1) (10 points) The idea is to define a second function \( g \), namely \( g_1(j) \) and \( g_2(j) \) such that \( g_i(j) \) is the time of the second fastest path ending at station \( j \) of assembly line \( i \), \( i = 1, 2 \).

Then the function \( g \) can be recursively computed in almost the same fashion as function \( f \) (the fastest time). In particular,

If \( f_1(j) = f_1(j - 1) + a_{1,j} \) then

\[
g_1(j) = \min\{g_1(j - 1) + a_{1,j}, f_2(j - 1) + t_{2,j-1} + a_{1,j}\}
\]

If \( f_1(j) = f_2(j - 1) + t_{2,j-1} + a_{1,j} \) then

\[
g_1(j) = \min\{f_1(j - 1) + a_{1,j}, g_2(j - 1) + t_{2,j-1} + a_{1,j}\}
\]

Similarly, the options taken to compute \( g \) can all be recorded for the path traceback purpose.

- 4 points for the idea;
- 6 points for the detailed recurrence.

(2) (10 points) One can extend the idea of finding the top two fastest times to finding top \( k \) fastest times, for \( k \geq 2 \). Technically, one can define functions \( g^p_j \), where \( p = 1, 2, \ldots, k \), to represent the \( p \)th fastest time of a path ending at station \( j \) of assembly line \( i \), \( i = 1, 2 \). It is easy to see that \( g_1^1(j) = f_i(j) \).
For each $p = 2, \ldots, k$, the function $g^p_i(j)$ is computed after $g^{p-1}_i(j)$ is computed. The time complexity is thus $O(kn)$. 