CS101 Algorithms and Data Structures
Fall 2023
Homework 4

Due date: 23:59, November 5th, 2023

1. Please write your solutions in English.
2. Submit your solutions to gradescope.com.
3. Set your FULL name to your Chinese name and your STUDENT ID correctly in Account Settings.
4. If you want to submit a handwritten version, scan it clearly. `CamScanner` is recommended.
5. When submitting, match your solutions to the problems correctly.
6. No late submission will be accepted.
7. Violations to any of the above may result in zero points.

Notes:

1. Some problems in this homework requires you to design divide-and-conquer algorithm. When grading these problems, we will put more emphasis on how you reduce a problem to a smaller size problem and how to combine their solutions with divide-and-conquer strategy.
2. Your answer for these problems **should** include:
   (a) **Clear description** of your algorithm design in **natural language**, with **pseudocode** if necessary.
   (b) Run-time Complexity Analysis
   (c) Proof of Correctness (If required)
3. Your answer for these problems is not allowed to include real C or C++ code.
4. In your description of algorithm design, you should describe each step of your algorithm clearly.
5. You are encouraged to write pseudocode to facilitate explaining your algorithm desgin, though this is not mandatory. If you choose to write pseudocode, please give some additional descriptions to make your pseudocode intelligible.
6. You are recommended to finish the algorithm design part of this homework with LaTeX.

1. (0 points) Binary Search Example

Given a sorted array a of n elements, design an algorithm to search for the index of given element x in a.

Solution:

Algorithm Design: We basically ignore half of the elements just after one comparison.

1. Compare x with the middle element.
2. If x matches with the middle element, return the middle index.
3. Else If x is greater than the mid element, then x can only lie in right half subarray after the mid element. So we recur for right half.
4. Otherwise (x is smaller) recur for the left half.

Pseudocode(Optional):

$left$ and $right$ are indices of the leftmost and rightmost elements in given array a respectively.

```
1: function BINARYSEARCH(a, value, left, right)
2:     if right < left then
3:         return not found
4:     end if
5:     mid ← ⌊(right − left)/2⌋ + left
6:     if a[mid] = value then
7:         return mid
8:     end if
9:     if value < a[mid] then
10:        return binarySearch(a, value, left, mid-1)
11:    else
12:        return binarySearch(a, value, mid+1, right)
13:    end if
14: end function
```

Proof of Correctness: If x happens to be the middle element, we will find it in the first step. Otherwise, if x is greater than the middle element, then all the element in the left half subarray is less than x since the original array has already been sorted, so we just need to look for x in the right half subarray. Similarly, if x is less than the middle element, then all the element in the right subarray is greater than x, so we just need to look for x in the front list. If we still can't find x in a recursive call where $left = right$, which indicates that x is not in a, we will return $not\ found$ in the next recursive call.

Time Complexity Analysis: During each recursion, the calculation of mid and comparison can be done in constant time, which is $O(1)$. We ignore half of the elements after each comparison, thus we need $O(\log n)$ recursions.

$$T(n) = T(n/2) + O(1)$$

Therefore, by the Master Theorem $\log_b a = 0 = d$, so $T(n) = O(\log n)$.

2. (9 points) Multiple Choices

Each question has **one or more** correct answer(s). Select all the correct answer(s). For each question, you will get 0 points if you select one or more wrong answers, but you will get 1 point if you select a non-empty subset of the correct answers.

Write your answers in the following table.

(a)	(b)	(c)
AB	D	ACD

(a) (3') Which of the following sorting algorithms can be implemented as the stable ones?

A. Insertion-Sort

B. Merge-Sort

C. Quick-Sort (always picking the first element as pivot)

D. None of the above

(b) (3') Which of the following implementations of quick-sort take $\Theta(n \log n)$ time in the **worst case**?

A. Randomized quick-sort, i.e. choose an element from $\{a_l, \cdots, a_r\}$ randomly as the pivot when partitioning the subarray $\langle a_l, \cdots, a_r \rangle$.

B. When partitioning the subarray $\langle a_l, \cdots, a_r \rangle$ (assuming $r - l \geqslant 2$), choose the median of $\{a_x, a_y, a_z\}$ as the pivot, where x, y, z are three different indices chosen randomly from $\{l, l+1, \cdots, r\}$.

C. When partitioning the subarray $\langle a_l, \cdots, a_r \rangle$ (assuming $r - l \geqslant 2$), we first calculate $q = \frac{1}{2}(a_{max} + a_{min})$ where a_{max} and a_{min} are the maximum and minimum values in the current subarray respectively. Then we traverse the whole subarray to find a_m s.t. $|a_m - q| = \min_{i=l}^{r} |a_i - q|$ and choose a_m as the pivot.

D. None of the above.

(c) (3') Which of the following statements are true?

A. If $T(n) = 2T(\frac{n}{2}) + O(\sqrt{n})$ with $T(0) = 0$ and $T(1) = 1$, then $T(n) = \Theta(n)$.

B. If $T(n) = 4T(\frac{n}{2}) + O(n^2)$ with $T(0) = 0$ and $T(1) = 1$, then $T(n) = \Theta(n^2 \log n)$.

C. If $T(n) = 3T(\frac{n}{2}) + \Theta(n^2)$ with $T(0) = 0$ and $T(1) = 1$, then $T(n) = \Theta(n^2)$.

D. If the run-time $T(n)$ of a divide-and-conquer algorithm satisfies $T(n) = aT(\frac{n}{b}) + f(n)$ with $T(0) = 0$ and $T(1) = 1$, we may deduce that the run-time for merging solutions of a subproblems of size $\frac{n}{b}$ into the overall one is $f(n)$.

3. (15 points) Element(s) Selection

(a) **Selection of the k-th Minimal Value**

In this part, we will design an algorithm to find the k-th minimal value of a given array $\langle a_1, \cdots, a_n \rangle$ of length n with *distinct* elements for an integer $k \in [1, n]$. We say a_x is the k-th minimal value of a if there are exactly $k-1$ elements in a that are less than a_x, i.e.

$$|\{i \mid a_i < a_x\}| = k - 1.$$

Consider making use of the '**partition**' procedure in quick-sort. The function has the signature

```
int partition(int a[], int l, int r);
```

which processes the subarray $\langle a_l, \cdots, a_r \rangle$. It will choose a pivot from the subarray, place all the elements that are less than the pivot before it, and place all the elements that are greater than the pivot after it. After that, the index of the pivot is returned.

Our algorithm to find the k-th minimal value is implemented below.

```
// returns the k-th minimal value in the subarray a[l],...,a[r].
int kth_min(int a[], int l, int r, int k) {
    auto pos = partition(a, l, r), num = pos - l + 1;
    if (num == k)
    return a[pos];
    else if (num > k)
    return kth_min(a, l, pos - 1, k);
    else
    return kth_min(a, pos + 1, r, k - num);
  }
```

By calling `kth_min(a, 1, n, k)`, we will get the answer.

i. (2') Fill in the blanks in the code snippet above.

ii. (2') What's the time complexity of our algorithm in the **worst case**? Please answer in the form of $\Theta(\cdot)$ and fully justify your answer.

Solution:
In the worst case, we can make every pivot be the minimum and the number we tend to find be the maximum. Thus in every recrusion the program can only remove 1 number. To find the final answer, the function kth_min will be operated for n times. In the function partition, the worst case will be the minimum we chosen is at the end of array, thus we need to swap $\frac{n}{2}$ times. So that the complexity for k-length array is $\theta(k)$. Then the whole complexity is $\sum_{k=1}^{n} k$, which is $\theta(n^2)$.

(b) **Batched Selection**

Despite the worse-case time complexity of the algorithm in part(a), it actually finds the k-th minimal value of $\langle a_1, \cdots, a_n \rangle$ in expected $O(n)$ time. In this part, we will design a divide-and-conquer algorithm to answer m selection queries for distinct $k_1, k_2, \cdots, k_m$ where $k_1 < k_2 < \cdots < k_m$ on an given array a of n distinct integers (i.e. finding the k_1-th, k_2-th,$\cdots$,k_m-th minimal elements of a) and here m satisfies $m = \Theta(\log n)$.

i. (1') Given that x is the k_p-th minimal value of a and y is the k_q-th minimal value of a for $1 \leq p < q \leq m$, which of the following is true?
√ $x < y$ ○ $x = y$ ○ $x > y$

ii. (2') Suppose by calling the algorithm in part(a), we have already found z to be the k_l-th minimal value of a for $1 < l < m$. Let $L = \{a_i \mid a_i < a_z\}$ and $R = \{a_i \mid a_i > a_z\}$. What can you claim about the k_1-th,$\cdots$,k_{l-1}-th minimal elements of a and the k_{l+1}-th,$\cdots$,k_m-th minimal elements of a?

Solution:
we know that for the k_1th to k_{l-1}th minimal elements, they are less than k_lth minimum, so that they belong to L. On the other hand, minimal elements that k_{l+1}th to k_mth are larger than k_lth minimum, thus belong to R.

iii. (6') Based on your answers of previous parts, design a divide-and-conquer algorithm, **which calls the algorithm in part(a) as a subroutine**, for this problem. Your algorithm should runs in **expected** $O(n \log m) = O(n \log \log n)$ time. Any algorithms that run in $\Omega(n \log n)$ time will get no credit. Make sure to provide **clear description** of your algorithm design in **natural language**, with **pseudocode** if necessary.

Solution:
Algorithm Design:

1. Sort m minimums into descending order, using quick sort or merge sort. Assume we use array A to store these minimums. The whole array called R.
2. Choose the median element of A and find its location in R, using some part of the code in part(a). Return the position of the madian element. Divide the R and A into two parts, with one all number is larger than median number and the other is less than median number.
3. For the divided R and A, called A' and R', choose the median number of A'. Find its location in R'. Return its location and divide A' and R' in the same way.
4. Repeat the operations above until all elements are found.

Pseudocode:
A is an array to restore the numbers to find. `left` and `right` are indices of the leftmost and rightmost elements in array A. `k` means the length of A. R is an array for all n numbers. `head` is the head address of whole n numbers. `tail` is the tail address of whole n numbers. In this pseudocode, some functions like

Quick$_s$ort, which is not changed, or partition, which is a part of functions that are not changed, will be used without concrete realization.

```
1: function quick_sort(A, left, right)
2: end function
3: function PARTITION(R, head, tail)
4: end function
5: function kth_min(R, head, tail, k)
6:     pos = partition(R, head, tail)
7:
8:     num = pos − head + 1
9:
10:    if num == k then
11:
12:        if k != 1 then
13:
14:            return kth_min(R, head, pos − 1, k/2)
15:
16:            return kth_min(R, pos + 1, tail, k/2)
17:
18:        else
19:            return pos
20:
21:        end if
22:    else if num > k then
23:        return kth_min(R, head, pos − 1, k)
24:    else
25:        return kth_min(R, pos + 1, tail, k − num)
26:    end if
27: end function
```

iv. (2') Provide your reasoning for why your algorithm in the previous part runs in expected $O(n \log m)$ time using the **recursion-tree** method.

Solution:
For R with length of n and A with length of m, we have the complexity

$$\begin{aligned} T(find) &= \theta(n) \\ T(partition) &= \theta(n) \end{aligned} \tag{1}$$

Thus for R of length n, the time complexity is $\theta(n)$.
Then persue the height of the recrusion tree. We have to find m numbers, and everytime we divide the array R into 2 subarraies, so that we have $height = log(m)$.
The recrusion will be

$$T(n) = 2T(\frac{n}{2}) + \theta(n) \tag{2}$$

Since the recrusion end in log(m) times, the whole time complexity is $\theta(nlog(m))$.

4. (13 points) Maximum area rectangle in histogram

We are given a histogram consisting of n parallel bars side by side, each of width 1, as well as a sequence A containing the heights of the bars where the height of the ith bar is a_i for $\forall\, i \in [n]$. For example, the figures below show the case where $n = 7$ and $A = \langle 6, 2, 5, 4, 4, 1, 3\rangle$. Our goal is to find the maximum area of the rectangle placed inside the boundary of the given histogram with a **divide-and-conquer** algorithm. (Here you don't need to find which rectangle maximizes its area.)

Reminder: There do exist algorithms that solve this problem in linear time. However, you are **not allowed** to use them in this homework. Any other type of algorithms except the divide-and-conquer ones will get **no** credit.

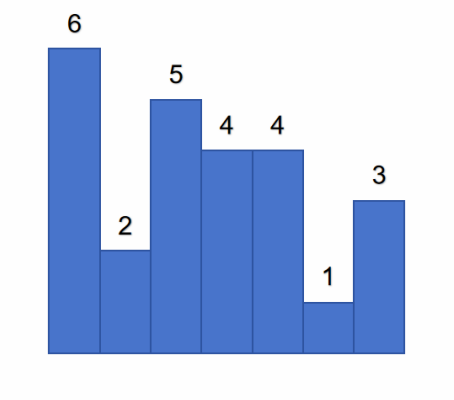

Figure 1: The Original Histogram

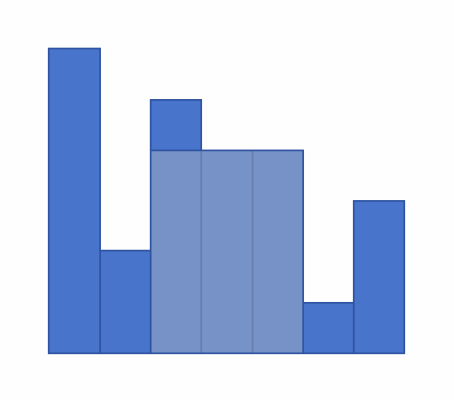

Figure 2: The Largest Rectangle in Histogram

You may use $\mathsf{Rect}(l, r, A)$ to represent the answer of the sub-problem w.r.t. the range $[l, r]$.

(a) (3') **Briefly** describe:

1. How would you divide the original problem into 2 sub-problems?
2. Under what circumstances will the answer to the original problem not be covered by the answers of the 2 sub-problems?
3. Given the answers of the 2 sub-problems, how would you get the answer of the original problem?

Solution:
For problem one and three,

1. Choose the first minimum and divide the array into two parts. Part one cotains the elements on the left of first minimum, and the other elements without the first element form the other part.
2. Update the answer, which is 0 at the begining, with the larger one between the length of subarray multiply minimum and the answer.
3. Repeat the operations above for every subarray.

For problem two,
If the array is strictly ascending or descending, the problem will not be divided into two subproblems.

(b) (8') Based on your idea in part(a), design a **divide-and-conquer** algorithm for this problem. Make sure to provide **clear description** of your algorithm design in **natural language**, with **pseudocode** if necessary.

Solution:

1. Find the minimum called m_1, with position pos.
2. Update the answer with $m_1 * n$.
3. Divide the array A into two parts. One is A from head to $pos - 1$, called A_1 and the other is A from $pos + 1$ to tail, called A_2.
4. Find the minimum in A_1 called m_2 and its position.
5. Update the answer with $max(answer, m_2 * (pos - head))$.
6. Divide the array A_1 into two parts in the same way as A.
7. Repeat the operations above through recrusion, and end progressing if the length of A_i is 1.

Pseudocode:
A is the original array with length n. `head` and `tail` is the head address or tail address of array A.

```
1: answer = 0
2: function find_min(A, head, tail)
3:     min = A[0]
4:     pos = head
5:
6:     for i=1,i < n,i++ do
7:
8:
9:         if min > A[i] then
10:
11:
12:            min = A[i]
13:
14:
15:            pos = i
16:
17:
18:        end if
19:
20:    end for
21:    return pos
22: end function
23: function maximum_area(A, head, tail)
24:    pos = find_min(A, head, tail)
25:    answer = max(A[pos]*(tail-head+1), answer)
26:    if head == tail then
27:        return answer
28:    end if
29:    if pos == head then
30:        return max(answer, maximum_area(A, pos+1, tail))
31:    else if pos == tail then
32:        return max(answer, maximum_area(A, head, pos-1))
33:    else
34:        return max(maximum_area(A, head, pos-1), maximum_area(A, pos+1,
   tail))
35:    end if
36: end function
```

(c) (2') Provide the run-time complexity analysis of your algorithm in part (b). Make sure to include the **recurrence relation** of the run-time in your solution.

Solution: For the minimum in array, the expected position is $\sum_{i=0}^{n-1} \frac{i}{n}$, which is the middle. Thus the problem will be divided into two equal parts in expectation. In the function $\mathtt{maximum_area}$ without recrusion, the time complexity is $\theta(n)$ since we should find the minimum. Thus, we have

$$T(n) = 2T(\frac{n}{2}) + \theta(n) \tag{3}$$

According to the master therom, the whole time complexity is $\theta(n\log n)$.

5. (17 points) Dividing with Creativity

In this question, you are required analyze the run-time of algorithms with different dividing methods mentioned below. For each subpart except the third one, your answer should include:

1. Describing the recurrence relation of the run-time $T(n)$. (Worth 1 point in 4)
2. Finding the asymptotic order of the growth of $T(n)$ i.e. find a function g such that $T(n) = O(g(n))$. Make sure your upper bound for $T(n)$ is tight enough. (Worth 1 point in 4)
3. Show your **reasoning** for the upper bound of $T(n)$ or your process of obtaining the upper bound starting from the recurrence relation step by step. (Worth 2 points in 4)

In each subpart, you may ignore any issue arising from whether a number is an integer as well as assuming $T(0) = 0$ and $T(1) = 1$. You can make use of the Master Theorem, Recursion Tree or other reasonable approaches to solve the following recurrence relations.

(a) (4') An algorithm $\mathcal{A}_1$ takes $\Theta(n)$ time to partition the original problem into 2 sub-problems, one of size λn and the other of size $(1-\lambda)n$ (here $\lambda \in (0, \frac{1}{2})$), then recursively runs itself on both of the 2 sub-problems and finally takes $\Theta(n)$ time to merge the answers of the 2 sub-problems.

Solution:

1. $T(n) = T(\lambda n) + T((1-\lambda)n) + \theta(n)$
2. g(n) = nlogn
3. Knowing that

$$\begin{aligned} T(n) &= T(\lambda n) + T((1-\lambda)n) + \theta(n) \\ T(\lambda n) &= T(\lambda^2 n) + T(\lambda(1-\lambda)n) + \theta(\lambda n) \\ T((1-\lambda)n) &= T((1-\lambda)\lambda n) + T((1-\lambda)^2 n) + \theta((1-\lambda)n) \\ T(n) &= T(\lambda^2 n) + 2T(\lambda(1-\lambda)n) + T((1-\lambda)^2 n) + \theta(2n) \end{aligned} \tag{4}$$

Thus assuming that the recrusion goes k times, the whole time complexity is $\theta(kn)$.
Repeat the operations above until $(1-\lambda)^k n \approx 1$, at which time other items ≈ 0.
So that $k = \theta(\log n)$, $g(n) = \theta(n\log n)$.

(b) (4') An algorithm $\mathcal{A}_2$ takes $\Theta(n)$ time to partition the original problem into 2 sub-problems, one of size k and the other of size $(n-k)$ (here $k \in \mathbb{Z}^+$ is a constant), then recursively runs itself on both of the 2 sub-problems and finally takes $\Theta(n)$ time to merge the answers of the 2 sub-problems.

Solution:

1. $T(n) = T(k) + T(n-k) + \theta(n)$
2. $g(n) = n^2$
3. $$\begin{aligned} T(n) &= T(k) + T(n-k) + \theta(n) \\ T(k) &= T(0) + T(k) + \theta(1) = T(k) + \theta(1) \\ T(n-k) &= T(k) + T(n-2k) + \theta(n) \\ T(n) &= 2T(k) + T(n-2k) + \theta(2n) \end{aligned} \tag{5}$$

Thus we have $T(n) = \frac{n}{k}T(k) + \theta(\frac{n^2}{k})$. The time complexity is $\theta(n^2)$.

(c) Solve the recurrence relation $T(n) = T(\alpha n) + T(\beta n) + \Theta(n)$ where $\alpha + \beta < 1$ and $\alpha \geq \beta$.

i. (2') Fill in the **four** blanks in the mathematical derivation snippet below.

$$\begin{aligned} T(n) &= T(\alpha n) + T(\beta n) + \Theta(n) \\ &= (T(\alpha^2 n) + T(\alpha\beta n) + \Theta(\alpha n)) + (T(\alpha\beta n) + T(\beta^2 n) + \Theta(\beta n)) + \Theta(n) \\ &= (T(\alpha^2 n) + 2T(\alpha\beta n) + T(\beta^2 n)) + \Theta(n)(1 + (\alpha + \beta)) \\ &= \dots \\ &= \sum_{i=0}^{k} \binom{k}{i} T(\alpha^i \beta^{k-i} n) + \Theta(n) \sum_{j=0}^{k} (\alpha + \beta)^j \end{aligned}$$

ii. (3') Based on the previous part, complete this question.

Solution:
Knowing $\alpha \geq \beta$, make $k = \log_\alpha n$. Thus the equation will be

$$T(n) = \theta(n) \sum_{j=0}^{\log(n)} (\alpha + \beta)^j = \theta(n) \tag{6}$$

(d) (4') An algorithm $\mathcal{A}_3$ takes $\Theta(\log n)$ time to convert the original problem into 2 sub-problems, each one of size $\sqrt{n}$, then recursively runs itself on both of the 2 sub-problems and finally takes $\Theta(\log n)$ time to merge the answers of the 2 sub-problems.

Hint: W.L.O.G., you may assume $n = 2^m$ for $m \in \mathbb{Z}$.

Solution:

From the problem we have

$$T(n) = 2T(\sqrt{n}) + \theta(\log(n)) \tag{7}$$

Let $n = 2^m$, we have $a = 2$, $b = 2^{\frac{m}{2}}$, $f(n) = m$. Since

$$n^{\log_b^a} = (2^m)^{\frac{2}{m}} = 4 < n^d = m \tag{8}$$

$T(n) = \theta(\log(n))$.