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.

```
function BINARYSEARCH(a, value, left, right)
    if right < left then
        return not found
    end if
    mid ← ⌊(right − left)/2⌋ + left
    if a[mid] = value then
        return mid
    end if
    if value < a[mid] then
        return binarySearch(a, value, left, mid-1)
    else
        return binarySearch(a, value, mid+1, right)
    end if
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 $\mathsf{left} = \mathsf{right}$, which indicates that x is not in a, we will return $\mathsf{not\ found}$ 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:
Time Complexity Analysis in the worst case: During each recursion, the operations can be done in $\Theta(n)$. In the worst case, the pivot selected is always the smallest or largest but incorrect element, so we need $\Theta(n)$ recursions. Therefore, $T(n) = \Theta(n^2)$ in the worst case.

(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 can claim that

1. The k_1-th,$\ldots$,k_{l-1}-th minimal elements of a are all in L.
2. The k_{l+1}-th, $\ldots$, k_m-th minimal elements of a are all in 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.

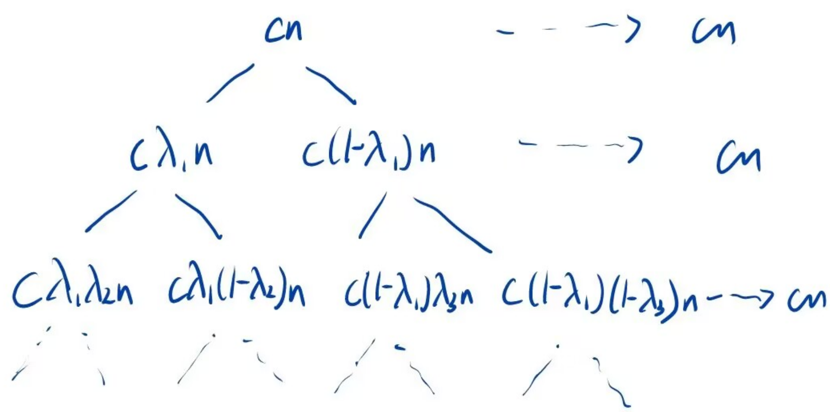
Solution:
Algorithm Design:

1. If $m = 1$, find the k_1-th minimal element in a using the algorithm from **part(a)**.
2. Otherwise, let $z = k_{\lceil \frac{m}{2} \rceil}$, find z-th minimal element in a using the algorithm from **part(a)** and partition the rest of a into L and R as in **ii.**.
3. Recursively find the k_1-th,$\ldots$,$k_{\lceil \frac{m}{2} \rceil - 1}$-th minimal elements in L, and the $(k_{\lceil \frac{m}{2} \rceil + 1} - k_{\lceil \frac{m}{2} \rceil})$-th, $\ldots$, $(k_m - k_{\lceil \frac{m}{2} \rceil})$-th minimal elements in R.

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:

Time Complexity Analysis: During each recursion, the operations can be done in $O(n)$. The depth of the recursion-tree is $\log_2 m$.



So, the algorithm in the previous part runs in expected $O(n \log m)$.

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 $\mathbf{a}_i$ 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.

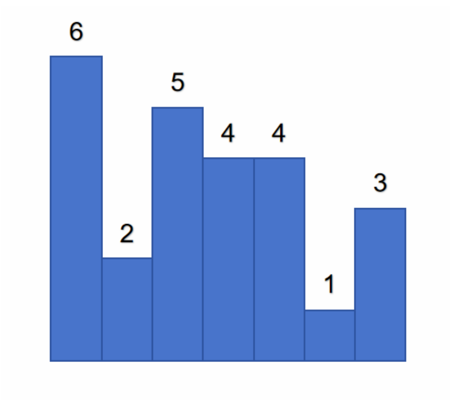


Figure 1: The Original Histogram

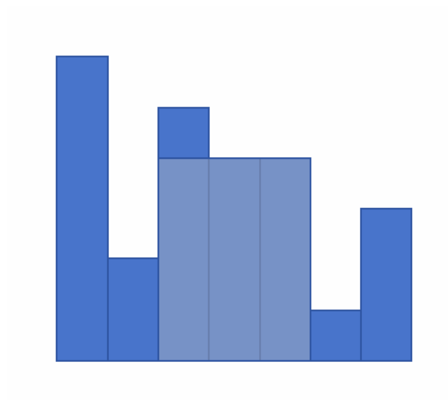


Figure 2: The Largest Rectangle in Histogram

You may use $\mathtt{Rect}(l, r, A)$ 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:

1. First we should find $\mathtt{pivot}$, the bar of the minimal height, in the original problem and then divide the original problem into the one of the bars on the left side of $\mathtt{pivot}$ and the one of the bars on the right side.
2. If the answer to the original problem is the area of the rectangle including $\mathtt{pivot}$, it will not be covered by the answers of the 2 sub-problems.
3. Compare them with the maximum area including $\mathtt{pivot}$, which is the height of $\mathtt{pivot}$ multiplied by the size of the original problem, and choose the biggest one as the answer to the original problem.

(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:
Algorithm Design:

1. If $n = 0$, return 0. If $n = 1$, return the height of the single bar.
2. Otherwise, find pivot, the bar of the minimal height, and calculate $\mathsf{mid\_max}$, the height of pivot multiplied by the size of the problem.
3. Divide the original problem into the one of the bars on the left side of pivot and the one of the bars on the right side of pivot.
4. Recursively solve the 2 sub-problem and compare $\mathsf{left\_max}$ and $\mathsf{right\_max}$, the answers, with $\mathsf{mid\_max}$. Return the maximum of the three at last.

Pseudocode:
l and r are indices of the leftmost and rightmost elements in given array A respectively.

```
function Rect(l,r,A)
    if l > r then
        return 0
    end if
    if l = r then
        return A[l]
    end if
    //FindMin(A) return the index of the minimal element in the array
    pivot ← FindMin(A)
    max_mid ← A[pivot] * (r − l + 1)
    max_left ← Rect(l, pivot − 1, A)
    max_right ← Rect(pivot + 1, r, A)
    return max(max_mid, max_left, max_right)
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:

Time Complexity Analysis: During each recursion, the operations can be done in $O(n)$ since the time complexity of $\mathsf{FindMin}(A)$ is $O(n)$. We need $O(\log n)$ recursions totally on average. Therefore, by the Master Theorem $\log_b a = 1 = d$, so $T(n) = O(n \log n)$ in average case.

In the worst case, the bar of the minimal height is always the rightmost element. In this case, the time complexity of $\mathsf{FindMin}(A)$ is $\Theta(n)$ and we need $\Theta(n)$ recursions totally.

$$T(n) = T(n-1) + \Theta(n)$$

So, $T(n) = \Theta(n^2)$.

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:

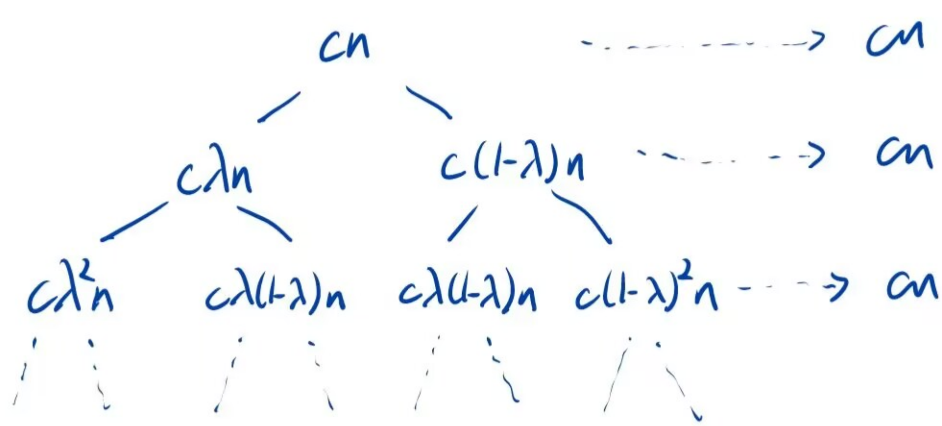
1. Describing the recurrence relation of the run-time $T(n)$. (Worth 1 point in 4)
2. Finding the asymptotic order of the growth of $T(n)$ i.e. find a function g such that $T(n) = O(g(n))$. Make sure your upper bound for $T(n)$ is tight enough. (Worth 1 point in 4)
3. Show your **reasoning** for the 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 $\lambda \in \left(0, \frac{1}{2}\right)$), 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. $T(n) = T(\lambda n) + T((1-\lambda)n) + \Theta(n) + \Theta(n) = T(\lambda n) + T((1-\lambda)n) + \Theta(n)$
2. $g(n) = n \log n$
3. During each recursion, the operations can be done in $\Theta(n)$. The depth of the recursion-tree is $\log_{\frac{1}{1-\lambda}} n$.



So, $T(n) = \Theta(n) \cdot \log_{\frac{1}{1-\lambda}} n = O(n \log n)$.

(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. $T(n) = T(k) + T(n-k) + \Theta(n) + \Theta(n) = T(k) + T(n-k) + \Theta(n)$
2. $g(n) = n^2$
3. $$\begin{aligned} T(n) = &\ T(k) + T(n-k) + \Theta(n) \\ = &\ \cdots \\ = &\ \lfloor \frac{n}{k} \rfloor \cdot T(k) + T(n - k \cdot \lfloor \frac{n}{k} \rfloor) + \lfloor \frac{n}{k} \rfloor \cdot \Theta(n) \\ = &\ O(n^2) \end{aligned}$$

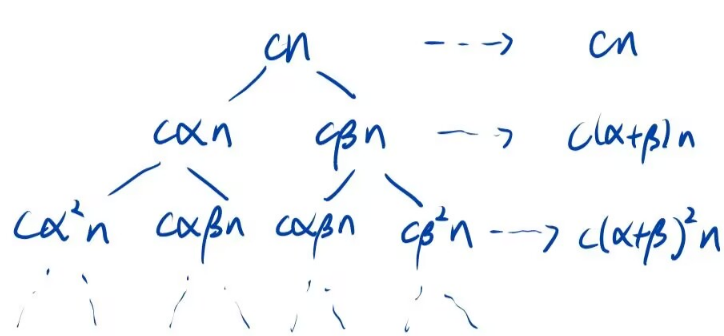
(c) Solve the recurrence relation $T(n) = T(\alpha n) + T(\beta n) + \Theta(n)$ where $\alpha + \beta < 1$ and $\alpha \geq \beta$.

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

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

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

Solution:
Since $\alpha + \beta < 1$ and $\alpha \geq \beta$, we have that the depth of the recursion $k = \log_{\frac{1}{\alpha}} n$, and the operations during each recursion can be done in $(\alpha+\beta)^k O(n)$.



So, $T(n) = \sum_{k=0}^{\log_{\frac{1}{\alpha}} n} (\alpha+\beta)^k O(n) = O(n)$.

(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:

1. $T(n) = 2T(\sqrt{n}) + \Theta(\log n) + \Theta(\log n) = 2T(\sqrt{n}) + \Theta(\log n)$
2. $g(n) = \log n \log \log n$
3. Since $T(n) = 2T(\sqrt{n}) + \Theta(\log n)$, we have $T(2^m) = 2T(2^{\frac{m}{2}}) + \Theta(m)$.
   Denote $S(m) = T(2^m)$, then $S(m) = 2S(\frac{m}{2}) + \Theta(m)$.
   By Master's Theorem, $\log_b a = 1 = d$, so $S(m) = O(m \log m)$.
   So, $T(n) = O(\log n \log \log n)$.