divide-and-conquer **May 11, 2025**

Homework 3

Spring 2025 *COSC31: Algorithms*

1. **implementation**
   We studied in class the $Select(A[1...n], k)$ method, which returns the kth smallest element of an unsorted array $A[1...n]$ with a time complexity of $O(n)$. As you know, when designing any algorithm, we typically express the algorithm at a high level (in pseudocode) that conveys all salient points, but suppresses implementation details. It is important for you to develop the skill and confidence that you can take things forward from here, i.e., that you can turn the pseudocode description into a full implementation in a programming language of your choice and run the code. So, I ask you in this problem to implement the select algorithm in a language of your choice, test it, and turn in your code and two sample runs.

   Answer

   colab notebook: `https://colab.research.google.com/drive/125n1qAVGQll9jhfgzQe80_gZAQ9MFJcR?usp=sharing`

```python
def FindPivot(arr):
    '''Find the median of medians of sub-arrays of size at most 5'''
    sub_arr_medians = []
    n = len(arr)
    ptr = 0

    while ptr  < n:
        if (n - ptr) < 5:
            sorted_sub_arr = sorted(arr[ptr: len(arr)])
        else:
            sorted_sub_arr = sorted(arr[ptr: ptr+5])
        sub_arr_medians.append(sorted_sub_arr[len(sorted_sub_arr)//2])
        ptr += 5
    if len(sub_arr_medians) == 1:
        return sub_arr_medians[0]
    return Select(sub_arr_medians, k = (len(sub_arr_medians) + 1) // 2)

def Partition(arr, _pivot):
    '''Quick sort, find left and right array of the pivot, and rank of pivot'''
    left_arr = [x for x in arr if x < _pivot]
    right_arr = [x for x in arr if x > _pivot]
    pivot_index = len(left_arr) + 1

    return left_arr, right_arr, pivot_index

def Select(arr, k):

    if k < 1 or k > len(arr): return None
    if len(arr) == 1: return arr[0]

    # Find the pivot
    _pivot = FindPivot(arr)
    left, right, _rank = Partition(arr, _pivot)

    # recusive cases
    if k == _rank: return _pivot
```

```
    elif k < _rank: return Select(left, k)
    else: return Select(right, k - _rank)

# Test cases
## edge cases
assert(Select([3, 2, 1, 4, 0, 5, 6, 7], 0) == None)
assert(Select([3, 2, 1, 4, 0, 5, 6, 7], 10) == None)
assert(Select([], 10) == None)

## normal cases
assert(Select([109], 1) == 109)
assert(Select([3, 2, 1, 4, 0, 5, 6, 7], 5) == 4)
assert(Select([3, 2, 1, 4, 0, 5, 6, 7], 1) == 0)
assert(Select([-3, 2, 1, -4, 0, 5, 6, 7], 2) == -3)
assert(Select([x for x in range(1001)], 99) == 98)
```

2. Design by divide-and-conquer a $O(\lg m + \lg n)$-time algorithm that, given a positive integer k and two sorted arrays a and b of size m and n, returns the kth smallest element of the union of the two arrays. Answer the following:

   (a) Design a divide-and-conquer algorithm by completing the pseudocode for the method below.

Answer

Algorithm SelectFromTwoSortedArrays

Function `SelectFromTwoSortedArrays`(a, b, i, j, p, q, k)**:**

```
if k ≤ 0 then
    return ⊥
end
if len(A) = 0 then
    return b[k]
end
if len(B) = 0 then
    return a[k]
end
a_m ← ⌊(i+j)/2⌋
b_m ← ⌊(p+q)/2⌋

a_left ← a_m − i + 1
b_left ← b_m − p + 1

if a_left + b_left < k then
    if a[a_m] < b[b_m] then
        return SelectFromTwoSortedArrays(a, b, a_m + 1, j, p, q, k − a_left)
    end
    else
        return SelectFromTwoSortedArrays(a, b, i, j, b_m + 1, q, k − b_left)
    end
end
else if  a_left + b_left > k then
    if a[a_m] < b[b_m] then
        return SelectFromTwoSortedArrays(a, b, i, j, p, b_m − 1, k)
    end
    else
        return SelectFromTwoSortedArrays(a, b, i, a_m − 1, p, q, k)
    end
end
else
    return min(a[a_m], b[b_m])
end
```

(b) Argue the correctness of your algorithm. It is sufficient to state and justify the key observation that your algorithm is based on

Answer

The key observation is that the kth smallest element in the union of two sorted arrays can be found by comparing the middle elements of the current subarrays. At each step, we determine how many elements lie in the left halves of both arrays (up to their respective midpoints).

If the total number of elements in these left halves is less than k, then the kth smallest element must lie outside this region — specifically in the right half of one of the arrays. In this case, we discard the left half of the array with the smaller middle element and adjust

k accordingly.
On the other hand, if the total number of elements in the left halves is at least k, then the kth smallest element must lie within this region. We can safely discard the right half of the array with the larger middle element, without changing k.
This divide-and-conquer process eliminates about half of the elements from consideration in each step, leading to a logarithmic number of recursive calls in the sizes of the arrays. The algorithm terminates once one of the arrays is empty or when the base case is met, returning the correct kth smallest element.

(c) Let $T(m, n)$ be the time complexity of your algorithm. State the recurrence for $T(m, n)$, where m and n are as defined in the pre-condition.

Answer

The algorithm either chops off half of the elements from one of the arrays so it is either $T(\frac{m}{2}, n)$ or $T(m, \frac{n}{2})$. The combine time is dominated by the comparison of the two middle elements, which is $O(1)$. Hence, the recurrence relation is:

$$T(m,n) = \max\left(T\left(\frac{m}{2}, n\right), T\left(m, \frac{n}{2}\right)\right) + O(1)$$

(d) Prove by substitution that $T(m, n) = O(\lg m + \lg n)$.

Answer

As shown above, we have the following recurrence relation:

$$T(m,n) = \max\left(T\left(\frac{m}{2}, n\right), T\left(m, \frac{n}{2}\right)\right) + O(1)$$

We claims that the $T(m, n) = O(\log m + \log n)$, which can be written as: $T(m, n) \leq c \cdot (\log m + \log n)$ for some constant $c > 0, n_0 > 0, m_0 > 0$, and $\forall n > n_0$ and $\forall m > m_0$.
Inductive hypothesis: Assume that $T(a, b) \leq c \cdot (\log a + \log b)$ holds $\forall a < m$, and $\forall b < n$.
Inductive case: $T(m, n) \leq c(\log m + \log n)$.
Suppose we take $T(m/2, n)$ yields the max value, we know that

$$\begin{aligned} T\left(\frac{m}{2}, n\right) &\leq T\left(\frac{m}{2}, n\right) + d \\ &= c \cdot (\log \frac{m}{2} + \log n) + d \\ &= c \cdot (\log m - 1 + \log n) + d \\ &= c \cdot (\log m + \log n) - c + d \end{aligned}$$

If we pick $c > d$, we see that $T(m, n) \leq c \cdot (\log m + \log n)$. Therefore, for $c > d, n_0 > 0, m_0 > 0$ and $\forall n > n_0$ and $\forall m > m_0$, we have: $T(m, n) = O(\log m + \log n)$
Otherwise, we can show that:

$$\begin{aligned} T\left(m, \frac{n}{2}\right) &\leq T\left(m, \frac{n}{2}\right) + d \\ &= c \cdot (\log m + \log \frac{n}{2}) + d \\ &= c \cdot (\log m + \log n - 1) + d \\ &= c \cdot (\log m + \log n) - c + d \end{aligned}$$

If we pick $c > d$, we see that $T(m,n) \leq c \cdot (\log m + \log n)$. Therefore, for $c > d, n_0 > 0, m_0 > 0$ and $\forall n > n_0$ and $\forall m > m_0$, we have: $T(m,n) = O(\log m + \log n)$

3. **Local Minimum 2-D**

   (a) Clearly describe your algorithm.

Algorithm Local Minimum in a 2-D Matrix

Function `FindLocalMinimaTwoDMatrix`(G, n):

$mid \leftarrow \lfloor \frac{n+1}{2} \rfloor$

Identify minVal and its coordinates (minRow, minCol) among the middle row, middle column, and boundary edges.

if *minVal is smaller than all valid neighbors* **then**
  return minVal
end

Identify and cache valid neighbors of minVal to be considered:
If minVal in middle column: check left and right neighbors (if within bounds)
If minVal in middle row: check top and bottom neighbors (if within bounds)
If on boundary:
If in column 1: check right neighbor
If in column n: check left neighbor
If in row 1: check below
If in row n: check above

if *minVal lies on the middle column, and is not a local minimum* **then**
  if *left neighbor* $<$ *right neighbor* **then**
    if *left neighbor is above middle row* **then**
      return `FindLocalMinimaTwoDMatrix` $(G[1 \dots mid-1, 1 \dots mid-1], mid-1)$
    end
    else if *left neighbor is below middle row* **then**
      return `FindLocalMinimaTwoDMatrix` $(G[mid+1 \dots n, 1 \dots mid-1], mid-1)$
    end
  end
  else
    if *right neighbor is above middle row* **then**
      return `FindLocalMinimaTwoDMatrix` $(G[1 \dots mid-1, mid+1 \dots n],\ mid-1)$
    end
    else if *right neighbor is below middle row* **then**
      return `FindLocalMinimaTwoDMatrix` $(G[mid+1 \dots n, mid+1 \dots n],\ mid-1)$
    end
  end
end
else if *minVal lies on the middle row, and is not a local minimum* **then**
  if *upper neighbor* $<$ *lower neighbor* **then**
    if *upper neighbor is left of middle column* **then**
      return `FindLocalMinimaTwoDMatrix` $(G[1 \dots mid-1,\ 1 \dots mid-1],\ mid-1)$
    end
    else if *upper neighbor is right of middle column* **then**
      return `FindLocalMinimaTwoDMatrix` $(G[1 \dots mid-1,\ mid+1 \dots n],\ mid-1)$
    end
  end
  else
    if *lower neighbor is left of middle column* **then**
      return `FindLocalMinimaTwoDMatrix` $(G[mid+1 \dots n,\ 1 \dots mid-1],\ mid-1)$
    end
    else if *lower neighbor is right of middle column* **then**
      return `FindLocalMinimaTwoDMatrix` $(G[mid+1 \dots n,\ mid+1 \dots n],\ mid-1)$
    end
  end
end
else if *minVal lies on the boundary, and is not a local minimum* **then**
  Identify neighbor with minimum value.
  Recurse into the quadrant that contains that neighbor,
  and compute the correct submatrix size.
end

(b) Rigorously prove the correctness of your algorithm.

Answer

Let the predicate $P(n)$ be true if for any 2-D matrix $G[1\ldots n,\ 1\ldots n]$ of distinct integers, where $n \geq 1$, and such that $G[i,j] = \infty$ for all $i \in \{0, n+1\}$ or $j \in \{0, n+1\}$, there exists an index pair (i,j) with $1 \leq i,j \leq n$ such that $G[i,j] < G[i-1,j]$, $G[i,j] < G[i+1,j]$, $G[i,j] < G[i,j-1]$, $G[i,j] < G[i,j+1]$. That is, $G[i,j]$ is strictly less than all four of its neighbors, including those that lie on the boundary of the matrix, which are defined to have value ∞. We will prove that $P(n)$ holds for all $n \in \mathbb{N}$ by strong induction.

Base case: For $n = 1$, $G[1,1]$ is bounded by ∞ on all sides, so it is trivially a local minimum. This establishes $P(1)$.

Inductive step: Fix $k \geq 1$, and assume $P(a)$ holds for all $1 \leq a \leq k$. We want to prove that $P(k+1)$ holds.
Let $G'[1\ldots k+1,\ 1\ldots k+1]$ be any matrix of distinct integers, with boundary values $G'[i,j] = \infty$ for all $i \in \{0, k+2\}$ or $j \in \{0, k+2\}$.
The algorithm examines the middle row, middle column, and boundary edges to identify the global minimum among these. Let minVal denote this minimum value. We consider the following cases:

(a) If minVal is smaller than all its four neighbors, then it is a local minimum and is returned.

(b) If minVal lies on the middle column and is not a local minimum, we compare its left and right neighbors. We recurse into the left half if the left neighbor is smaller, and into the right half otherwise.

(c) If minVal lies on the middle row and is not a local minimum, we compare its top and bottom neighbors. We recurse into the upper half if the top neighbor is smaller, and into the lower half otherwise.

(d) If minVal lies on the boundary and is not a local minimum, we compare its in-bounds neighbors and recurse into the quadrant that contains the neighbor with the smallest value.

In all cases, the submatrix we recurse into has size at most $\lfloor \frac{k+1}{2} \rfloor by \lfloor \frac{k+1}{2} \rfloor$ where $\lfloor \frac{k+1}{2} \rfloor \rfloor \leq k$. By the inductive hypothesis, the recursive call returns a local minimum within that submatrix. This local minimum is also valid in the original matrix because we recurse toward the smallest neighbor of minVal, and any neighbor outside the submatrix is either ∞ or was already compared when identifying the global minimum and found to be larger than the minVal, and consequently any value smaller than the minVal. Thus, the returned element is smaller than all four of its neighbors in G'. Therefore, $P(k+1)$ holds. By strong induction, $P(n)$ holds for all $n \in \mathbb{N}$.

(c) State the algorithmś time complexity as a recurrence, and prove that the algorithmś time complexity is $O(n)$.

Answer

The algorithm has a time complexity of $O(n)$. The combine time, $f(n)$, considerations are $O(1)$ for the comparison of the potenial local minimum with its neighbors, and $O(n)$ for

finding the minimum in the middle column for a given recursive call, hence $f(n) = O(n)$. For each recursive call, we are decrementing the number of columns and rows of the matrix by half each, hence the recurrence relation, $T(n)$, is as follows:

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

By masters theorem, we have:

$$f(n) = O(n)$$
$$n^{\log_b a} = n^{\log_2 2^0} = 1$$

We see that this fits into case 3, for $\epsilon = 1$:

$$\begin{aligned} f(n) &= \Omega(n^{\log_b a + 1}) \\ &= \Omega(n^{\log_2 2^0 + 1}) \\ &= \Omega(n) \end{aligned}$$

Checking the regularity condition:

$$T(n) \leq T(\frac{n}{2}) + cn$$
$$af\left(\frac{n}{b}\right) \leq cf(n)$$
$$1 \cdot \frac{n}{2} \leq cn$$

For some c where $\frac{1}{2} < c < 1$, the regularity condition is satisfied. Hence $T(n)$ is:

$$\begin{aligned} T(n) &= \Theta(f(n)) \\ &= \Theta(n) \end{aligned}$$

Since $T(n)$ is $\Theta(n)$, we can conclude that the algorithm runs in $O(n)$ time.