COL351: Analysis and Design of Algorithms **Fall 2024**

Tutorial Sheet 2 Solutions

Announced on: Aug 01 (Thurs)

Problems marked with ($\star$) will not be asked in the tutorial quiz.

1. Consider the following algorithm for checking whether a given number n is prime.

ALGORITHM 1: Checking primality

Input: An integer $n > 1$.
Output: Yes/No

if n *equals* 2 *or* 3 **then**
  return Yes
for $i = 2$ *to* $\lfloor \sqrt{n} \rfloor$ **do**
  if i *divides* n **then**
    return No
return Yes

Prove or disprove:

a) The algorithm is correct.

b) The algorithm runs in polynomial time.

a) If the algorithm outputs "No", it means that there exists an integer $2 \leqslant i \leqslant \lfloor \sqrt{n} \rfloor$ such that i divides n. So, n is not a prime.

Now, we'll prove that if the algorithm outputs "Yes", then n is indeed prime. *Proof by contradiction*:

Let's assume $\exists n > 1$ such that the algorithm outputs "Yes" but n is in fact composite. So, n has a non-trivial factor p_1, i.e., $1 < p_1 < n$.

If $n == 2$ or $n == 3$, clearly n is prime and our output is correct. So, $n > 3$.

if $p_1 <= \lfloor \sqrt{n} \rfloor$, then output cannot be "Yes" because we brute force checked divisibility of each of these numbers with n before outputting a "Yes". This violates the working of the algorithm since $n \neq 2, n \neq 3$ was already there. Thus existence of such an integer p_1 is impossible and we know factors are integers.

Consider $p_1 > \lfloor \sqrt{n} \rfloor$. So, $p_1 \geqslant \sqrt{n}$. Since p_1 is a factor of n, $\exists p_2$ such that $p_1 * p_2 = n$. This implies that $p_2 \leqslant \sqrt{n}$.

Since p_2 is an integer, we get $p_2 \leqslant \lfloor \sqrt{n} \rfloor$, which is a contradiction!

b) The given claim is incorrect, i.e., the algorithm does not run in polynomial time (w.r.t input size). It can be seen easily that the algorithm has a runtime of $O(\sqrt{n})$.

However, the input integer n can be represented using $O(\log_2 n)$ bits. So, the input size is actually $O(\log n)$, whereas the runtime complexity is $O(\sqrt{n})$. So, for an input $\mathcal{I}$ consisting of $|\mathcal{I}|$ bits, the given algorithm runs in $O(\sqrt{2}^{|\mathcal{I}|})$ time, which is exponential w.r.t input size, and hence, not polynomial.

Note: Note that the time complexity of the given algorithm varies with the representation of n, i.e., suppose our input for n was in unary system instead of binary. Then, we will need n bits to represent the input n (as compared to $O(\log n)$ bits when we represented n in binary). In this case, $O(\sqrt{n})$ actually becomes polynomial time. These types of algorithms, whose time complexity changes from non-polynomial time to polynomial time by changing the representation of the input in a different base, are called *pseudo-polynomial* time algorithms. These are not polynomial time algorithms, as we generally assume that the input is given in binary form.

a)
- Written "I do not know how to approach this problem" - 0.3 points
-
  - Claiming that algorithm is correct - 0.5 points
  - Proving that the algorithm is correct - 1 point

b)
- Written "I do not know how to approach this problem" - 0.3 points
-
  - Claiming that the algorithm is not polynomial time - 0.5 points
  - Providing the correct reason for the claim - 1 point

2. Consider the following algorithm for calculating a^b where a and b are positive integers.

ALGORITHM 2: `FastPower`(a,b)

Input: Positive integers a and b.
Output: a^b.

if $b = 1$ **then**
  return a
$c := a \cdot a$
$d :=$ `FastPower`$(c, \lfloor b/2 \rfloor)$
if b *is odd* **then**
  return $a \cdot d$
return d

Suppose each multiplication and division operation can be performed in constant time. Determine the asymptotic running time of `FastPower` as a function of b.

Let $T(b)$ denote the asymptotic running time of `FastPower`. Since each multiplication and division can be performed in constant time, we can write the recursive relation for

$T(b)$ as:

$$T(b) = \begin{cases} O(1) & \text{if } b = 1, \\ T\left(\frac{b}{2}\right) + O(1) & \text{if } b \text{ is even,} \\ T\left(\left\lfloor \frac{b}{2} \right\rfloor\right) + O(1) & \text{if } b \text{ is odd.} \end{cases}$$

Hence, the recursive relation is;

$$T(b) = T\left(\left\lfloor \frac{b}{2} \right\rfloor\right) + O(1)$$

As $O(1)$ refers to a function upper bounded by a fixed constant $c > 0$, we can write:

$$T(1) \leqslant c$$
$$T(b) \leqslant T\left(\left\lfloor \frac{b}{2} \right\rfloor\right) + c$$

Claim 1. *For any integer $i \geqslant 1$, let $k > 0$ be an integer such that $2^{k-1} \leqslant i < 2^k$. Then, we claim that $T(i) \leqslant c \cdot k$.*

Proof. We will prove this by (strong) *induction*.
Induction Hypothesis (P(i)): For any $i > 0$, let $k > 0$ be an integer such that $2^{k-1} \leqslant i < 2^k$. Then, $T(i) \leqslant c \cdot k$.
Base Case (P(1)): It can be seen that $2^0 \leqslant 1 < 2^1$ and hence, $k = 1$. We already know that $T(1) \leqslant c$. Hence, Base Case holds true.
Inductive step ($P(i) \implies P(i+1)\ \forall i \geqslant 1$): Let k be such that:

$$2^{k-1} \leqslant i + 1 < 2^k$$
$$\implies 2^{k-2} \leqslant \frac{i+1}{2} < 2^{k-1}$$
$$\implies 2^{k-2} \leqslant \left\lfloor \frac{i+1}{2} \right\rfloor < 2^{k-1}$$

Since, $i \geqslant 1$, so $i > \lfloor \frac{i+1}{2} \rfloor \geqslant 1$ and hence, by induction hypothesis, $P(\lfloor \frac{i+1}{2} \rfloor)$ holds true. Hence, $T(\lfloor \frac{i+1}{2} \rfloor) \leqslant (k-1) \cdot c$. This means that:

$$T(i+1) \leqslant T\left(\left\lfloor \frac{i+1}{2} \right\rfloor\right) + c$$
$$\implies T(i+1) \leqslant (k-1) \cdot c + c$$
$$\implies T(i+1) \leqslant k \cdot c$$

Hence, the claim holds true. □

By the above claim, we note that $T(b) \leqslant c \cdot k$ for $2^{k-1} \leqslant b < 2^k$ and hence, $k \leqslant \log_2 b + 1$. Hence, $T(b) \leqslant c \cdot (\log_2 b + 1)$, which means that $T(b) = O(\log b)$.

3. Let A and B be two sorted arrays of length n each. Assume that all elements within and across the two arrays are distinct. Design an $\mathcal{O}(\log n)$ algorithm to compute the n^{th} smallest element of the union of A and B.

For an array A, we'll denote by $A[i]$ the i^{th} element of A (assuming 1-based indexing). Also, $A[: i])$ denotes the subarray consisting of first i elements of A and $A[i :]$ denotes the subarray from i^{th} element to last element of A. We give an algorithm, which for any k, outputs the k^{th} smallest element in the union of A and B.

ALGORITHM 3: `kthsmallest(`A, B, k`)`

Input: Two sorted arrays A and B, and an integer k.
Output: kth smallest element in the union of A and B.

if A *is* *empty* **then**
  return $B[k]$
else if B *is* *empty* **then**
  return $A[k]$
$midA \leftarrow \left\lfloor \frac{\text{len}(A)}{2} \right\rfloor$
$midB \leftarrow \left\lfloor \frac{\text{len}(B)}{2} \right\rfloor$
if $midA + midB < k$ **then**
  if $A[midA] > B[midB]$ **then**
    return `kthsmallest(`A, $B[midB + 1 :]$, $k - midB$`)`
  else
    return `kthsmallest(`$A[midA + 1 :]$, B, $k - midA$`)`
else
  if $A[midA] > B[midB]$ **then**
    return `kthsmallest(`$A[: midA]$, B, k`)`
  else
    return `kthsmallest(`A, $B[: midB]$, k`)`

Proof of Correctness: We will use (strong) induction to prove the claim.

Inductive Hypothesis $(P(n))$*:* If $length(A) + length(B) \leqslant i$, `kthsmallest(`$A$, B, k`)` returns the correct answer.
Base Case: $length(A) + length(B) = 1$: Here, either $length(A) = 0$ or $length(B) = 0$. It is easy to see that the algorithm gives the correct output if $length(A) = 0$ or $length(B) = 0$. Hence, base case holds.
Inductive Step: Consider $length(A) + length(B) = i + 1$. Since the algorithm returns the correct output if $length(A) = 0$ or $length(B) = 0$, we'll assume that $length(A), length(B) > 0$. Also, denote the k^{th} smallest number of union of A and B by n_k. Also, the middle elements of A and B are denoted by $A[midA]$ and $B[midB]$ respectively.

Assume $midA + midB < k$ (proof for the other case will follow similarly). WLOG, we can assume $B[midB] < A[midA]$. In this case, we claim the following:

Claim 2. $n_k > B[midB]$

Proof. By Contradiction: Assume n_k lies in $B[: midB]$, i.e, $n_k = B[j]$, where $1 \leqslant j \leqslant midB$. This means that $n_k \leqslant B[midB]$. Since $B[midB] < A[midA]$, this implies that $n_k < A[midA]$ and so, n_k is smaller than all elements of $A[midA :]$. So, the maximum number of elements smaller than n_k in union of A and $B \leqslant (j-1) + midA \leqslant midA + midB - 1 < k - 1$ and hence, n_k cannot be the k^{th} smallest number. Hence, we obtain a contradiction. □

Due to the above claim, it follows that we can throw away $B[: midB]$ and so, n_k is equal to the $(k - midB)^{th}$ smallest element in the union of remaining arrays, i.e., A and $B[midB + 1 :]$. Since $midB \geqslant 1$, $length(A) + length(B[midB + 1 :]) = length(A) + length(B) - length(B[: midB]) \leqslant i$. Hence, by Induction Hypothesis, `kthsmallest(`A`,` $B[midB+1:]$`,` $k - midB$`)` returns the correct answer and hence, `kthsmallest(`A`,` B`,` k`)` returns the correct answer. Hence, proved.

Time Complexity: In every recursive call, we do constant amount of work and length of either A or B is halved (or even lesser than halved). Since A and B are of length n initially, they can be halved $O(\log_2 n)$ times each. So, total calls made = $O(\log n)$ and hence, time complexity = $O(\log n)$.

- Written "I do not know how to approach this problem" - 0.6 points
-
  - Correct algorithm - 1.5 points
  - High level proof ideas correct - 1 point
  - Proving time complexity - 0.5 points

4. Design an $\mathcal{O}(\log^2 n)$ algorithm that, given a positive integer n, determines whether n is of the form a^b for some positive integers a and $b > 1$. For the purpose of this problem, you may assume exponentiation to be $O(1)$ time, i.e., computing p^q for two integers p and q takes constant time. Similarly, you can assume that comparison of two integers (i.e., determining whether p equals q, $p > q$ or $p < q$) takes constant time.

Note that we can assume $a > 1$ as well since its positive and for all non trivial cases, $a == 1$ is not useful.
Hint: Would binary search help? Binary search on what?
Solution

ALGORITHM 4: `IsPowerOfForm(n)`

Input: A positive integer n.

Output: True if n is of the form a^b for some positive integers a and $b > 1$, False otherwise.

```
if n ⩽ 1 then
    return False
b ← 2
while 2^b ⩽ n do
    Binary Search on a: low ← 2
    high ← n
    while low ⩽ high do
        mid ← ⌊(low+high)/2⌋
        power ← mid^b
        if power == n then
            return True
        else if power < n then
            low ← mid + 1
        else
            high ← mid − 1
    b ← b + 1
return False
```

Correctness Idea: Note that the function x^b is an increasing function for a fixed b and hence, we can perform a binary search to find whether $n = x^b$ for some b. Since we are searching over all b such that $2^b \leqslant n$, if $n = a^b$, we will indeed find such a and b. We do not search for the values of b such that $2^b > n$ since it is clear that $a^b > n$ for any such b.

Time Complexity: Since we search over $O(\log n)$ values of b and for each b, the binary search takes $O(\log n)$ time, we get a total time complexity of $O(\log^2 n)$.

- Written "I do not know how to approach this problem" - 0.6 points
-
  - Correct algorithm - 1.5 points
  - High level proof ideas correct - 1 point
  - Proving time complexity - 0.5 points

5. You are given a sorted (from smallest to largest) array A of n distinct integers which can be positive, negative, or zero. You want to decide whether or not there is an index i such that $A[i] = i$. Design the fastest algorithm you can for solving this problem.

A naive solution is to iterate over the entire array. Can we do better?
Hint: Consider an arbitrary index i. What happens when $A[i] > i$? If it is equal to i, we are done. What about $A[i] < i$. Note that elements are distinct and array is sorted.

Solution

ALGORITHM 5: `FindFixedPoint(`A`)`

Input: A sorted array A of n distinct integers.
Output: Index i such that $A[i] = i$, or -1 if no such index exists.

$left \leftarrow 0$
$right \leftarrow n - 1$
while $left \leqslant right$ **do**
  $mid \leftarrow \lfloor \frac{left+right}{2} \rfloor$
  if $A[mid] == mid$ **then**
    return mid
  else if $A[mid] > mid$ **then**
    $right \leftarrow mid - 1$
  else
    $left \leftarrow mid + 1$
return -1 (indicating no such index exists)

Correctness Idea: Note that if we find an index i such that $A[i] > i$, then $A[j] > j \, \forall j \geqslant i$. This is because A is sorted and all elements of A are distinct. Hence, if $A[i] > i$, then $A[i+1] \geqslant A[i] + 1 > i + 1$, $A[i+2] \geqslant A[i+1] + 1 > i + 2$ and so on.
So, if $A[i] > i$, and an index k exists such that $A[k] = k$, then $k < i$, so we only need to search in the sub-array $A[: i-1]$. Similar idea holds if we find an i such that $A[i] < i$.

Time Complexity: This is identical to binary search, and hence, it takes $O(\log n)$ time.

- Written "I do not know how to approach this problem" - 0.6 points
-
  - Correct algorithm - 1.5 points
  - Proof of Correctness - 1 point
  - Proving time complexity - 0.5 points

6. ($\star$) You are given an n-by-n grid of distinct numbers. A number is a *local minimum* if it is smaller than all its neighbors. A *neighbor* of a number is one immediately above, below, to the left, or to the right. Most numbers have four neighbors; numbers on the side have three; the four corners have two.

   (a) Prove that a local minimum always exists.

   (b) Use the divide-and-conquer algorithm design paradigm to compute a local minimum with only $\mathcal{O}(n)$ comparisons between pairs of numbers. (Note: since there are n^2 numbers in the input, you cannot afford to look at all of them.)

(a) Let x be the minimum number in the grid, then x will be smaller than all its neighbors because all the numbers in the grid are distinct. Hence, a local minimum always exists.

(b) **Informal Idea:** We use divide-and-conquer to solve the problem. We divide the problem into $\frac{n}{2} \times \frac{n}{2}$ submatrices and find the minimum of the elements in the boundaries of the submatrices. If minimum is indeed the local minimum we return else we find the local minimum in the submatrix containing minimum element.

ALGORITHM 6: Find Local Minimum in $n \times n$ Grid

Input: $n \times n$ grid of distinct numbers
Output: A local minimum

```
1  Function FindLocalMinimum(grid, n):
2      if n == 1 then
3          return grid[1][1] // Base case: Single element is a local minimum
4      Divide the grid into four submatrices A_1, A_2, A_3, A_4 of size n/2 × n/2
5      Define boundaries b_1, b_2, b_3, b_4 for submatrices A_1, A_2, A_3, A_4
6      for each i ∈ {1,2,3,4} do
7          b_i ← Union of the first row, last row, first column, and last column of A_i
8      m ← minimum element over b_1, b_2, b_3, b_4
9      if Check(m) then
10         return m
11     else
12         for each i ∈ {1,2,3,4} do
13             if m lies in boundary b_i then
14                 return FindLocalMinimum(A_i, n/2) // Recursively search in the
                       submatrix containing m
15 Function Check(element):
       // Check if the element is a local minimum by comparing with its
          neighbors
16     Compare element with its neighbors in the grid
17     if element is smaller than all its neighbors then
18         return True // Element is a local minimum
19     return False // Element is not a local minimum
```

Claim 3. *In Line 13 of the Algorithm, if m lies in the boundary b_i, then a local minimum of A exists in A_i.*

Proof. Note that m has at most 1 *neighbour* not lying on any of b_1, b_2, b_3 and b_4. If such a *neighbour* does not exist, then m is already a local minimum. So, we assume that such a *neighbour* exists (let's call it n_m). As m lies in b_i, n_m also lies in A_i. Consider the minimum element of A_i (let's call it m'). Then, there are 2 cases:

a) $m' = m$: Since all elements of A are distinct, this means that $m < n_m$. We know that m is already smallest of all the boundary elements. Also, n_m is the only *neighbour* of m not lying on any of the boundaries and $m < n_m$. Hence, m is smaller than all its *neighbours* and hence, is a local minimum.

b) $m' < m$: Since m is the smallest element over all boundaries, this means m' does not lie on the boundary b_i and hence, all *neighbours* of m' lie in A_i. Since m' is the smallest element of A_i and all *neighbours* of m' lie in A_i, m' is smaller than all of its *neighbours* and hence, is a local minimum of A.

So, a local minimum of A exists in A_i in both the cases. □

Proof of Correctness: We prove the above algorithm gives the local minimum in the grid using strong induction:

Inductive Hypothesis $(P(n))$*: FindLocalMinimum(grid, n)* returns a local minimum in the grid.
Base Case: For $n = 1$, the algorithm returns *grid[1][1]* which is a local minimum.
Inductive Step: Consider the grid of size $n \times n$. The algorithm first finds the minimum element m over all the boundaries b_i of submatrices A_i of size $(\frac{n}{2} \times \frac{n}{2})$. Following two cases are possible on the minimum element m:

a) If m is less than all of it's local neighbors, then it is a local minimum and algorithm returns it.

b) If m is not a local minimum then the algorithm recurses over the grid A_i containing m. The Claim 3 proves that the grid A_i contains a local minimum of A. By strong induction we can argue that algorithm returns a local minimum in the grid A_i (which is of size $\frac{n}{2} \times \frac{n}{2}$) and hence, we obtain the local minimum of A.

Time Complexity: In a recursive call, calculating and checking whether the minimum m lies in b_i takes $O(n)$ time and if m is not the local minimum then in next recursive call n is halved.

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

By masters theorem, T(n) = $O(n)$.

7. ($\star$) You are given a sequence of n numbers $a_1, a_2, \ldots, a_n$. Design an $\mathcal{O}(n)$ algorithm to find i, j with $i \leqslant j$ such that the sum $a_i + a_{i+1} + \cdots + a_j$ is maximum. Note that the numbers may not be positive.

The high-level idea is to find the maximum contiguous sum ending at and including index i for $1 \leqslant i \leqslant n$ (assuming 1-based indexing) and taking the maximum over all these contiguous sums. To find the maximum contiguous sum ending at and including index i for $i \geqslant 2$, we divide it into 2 cases

- If a_{i-1} is included in the contiguous array, the maximum possible sum will be a_i + maximum contiguous sum ending at a_{i-1}.
- If a_{i-1} is not included in the contiguous array, then the contiguous array ending at a_i consists only of the element a_i.

So, we take the maximum over the above cases to find the maximum contiguous sum ending at and including index i for $i \geqslant 2$.

Claim 4. *Define M_i as the maximum contiguous sum ending at and including index i for $1 \leqslant i \leqslant n$. Then we have $M_1 = a_1$ and $M_i = \max(M_{i-1} + a_i, a_i)$.*

Proof. It's clear that $M_1 = a_1$ since a_1 is the only contiguous sum ending at index 1. Consider M_i for some $2 \leqslant i \leqslant n$.
Denote by C_i the set of contiguous sums ending at index i and denote by C_{i-1} the set of contiguous sums ending at index $i-1$.
Then we have:

$$\begin{aligned}
C_i &= \{x + a_i | x \in C_{i-1}\} \cup \{a_i\} \\
M_i &= \max(C_i) = \max(\{x + a_i | x \in C_{i-1}\} \cup \{a_i\}) \\
M_i &= \max(\{x + a_i | x \in C_{i-1}\}, \{a_i\}) \\
M_i &= \max(M_{i-1} + a_i, a_i)
\end{aligned}$$

□

From the above claim, the following algorithm is immediate: Progress through the array A, calculating the maximum contiguous sum ending at and including index $i \in \{2, 3, \ldots, n\}$ using the previous maximum contiguous sum ending at and including index $i-1$.
The maximum subarray sum is $\max(M_i)\ \forall i$ and hence, can be found in $O(n)$ time.

Alternative Approach: We can find the maximum sum subarray using divide and conquer approach also, albeit at a slightly higher time complexity. The approach is as follows:

Divide the array into two halves A_1 and A_2. The maximum sum subarray would be either in (i) A_1, or (ii) A_2 or (iii) it starts in A_1 and ends in A_2 i.e crossing the two halves. The maximum sum subarray will be the maximum of these 3 cases. First 2 cases can be calculated using recursion.

For the third case, we can easily find the crossing sum in linear time. The idea is simple, find the maximum sum starting from mid point and ending at some point on left of mid, then find the maximum sum starting from mid + 1 and ending with some point on right of mid + 1. Finally, combine the two and return the maximum among left, right and combination of both.

Time Complexity:

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n)$$
$$T(n) = O(n \log n)$$