Instructor:
Eleni Drinea
CSOR W4246–Fall, 2021

Student:
Vishweshwar Tyagi
vt2353@columbia.edu

Homework 1 Theoretical (110 points)

Out: Monday, September 20, 2021
Due: 11:59pm, Monday, October 4, 2021

Homework Problems

1. (20 points)

   (a) (5 points) Suppose that when doing a Mergesort you recursively break lists into 4 equal-sized sublists (or as close to equal as possible) instead of 2. What is the asymptotic running time of this modified Mergesort?

   Solution (a) Here we're simply breaking down our problem into four (roughly) equi-sized subproblems. We first break our original array (say A), into four (roughly) equi-sized contiguous subarrays. We then call the same procedure on these subarrays which sorts them one after another. Call the resultant sorted subarrays as $A_1,\ A_2,\ A_3$ and A_4. We then have to merge these subarrays into one non-decreasing array. We do this by first merging A_1 and A_2 with each other (both of which are of approximate length $\frac{n}{4}$) and A_3 and A_4 with each other (both of which are of approximate length $\frac{n}{4}$). Then we merge the resulant two subarrays (both of which are of approximate size $\frac{n}{2}$) into a single non-decreasing array and return.

   Suppose $T(n)$ denotes the time-complexity of this merge-sort for input list/array of length n. We know that merging two non-decreasing arrays of length $O(n)$ takes $O(n)$ time and we have to perform such merges only a constant number of times (3 to be precise). Hence, we have

   $$T(n) = 4T(\frac{n}{4}) + O(n)$$
   $$\rightarrow T(n) = O(n \log n) \tag{1}$$

   where (1) follows from the Master Theorem.

   (b) (15 points) Give a (deterministic) $\Theta(n \log n)$ algorithm that, on input a set S of n integers and another integer x, determines whether or not there exist two elements in S whose sum is exactly x.

Solution (b) We will first sort the array S in non-decreasing order. Once sorted, we simultaneously start iterating from both ends of the array and move inwards. At each iteration, we only move inwards once. The rule that governs which side, left or right, moves inwards depends on whether the sum of the two elements (indexed by the left and right pointers) is less or more than x, respectively. If at any moment we find sum equal to x, we return True, else, we iterate until left and right pointers coincide, in which case we return False.

pseudocode

```
1  procedure sum-equal-x(S[1...n], x):
2      S = merge-sort(S)   # sorts in non-decreasing order
3      i = 1, j = LEN(S)
4
5      while j - i > 0:
6          if S[i] + S[j] > x:
7              j -= 1
8          else if S[i] + S[j] < x:
9              i += 1
10         else:
11             return True
12         end if
13     end while
14
15     return False
```

Correctness

Loop Invariant for $[5-13]$: For each iteration of the while loop indexed by $n-(j-i)$ for ordered pair (i,j), we either return True if we find our desired pair (such that $S[i]+S[j]=x$) or conclude at the end of this iteration that $\nexists\ (a,b)$ such that $a+b=x$ where $a \in S[1...i]$ and $b \in S[j...n]$

Base Case: $n-(j-i)=1 \iff i=1$ and $j=n$

One and only one of these is possible: $S[i]+S[j]=x$, or $S[i]+S[j] \neq x$. If $S[i]+S[j]=x$, $[10-11]$ returns True and our loop invariant holds. Otherwise, we either have $S[i]+S[j] > x$ or $S[i]+S[j] < x$. In any case, it implies that $\nexists\ (a,b)$ such that $a+b=x$ where $a \in S[1...i]=S[i]$ and $b \in S[j...n]=S[n]$ and again our loop invariant holds.

Assume that it holds for $n-(j-i)=l>1$ and WLOG suppose we move inwards from the left. We now want to prove for $n-(j-(i+1))=l+1$ corresponding to ordered pair $(i+1,j)$

Suppose our loop invariant fails for iteration indexed by $n-(j-(i+1))=l+1$ for ordered pair $(i+1,j)$. This means that we didn't return True during this iteration and at the end of this iteration, it is true that $\exists(a,b)$ such that $a+b=x$ where $a \in S[1...(i+1)]$ and $b[j...n]$. However, our hypothesis states that $\nexists(a,b)$ such that $a+b=x$ where $a \in S[1...i]$ and $b \in S[j...n]$. This means that $a=S[i+1]$ and $b \in S[j...n]$. $b=S[j]$ contradicts the fact that we didn't return True during this iteration. So, $b \in S[j+1...n]$ but this is not possible either because our right pointer had to come down to j so the following must hold:

$$\forall q \in \{j+1, j+2 \dots n\} \;\; \exists i_q \in \{1,,2 \dots i\} \text{ such that } S[i_q]+S[q]>x$$

and hence,

$$\forall q \in \{j+1, j+2 \dots n\} \;\; S[i+1]+S[q]>x \text{ because } S \text{ is non-decreasing.}$$

therefore, $q \notin S[j+1, \dots n]$ either. This shows our loop invariant must hold as its negation can't hold.

Termination: Unless we return True at any point before, the while loop will run successfully for the last time when $i+1=j$, i.e, for ordered pair (i,j), when it is indexed by $n-(j-i)=n-1$ $\big($ $\iff$ $i=j+1$ because $i<j$ $\big)$. We either return True during this iteration or conclude at its end that $\nexists(a,b)$ with $a+b=x$ and $a \in S[1 \dots i]$, $b \in S[i+1, \dots n]$, which means required pair doesn't exists. This establishes our proof.

Time-Complexity

$\{2\} \to O(n \log n)$ because of merge-sort

$[5-13] \to O(n)$ because for every iteration of this while loop, we move inwards only once and in the worst case we will have to iterate $(n-1)$ times (or check loop condition n times)

$\{4, 13\} \to O(1)$

So, if $T(n)$ denotes time complexity of our procedure, we have

$$\begin{aligned} T(n) &= O(n \log n) + O(n) + O(1) \\ \to T(n) &= O(n \log n) \end{aligned}$$

2. (20 points) Consider the following high-level description of a recursive algorithm for sorting. On input a list of n distinct numbers, the algorithm runs in three phases. In the first phase, the first $\lceil 2n/3 \rceil$ elements of the list are sorted recursively; the recursion bottoms out when the list has size 1 in which case you do nothing, or if the list has size 2, in which case you return the list if it is ordered, otherwise you swap the elements and return the resulting list. In the second phase, the last $\lceil 2n/3 \rceil$ elements are sorted recursively. Finally, in the third phase, the first $\lceil 2n/3 \rceil$ elements are sorted recursively again.

   Give pseudocode for this algorithm, prove its correctness and derive a recurrence for its running time. Use the recurrence to bound its asymptotic running time. Would you use this algorithm in your next application to sort?

   Solution: The idea is to push larger elements to the back in first phase, smaller elements to the front in second phase and again push larger elements to the back in third phase. Doing so with atleast $\lceil \frac{2n}{3} \rceil$ first and last elements ensures that there is enough overlap to sort our list. After giving the pseudocode, we prove this strategy for $n = 3^k$ $k \geq 0$ as it immensely simplifies our argument which can further be extended to a more general case.

   pseudocode

```
procedure recursive-sort(A[1, 2...n], i, j):
    if i = j:
        return
    end if

    if i+1 = j:
        if A[i] > A[j]:
            swap(A[i], A[j])         # in place
        end if
        return
    end if

    k = ceil[(2/3)*(j-i+1)]
    recursive-sort(A, i, i+k-1)
    recursive-sort(A, j-k+1, j)
    recursive-sort(A, i, i+k-1)

    return

```

Correctness

We show that our procedure sorts all arrays of size n by carrying out induction on

Base Case: For $n = 1$, and our procedure correctly returns A.

Hypothesis: Assume that our procedure sorts arrays of all sizes $< n$ (strong induction). We now prove that it also sorts arrays of size n. WLOG assume that $n = 3^k$ for some $k \geq 1$. This assumption is reasonable as it immensely simplifies our argument and the same argument can also be extended to a more general case.

Claim 1: The procedure places the largest 3^{k-1} (one-third) elements of A at their optimal positions at the end of the list

Proof: Note that $\{14\}$ operates on the first two-third elements and sorts them by hypothesis. This means that after $\{14\}$ is done, the middle 3^{k-1} elements are not only internally sorted but also $\geq$ than the first 3^{k-1} elements. Now, $\{15\}$ operates on the last two third elements (which include the middle 3^{k-1} elements too, leaving first one-third untouched) and sorts them, again by hypothesis. This means that after $\{15\}$ is done, the last 3^{k-1} elements are not only internally sorted but also $\geq$ than rest of the elements. Hence, the largest 3^{k-1} elements assume their optimal positions at the end. Now note that $\{16\}$ doesn't touches the last one-third elements, and therefore, our claim is justified.

Claim 2: The procedure places the smallest 3^{k-1} elements of A at their optimal positions at the start of the list

Proof: Arguing in a similar fashion, after $\{15\}$, the middle 3^{k-1} elements are internally sorted and $\leq$ than the last 3^{k-1} elements. After $\{16\}$, the first 3^{k-1} elements are internally sorted and $\leq$ rest of the elements.

Claim 3. Procedure sorts arrays of size $n = 3^k$

Proof: The procedure places the largest one-third (3^{k-1}) elements and the smallest one-third elements at their optimal positions at the end and start of the array respectively. However, the procedure also terminates after $\{16\}$, which sorts the remaining one-third elements (in middle) internally too, by hypothesis. **QED**

Time-Complexity

If $T(n)$ denotes the time-complexity of our procedure for input list of length n, then,

$$T(n) = 3T(\lceil \frac{2n}{3} \rceil) + O(1)$$
$$\rightarrow T(n) = 3T(\lceil \frac{n}{3/2} \rceil) + O(n^0)$$

Using the Master Theorem, we get

$$T(n) = O(n^{\log_{1.5}(3)})$$
$$\rightarrow T(n) = O(n^{2.709})$$

Note that, $T(n) \notin O(n^2)$ but $O(n^2) \subsetneq O(n^{2709})$. Hence, I would not use this algorithm for sorting as I can do much better that using merge-sort or quick-sort.

3. (35 points) Matrix multiplication is a fundamental primitive in numerical linear algebra and data science.

   (a) (10 points) Recall the traditional matrix multiplication algorithm: When you multiply an $m \times n$ matrix A by an $n \times p$ matrix B you obtain an $m \times p$ matrix C such that entry c_{ij} is given by the dot product of the i-th row of A and the j-th column of B, for every $1 \leq i \leq m$, $1 \leq j \leq p$. In symbols, $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$.

   Give pseudocode for this algorithm. (You do not need to analyze its correctness.) Then analyze the total number of scalar multiplications and the total number of scalar additions required to compute C. What is the running time of the algorithm in asymptotic notation?

   Solution(a) We know that $A_{m \times n}\ B_{n \times p} = (AB)_{m \times p} = C_{m \times p}$

   We then have for $1 \leq i \leq m,\ 1 \leq j \leq p, \quad c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$

   We will use this formula to calculate all the mp entries of C

pseudocode

```
1  procedure mat-mul(A, B, m, n, p):
2      initialize C          # (m x p) matrix
3
4      for i = 1 to m:
5          for j = 1 to p:
6              C[i, j] = A[i, 1]*B[1, j]
7              for k = 2 to n:
8                  C[i, j] += A[i, k] * B[k, j]
9              end for
10         end for
11     end for
12
13     return C
14
```

From $[6-9]$, it takes exactly n scalar multiplications and $(n-1)$ scalar additions to compute $C[i,j]$, the $(i,j)^{\text{th}}$ entry of $C \qquad \forall\, 1 \le i \le m, 1 \le j \le p$

Hence, it takes a total of (mnp) scalar multiplications and $m(n-1)p$ scalar additions to compute C, which is same as the the number of scalar multiplications and additions in our formula above.

Time-Complexity

Suppose that $\{6\}$ take c_6 constant time and $\{8\}$ takes c_8 constant time, and assume in the worst case that it takes $O(mp)$ time to initialize C in $\{2\}$

Let $T(m,n,p)$ denote the time-complexity of multiplying two matrices $A_{m\times n}$ and $B_{n\times p}$, we have

$$T(m,n,p) = O(mp) + \sum_{i=1}^{m}\sum_{j=1}^{p}\left(c_6 + \sum_{k=2}^{n} c_8\right)$$

$$\rightarrow T(m,n,p) = O(mnp) \quad \because n \ge 1$$

(b) (5 points) Suppose you need to compute the product $A \cdot B \cdot x$, where A and B are matrices with dimensions $m \times n$ and $n \times p$ respectively, and x is a $p \times 1$ vector with $m, n, p \gg 1$. How would you compute this product and why?

Solution (b) There are only two possibilities, we either compute $(AB)x$ or $A(Bx)$

Time complexity for computing AB is $O(mnp)$ and for computing $(AB)x$, it will be $O(mnp) + O(mp)$
On the other hand, time complexity for computing Bx is only $O(np)$ and for computing $A(Bx)$, it will be $O(np) + O(mn)$

Since m, n, p >>1,

$$O(mp) + O(mn) \subsetneq O(mnp) + O(mp) \text{ but } \; O(mnp) + O(mp) \quad \not\subset O(mp) + O(mn)$$

Hence, I would compute $A(Bx)$ instead of $(AB)x$

(c) (5 points) Show that 5 integer multiplications suffice to compute the square of a 2×2 matrix A.

Solution (c)

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^2 = \begin{bmatrix} a^2 + bc & b(a+d) \\ c(a+d) & bc + d^2 \end{bmatrix}$$

Hence, exacly 5 integer multiplcations are required: a^2, d^2, bc, $b(a+d)$, and $c(a+d)$

(d) (15 points) In fact, squaring matrices is no easier than matrix multiplication. Prove that, if $n \times n$ matrices can be squared in time $O(n^c)$, then any two $n \times n$ matrices can be multiplied in time $O(n^c)$.

Solution (d) Let $A_{n\times n}$ and $B_{n\times n}$ be two square matrix of $(n \times n)$ size. Consider the following anti-diagonal block formation of these matrices:

$$P = \begin{bmatrix} 0_{n\times n} & A_{n\times n} \\ B_{n\times n} & 0_{n\times n} \end{bmatrix}_{2n\times 2n}$$

where $0_{p\times q}$ represents a $(p \times q)$ matrix with all its entries set to 0. Using rules of block matrix multiplication, we have

$$P^2 = \begin{bmatrix} (AB)_{n\times n} & 0_{n\times n} \\ 0_{n\times n} & (BA)_{n\times n} \end{bmatrix}_{2n\times 2n} \tag{1}$$

According to question, we can calculate LHS of (1) in $O((2n)^c) = O(n^c)$ time.
So, we can calculate the RHS of (1) in $O(n^c)$ time too.
Assuming constant time access, we can get $AB = P^2[1:n, 1:n]$, and hence, we can calculate AB in $O(n^c)$ time as well.

4. (35 points) You are given a sequence of n distinct numbers $x_1, \ldots, x_n$ and want to understand how far this sequence is from being in ascending order.

   One way to define a measure for this is by counting how many pairs of numbers x_i, x_j appear "out of order" in the sequence, that is, $x_i > x_j$ but x_i appears *before* x_j.

   We will define the *disorder* of a sequence to be the number of pairs (x_i, x_j) such that $x_i > x_j$ but $i < j$. For example, if the input sequence is $2, 4, 1, 3, 5$, then the pairs $(2, 1)$, $(4, 1)$ and $(4, 3)$ are out of order and the *disorder* of the sequence is 3.

   (a) (10 points) Give a brute force algorithm to compute the *disorder* of an input sequence of size n.

   Solution (a) We traverse the list using two nested loops to check all potential pairs for disorder. These potential pairs are precisely (a, b) such that a comes before b in the list.

   Brute Force Algorithm - pseudocode

```
1  procedure disorder-brute(A):
2      n = LEN(A)
3      n_disorder = 0          # number of disorders
4      for i = 1 to n:
5          for j = i+1 to n:
6              if A[j] < A[i]:     # check for disorder
7                  n_disorder += 1
8              end if
9          end for
10     end for
11     return n_disorder
12
```

   Correctness

   In the above procedure, the outer loop traverses all elements of the list from left to right while the inner loop traverses all the elements that follow after the element indexed by the outer loop, again from left to right. Hence, we encounter all the potential pairs (defined above) and for each such pair, $[6-8]$ checks to see if it's in disorder and increases n_{disorder} by one if it is. Hence, we correctly count the disorder of input list.

Time-Complexity

Let $T(n)$ denote the time-complexity of above procedure for input list A of length n. If c_j denotes the constant time taken by $[6, 8]$ when the inner loop runs for the j^{th} time, we have the following recurrence

$$T(n) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} c_j + C$$
$$T(n) = O(n^2)$$

(b) (25 points) Can you give a faster algorithm to compute the *disorder*?

Solution (b) The idea is same as usual merge-sort, except that while merging two non-decreasing arrays we count the number of new disorders, i.e, pairs (a, b) such that $a > b$ and a is present in the left half whereas b is present in the right half.

Fast Algorithm - pseudocode

```
procedure disorder-merge(L, R):
    # initialize empty array to merge L and R
    # and count new disorders
    M = [], M_disorders = 0

    i, j = 1
    while i < LEN(L) and j < LEN(R):
        if L[i] > R[j]:          # is a disorder
            # all elements of L from index i onwards
            # are in disorder with R[j]

            M_disorders += LEN(L)-i+1
            append M by R[j]
            j += 1
        else:                    # not a disorder
            append M by L[i]
            i += 1
        end if
    end while

    while i < LEN(L): append M by L[i], i += 1  end while
    while j < LEN(R): append M by R[j], j += 1  end while

    return M, M_disorders

```

```
procedure disorder-fast(A):
    if LEN(A) = 1:
        return A, 0

    mid = floor(LEN(A)/2)

    L, L_disorders = disorder-fast(A[1 ... mid])
    R, R_disorders = disorder-fast(A[mid+1 ... LEN(A)])
    M, M_disorders = disorder-merge(L, R)

    return M, L_disorders+R_disorders+M_disorders

```

To establish correctness of procedure disorder-fast(A), it suffices to argue that procedure disorder-merge(L,R) merges two non-decreasing lists L and R into one non-decreasing list M and correctly counts the new disordered pairs defined as (a, b) such that $a > b$ and a is present in L and b is present in R
That it merges correctly into M is trivial from the merge-sort algorithm. We only show that it counts the new disorders correctly by a simple argument. See $[7 - 19]$ code block.

While merging, we compare the current element in L indexed by i and the current element in R indexed by j traversing both L and R in left to right direction.
If $L[i] > R[j]$, then they are in disorder aswell as all elements in L that follow after index i are also in disorder with $R[j]$ as L is non-decreasing. Thus we get $\text{LEN}(L) - i + 1$ new disorders and crucial thing to note is that no element in L preceding $L[i]$ can be in disorder with $R[j]$ because we are traversing L from left to right starting at index 1 and L is non-decreasing. Thus we increase $M_{\text{disorders}}$ by $\text{LEN}(L)-i+1$ and increase j by 1 and move to the next element in R because we have already accounted for all pairs $(a, R[j])$ such that $a > R[j]$ and a is present in L.

If $L[i] \leq R[j]$, then they are not in disorder, infact $L[i]$ is not in disorder with any element in R which follow after index j because R is non-decreasing. Hence we won't find any new disorders with first coordinate as $L[i]$ and second coordinate to the right of $R[j]$ and therefore, we correctly leave $M_{\text{disorders}}$ untouched and increase i by 1.

Since in each iteration of the first while loop, we either increase i or j by 1, the loop is bound to terminate. After this loop has ended, it means that we have exhausted all the elements in either L or R and so we have counted all new disordered pairs (a, b) such that $a > b$ and a is present in L and b is present in R. The remaining while loops are simply appending M by remaining elements to complete merging. **QED**.

Time-Complexity

Let $T(n)$ denote time-complexity of procedure disorder-fast for input list A of length n Since we are adding only $O(1)$ time steps to the original merge-sort algorithm, it is easy to see that

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

Remarks:

- $[x - y]$ or $[x, y]$ refers to code lines starting at x and ending at y for referenced procedure
- $\{x, y, z\}$ refers to code lines x, y and z for referenced procedure and so forth.
- Initial index for lists/arrays is from 1, and are not 0-indexed
- For list/array L, LEN(L) returns length of L in constant time.
- Algorithms studied in lectures have been directly used.