Fall 2021 Rick Ramirez

Computer Science 5300

Advanced Algorithm Design and Analysis

Assignment # 1

Problem 1

Let X(1..n) and Y(1..n) contain two lists of n integers, each sorted in non-decreasing order. Give the best (worst-case complexity) algorithm that you can think for finding
(a) the largest integer of all 2n combined elements.
(b) the second largest integer of all 2n combined elements.
(c) the median (or the nth smallest integer) of all 2n combined elements.
For instance, X = (4, 7, 8, 9, 12) and Y = (1, 2, 5, 9, 10), then median = 7, the nth smallest, in the combined list (1, 2, 4, 5, 7, 8, 9, 9, 10, 12). [Hint: use the concept similar to binary search]

Solution:

(a)

Algorithm 1 Calculates the largest element of two sorted arrays

```
1: procedure largestElement (int X[n], int Y[n])
2:     return max(X[n-1], Y[n-1])
3: end procedure
```

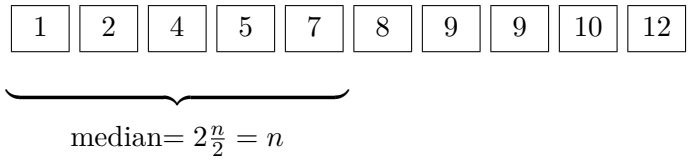
(b)

Algorithm 2 Calculate second largest element of two sorted arrays

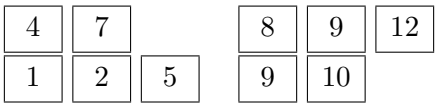
```
1: procedure secondLargest (int X[n], int Y[n])
2:     if X[n-1] == Y[n-1] then
3:         return max(X[n-2], Y[n-2])
4:     else if X[n-1] < Y[n-1] then
5:         return max(X[n-1], Y[n-2])
6:     else
7:         return max(X[n-2], Y[n-1])
8:     end if
9: end procedure
```

(c)

By looking at the two sorted arrays, we see that the median value will correspond to the nth term in the list of length 2n, and that this portion of the list contains part of X[ ] and Y[ ]. Also, note that the last element of any subset of the two sorted lists, starting at index zero, will always correspond to the k^{th} element.



If we look at the two list individually, we find that the elements that lead up to the partition of the k^{th} element are



Let the elements to the left and right of the partition of the first array be l_x and r_x respectively. And similarly, for the second array, l_y and r_y. Notice that for the solution to be valid, ie, for all the elements to the left of the partition to be less than all the elements to the right, requires

$$l_x \leq r_y$$
$$l_y \leq r_x$$

Only these conditions need to be checked because we are guaranteed to have $l_x \leq r_x$ and $l_y \leq r_y$. Once these conditions are satisfied, the k^{th} element will be $max(l_x, l_y)$ because we know that in the combined array, the largest element to the left of the partition corresponds to the k^{th} element. To find the partition, we apply a binomial search to only one array while checking the conditions against both arrays. We start by taking the midpoint of the first array

$$mid1 = \frac{low + high}{2}$$

and let the remaining $mid2 = n - mid1$ elements be taken from the second array, where low and high are initially set to 0 and n respectively. If we find that $l_x > r_y$, we cut out the right half of the array by assigning $high = mid1 - 1$ and if we find that $l_y > r_x$, we remove the left half of the array by assigning $low = mid + 1$. Since the algorithm utilizes a binomial search, the time complexity will be $\mathcal{O}(\log n)$.

Algorithm 3 Calculate median element of two sorted arrays

```
1: procedure MEDIANELEMENT (int X[n], int Y[n])
2:     int low = 0, high = X.length
3:     while low ≤ high do
4:         int mid1 = (low + high)/2,    mid2 = X.length - mid1
5:         lx = X[mid1 - 1]
6:         ly = Y[mid2 - 1]
7:         rx = X[mid1]
8:         ry = Y[mid2]
9:         if lx ≤ ry  &&  ly ≤ rx then
10:            return max(lx, ly)
11:        else if lx > ry then
12:            high = mid1-1
13:        else if ly > rx then
14:            low = mid1 + 1
15:        end if
16:    end while
17: end procedure
```

Problem 2

1-to-2 PARTITION:

Instance: A finite set of positive integers Z = z1 , z2 , ... , zn .

Question: Is there a subset Z' of Z such that Sum of all numbers in Z' = 2 x Sum of all numbers in Z-Z'

(a) Obtain the dynamic programming functional equation to solve the 1-to-2 PARTITION problem.

(b) Give an algorithm to implement your functional equation.

(c) Give an example of 5 numbers with a total of 21 as an input instance for 1-to-2 PARTITION problem, and show how your algorithm works on this input instance.

(d) What is the complexity of your algorithm?

Solution

(a)

Consider the case of $\sum Z = \sum Z - Z'$. For a solution to exist, two conditions must be satisfied:

• The sum of the total number integers $\sum_{i=1}^{n} z_i$ must be even. If it is odd, then the set cannot be divided into two equal subsets.

• If $\exists$ a sum $S_1 = \sum_{i=1}^{m} z_i = \frac{1}{2}\sum_{i=1}^{n} z_i$, where $m < n$, then we are guaranteed to have the remaining

$Z - Z'$ elements equal to $S_2 = \frac{1}{2}\sum_{i=1}^{n} z_i$. The set can be broken into two equal sums.

Consider the example of $Z = \{1, 3, 5, 11\}$. The fist condition passes with $\sum_{i=1}^{n} z_i = 20$ ✓ however, the second condition requires us to find a subset whose sum is equal to $S_1 = \frac{1}{2}\sum_{i=1}^{n} z_i = 10$. The best we can do in this case is $Z_1 = \{1, 3, 5\} \rightarrow S_1 = 9 \neq 10$ and $Z_2 = \{11\} \rightarrow S_2 = 11$ ✗. $\therefore$ this Z input will return false.

Now consider $Z = \{3, 1, 1, 2, 2, 1\}$. The first condition passes with $\sum_{i=1}^{n} z_i = 10$ ✓, and the second condition passes since $Z_1 = \{3, 1, 1\} \rightarrow S_1 = 5$ and $Z_2 = \{2, 2, 1\} \rightarrow S_2 = 5$ ✓. $\therefore$ this Z input will return true.

Initial

Z ↓ \ Sum (S) →	0	1	2	3	4	5
	1	0	0	0	0	0
z_1=3	1					
z_2=1	1					
z_3=1	1					
z_4=2	1					
z_5=2	1					
z_6=1	1					

⇒

Final

Z ↓ \ Sum (S) →	0	1	2	3	4	5
	1	0	0	0	0	0
z_1=3	1	0	0	1	0	0
z_2=1	1	1	0	1	1	0
z_3=1	1	1	1	1	1	1
z_4=2	1	1	1	1	1	1
z_5=2	1	1	1	1	1	1
z_6=1	1	1	1	1	1	1

Starting with the first empty cell of the matrix:
$1 - 3 = \textit{undefined} \rightarrow$ false $\therefore$ we cannot form the sum=1 using integer 3
$2 - 3 = \textit{undefined} \rightarrow$ false $\therefore$ we cannot form the sum=2 using integer 3
$3 - 3 = 0 \rightarrow$ true[1] $\therefore$ we CAN form S=3 using z=3 (populate cell with 1) Note: s(3) = 0 and s(0) = 1
$4 - 3 = 1 \rightarrow$ false Note: s(4) = 0 and s(1) = 0
$5 - 3 = 2 \rightarrow$ false Note: s(5) = 0 and s(2) = 0
The emerging pattern can be summarized as: if the ith-1 cell is equal to true OR the difference between s and z for that particular row and column is true, then we take true. ie if $f(s) = 1 \quad || \quad f(s - z) = 1$. Therefore, our functional equation can be expressed as

$$\boxed{f_i[s, z_i] = max\{f_{i-1}(s), f_{i-1}(s - z_i)\}}$$

If we find the last cell to be true, then it IS possible to form the value $\frac{S}{2}$ with the given assortment of Z. This relation also holds for the case of $\sum Z' = \sum 2(Z - Z')$ since we are just moving the partition while preserving the total sum. However we do need to change the first condition to only allow a target sum divisible by 3, in which case the second condition is changed to $S_1 = \frac{2}{3}\sum_{i=1}^{n} z_i$ and $S_2 = \frac{1}{3}\sum_{i=1}^{n} z_i$.

[1] We take a step up to the previous row and traverse to the left until we reach the column value that is equal to the difference between the sum and integer.

(b)

Algorithm 4 1-to-2 PARTITION

```
1: procedure PARTITION(int Z[n])
2:     int sum = 0, s = 0
3:     for i ← 1 to n do                                          ▷ O(n)
4:         sum += Z[i]
5:     end for
6:     if sum % 3 ≠ 0 then
7:         return false                  ▷ Check that the first condition is satisfied
8:     end if
9:     s = sum / 3
10:    int f[n][s]                                                ▷ Initialize a matrix
11:    for j ← 1 to s do
12:        f[0][j] = 0
13:        f[j][0] = 1
14:    end for
15:    for i ← 1 to n do
16:        for j ← 0 to s do
17:            if j < Z[i-1] then
18:                f[i][j] = f[i-1][j]                            ▷ The cell above
19:            else
20:                f[i][j] = max{f[i-1][j] , f[i-1][j-Z[i-1]]}
21:            end if
22:        end for
23:    end for
24:    return f[n][s]
25: end procedure
```

(c)

Let $Z = 1, 2, 4, 6, 8$, $sum = \sum_{i=1}^{n} z[i] = 21$, and $M = \frac{1}{3} sum = 7$

Z ↓ \ Sum (S) →	0	1	2	3	4	5	6	7
	1	0	0	0	0	0	0	0
z_1=1	1	1	0	0	0	0	0	0
z_2=2	1	1	1	1	0	0	0	0
z_3=4	1	1	1	1	1	1	1	1
z_4=6	1	1	1	1	1	1	1	1
z_5=8	1	1	1	1	1	1	1	1

Since the algorithm halts with a 1 on cell $f(z_5, 7)$, the instance will return *true*. The algorithm can be

slightly improved by returning a *true* at the first occurrence of the target sum $s[7] \leftarrow 1$ because we know at that point that the remaining elements will sum to $\frac{2}{3}$ total sum.

(d)

Since the algorithm must iterate through $\frac{1}{3}sum$ columns and Z rows, the time complexity must be on the order of $O(Tn)$, where $T = \frac{sum}{3}$.

Problem 3

Decide True or False for each of the followings. You MUST briefly justify your answer.

Satisfiability:
Instance: Set U of variables, collection C of clauses over U.
Question: Is there a satisfying truth assignment for C?

(a) If P $\neq$ NP, then no problem in NP can be solved in polynomial time deterministically.
(b) If a decision problem A is NP-complete, proving that A is reducible to B, in polynomial time, is sufficient to show that B is NP-complete.
(c) It is known that SAT (Satisfiability) is NP-complete, and 3SAT (all clauses have size 3) is NP-complete. 1SAT (all clauses have size 1) is also NP-complete.

Solution

(a)

False
The class of problems defined as P consists of the set of decision problems Π that can be solved by a deterministic Turing machine using a polynomial amount of computation time. This class of problems is known to be a subset of NP, that is $P \subseteq NP$, and therefore, every decision problem solvable by a polynomial time deterministic algorithm is also solvable by a polynomial time nondeterministic algorithm. Suppose that the decision problem $\Pi \in P$ has a polynomial deterministic algorithm A. We can obtain a polynomial time nondeterministic algorithm for Π by using A as the checking stage and ignoring the guess. Therefore, $\Pi \in P \implies \Pi \in NP$.

(b)

False

The process of devising an NP-completeness proof for a decision problem Π consists of 4 stages:

i) Show that $\Pi \in NP$
ii) Select a known NP-complete problem Π'
iii) Construct a transformation $f : \Pi' \to \Pi$
iv) Prove that f is a polynomial time transformation

For the given scenario, conditions ii, iii, and iv are satisfied. However it was never stated that $B \in NP$. Therefore we cannot yet say that B is NP-complete.

(c)

False

Let $U = \{u_1, u_2 \dots u_n\}$ be a set of boolean variables and let the function t be the truth assignment of U such that $t : U \to \{T, F\}$. If we allow the set of literals over U to form a clause over U in the form $C = \{\{u_1\}, \{u_2\} \dots \{u_n\}\}$, then a satisfying truth assignment will take the form $t(u_1) = t(u_2) = \cdots = t(u_n) = T$. It a simple matter of scanning C for an instance of $\neg u_i$ from $1 \leq i \leq n$. Therefore, since 1SAT is solvable in a time on the order of $O(n)$ we can say that 1SAT $\in P$.

Problem 4

Given that PARTITION problem (described below) is a NP-Complete problem, prove that the SUM OF SUBSETS problem (described below) is NP-Complete by reducing PARTITION problem to it.

PARTITION:
Instance: A finite set of positive integers $Z = \{z_1, z_2, ..., z_n\}$.
Question: Is there a subset Z' of Z such that
Sum of all numbers in Z' = Sum of all numbers in Z-Z'

SUM OF SUBSETS:
Instance: A finite set of positive integers $A = \{a_1, a_2, \dots, a_m\}$ and M.
Question: Is there a subset A' in A s.t. $\Sigma_{a_i \in A'} a_i = M$?

(a) Give a nondeterministic polynomial time algorithm for the SUM OF SUBSETS problem.
(b) Define the transformation from the PARTITION problem to the SUM OF SUBSETS problem.
(c) Explain that the transformation described in part (b) satisfies: if the partition problem has a solution then the sum-of-subsets has a solution, and vice versa.

Solution

(a)

Algorithm 5 Nondeterministic algorithm for the Sum-of-Subsets problem

```
Guess a subset A' of A
Check that the sum of A' is equal to M
if yes then
    return 1
else
    return 0
end if
```

The algorithm will run in polynomial time since checking the sum of a sequence is on the order of $O(n)$. Since the algorithm utilizes a guessing and is able to verify the guess in polynomial time, we can say that the Sum-of-Subsets problem resides in NP.

(b)

Let $f(\langle Z \rangle)$ be a mapping that takes an instance of the Partition problem $\langle Z \rangle$ and returns an instance of Sum-of-Subsets problem $\langle A, M \rangle$. Let $f()$ create a target sum M from Z such that

$$\frac{1}{2} \sum_{i=1}^{n} z_i \equiv M$$

If we let $n = m$ and let $Z = A$, then $f : \langle Z \rangle \rightarrow \langle A, M \rangle$

(c)

Proof. We can evaluate the correctness of the mapping f() by evaluating the decision problems Π of $\langle Z \rangle$ and $\langle A, M \rangle$. That is, f() is correct iff for every $\Pi_{\langle Z \rangle} = Y$, $\Pi_{\langle A,M \rangle} = Y$, where $Y_\Pi \subseteq D_\Pi$.

- $Y_{\langle Z \rangle} \rightarrow Y_{\langle A,M \rangle}$

  $Y_{\langle Z \rangle}$: Assume that Z can be partitioned into two sets with equal sum.

  $$\sum_{z \in Z'} z \quad = \sum_{z \in Z - Z'} z$$

  $Y_{\langle A,M \rangle}$: For the above to be true, each of these sets sums to half of the total in Z, that is they both equal M, where

  $$\frac{1}{2} \sum_{i=1}^{n} z_i \equiv M$$

  Therefore $A = Z$ has a subset that sums to M.

To complete the correctness proof, we need to map all 'no' outputs of $\Pi_{\langle Z \rangle}$ to the 'no' outputs of $\Pi_{\langle A,M \rangle}$, however this can be accomplished by proving that an instance $Y_{\langle A,M \rangle}$ will map to $Y_{\langle Z \rangle}$, i.e. the reverse of the proof above.

- $Y_{\langle A,M \rangle} \rightarrow Y_{\langle Z \rangle}$
  $Y_{\langle A,M \rangle}$: Assume that A has a subset that sums to $M = \frac{1}{2} \sum_{i=1}^{n} z_i$

  We are guaranteed that the remainder of the elements in A will also be equal to $\frac{1}{2} \sum_{i=1}^{n} z_i$, so
  $Y_{\langle Z \rangle}$: can be partitioned into two sets with equal sum.

□

The above is an example of 'proof by restriction'. The idea consists of showing that Π contains a known NP-complete problem Π' as a special case by imposing a restriction on the instances of Π so that the resulting problem preserves a one-to-one correspondence between the 'yes' and 'no' answers. It's worth noting that the resulting problem need not be an exact duplicate of the original problem so long as the correspondence is obvious.

Problem 5

Prove that the 0/1 KNAPSACK problem is NP-Hard. (One way to prove this is to prove the decision version of 0/1 KNAPSACK problem is NP-Complete. In this problem, we use PARTITION problem as the source problem.)

(a) Give the decision version of the O/1 KNAPSACK problem, and name it as DK.
(b) Show that DK is NP-complete (by reducing PARTITION problem to DK).
(c) Explain why showing DK, the decision version of the O/1 KNAPSACK problem, is NP-Complete is good enough to show that the O/1 KNAPSACK problem is NP-hard.

Solution

(a)

DECISION O/1 KNAPSACK (DK):

Instance: Given a set of items $X = \{x_1, x_2 \dots x_n\}$, each with a set of weights $W = \{w_i, w_2 \dots w_n\}$, and a set of values $V = \{v_1, v_2, \dots v_n\}$, a maximum capacity W_{max} and a target profit V_t.

Question: Is there configuration of items in X such that the following conditions are satisfied:

$$\sum_{i=1}^{n} x_i w_i \leq W_{max} \quad \text{and} \quad \sum_{i=1}^{n} x_i v_i \geq V_t$$

where each $x_i \in \{0, 1\}$

(b)

i) $DK \in NP$

Algorithm 6 Nondeterministic algorithm for DK

1: Guess a sequence for X
2: Check the sums $x_i w_i \leq W_{max}$ && $x_i v_i \geq V_t$
3: **if** true **then**
4: **return** 1
5: **else**
6: **return** 0
7: **end if**

ii) Select a known NP-complete problem Π'

Let Π' = PARTITION

iii) Construct a transformation $f : \Pi' \to \Pi$

f: Consider an instance of the PARTITION problem $Z = \{z_1, z_2, \dots z_n\}$. Let

$$w_i = z_i \text{ and } v_i = z_i \qquad \forall\, 1 \leq i \leq n$$

$$W_{max} = V_t = M = \frac{1}{2}\sum_{i=1}^{n} z_i$$

If X is a 'yes' instance for PARTITION then ∃ Z' and Z-Z' such that

$$\sum_{i \in Z'} z_i = \sum_{i \in Z - Z'} z_i = \frac{1}{2}\sum_{i=1}^{n} z_i$$

To prove correctness, we need to show that 'yes' instances have a one-to-one mapping between the two problems.

Proof. If we let the knapsack contain items from Z' then it follows that

$$\sum_{i \in Z'} z_i = \sum_{i \in Z'} w_i = W_{max} \text{ and } \sum_{i \in Z'} z_i = \sum_{i \in Z'} v_i = V_t$$

$\therefore X \in Y$ for PARTITION $\Rightarrow f(X) \in Y$ for DK.

Likewise, if $f(X) \in Y$ for DK with the chosen set Z', then

$$\sum_{i \in Z'} w_i = \sum_{i \in Z'} z_i \leq W_{max} = \frac{1}{2}\sum_{i=1}^{n} z_i \text{ and } \sum_{i \in Z'} v_i = \sum_{i \in Z'} z_i \geq V_t = \frac{1}{2}\sum_{i=1}^{n} z_i$$

which implies that

$$\sum_{i \in Z'} z_i = \frac{1}{2}\sum_{i=1}^{n} z_i \quad \text{and} \quad \sum_{i \in Z - Z'} z_i = \sum_{i=1}^{n} z_i - \frac{1}{2}\sum_{i=1}^{n} z_i = \frac{1}{2}\sum_{i=1}^{n} z_i$$

$\therefore f(X) \in Y$ for DK $\Rightarrow X \in Y$ for PARTITION. □

iv) Prove that f is a polynomial time transformation

The transformation from PARTITION to DK consists only of assignment statements and so it is clear that the process takes polynomial time in the size of the input.

Since conditions (i-iv) are satisfied, we can say that DK is NP-complete.

(c)

To solve the 0/1 KNAPSACK problem requires a check of all possible solutions against each other. That is to say that each instance of DK that gives a 'yes' answer must be compared with every other solution before we can say that we have the optimal profit. Therefore solving for the optimal solution must be at least as hard as a single instance of DK. Even if we restrict V to a specific value, we find solving 0/1 KNAPSACK requires us to solve a sequence of decision problems (one for each configuration X) until a 'yes' answer is produced. This means that 0/1 KNAPSACK is *at least* as hard as DK, which we know to be NP-complete. Therefore we can assume that 0/1 KNAPSACK must reside in NP-hard.

Problem 6

Optimization PS(3) Problem: Given a set of n program and three storage devices. Let s_i be the amount of storage needed to store the i^{th} program. Let L be the storage capacity of each disk. Determine the maximum number of these n programs that can be stores on the three disks (without splitting a program over the disks).

Use the Approximation PS Algorithm given in the class for the PS(3) problem given above. Show that the following is true.

Let the approximation PS algorithm returns a number C, and let C^* be the optimal (maximum) number of programs that can be stores on the three disks.
(a) Show that the above approximation PS algorithm gives the performance ratio of

$$C^* \leq (C+2) \quad \text{OR} \quad \frac{C^*}{C} \leq 1 + \frac{2}{C}$$

(b) Give an example that achieves the performance ratio of $C^* = (C+2)$.

Solution:

Algorithm 7 PStore(s, n, L)

```
1: // Assume that s[i] ≤ s[i+1]                    ▷ non-decreasing order
2: i = 0; c = 0                                    ▷ c: count the number of stored programs
3: for j = 1 to 2 do
4:     sum = 0
5:     while sum + s[i] ≤ L do
6:         store i^th program into j^th device
7:         sum += s[i]
8:         i++; c++
9:         if i >n then
10:            return
11:        end if
12:    end while
13: end for
```

(a)

Let C^* be the maximum number of programs that can be stored on *two disks each of length* L and let C be the number of programs stored using PStore. Assume that k programs are stored when PStore is used so that $C = k$. If we consider the case of storing programs in non-decreasing order on a *single disk of length* $2L$, then the number of programs stored will be α. We can say that

$$\alpha \geq C^* \quad \text{and} \quad \sum_{1}^{\alpha} s[i] \leq 2L$$

If we let j be the largest index such that

$$\sum_{1}^{j} s[i] \leq L \quad \text{and} \quad \sum_{1}^{j+1} s[i] > L$$

then we can say $j \leq \alpha$ and that PStore assigns the first j programs to disk 1. We can also say that

$$\sum_{i=j+1}^{\alpha-1} s[i] \leq \sum_{i=j+2}^{\alpha} s[i] \leq L$$

Hence, PStore will assign at least the first $j+1, j+2, \ldots \alpha - 1$ programs to disk 2. This relation holds for any number of programs stored on any number of disks. If we consider an arbitrary number of disks k and load them with programs in a non-decreasing order, then eventually the programs still be stored on the last $k^{th} - 1$ and k^{th} disk. We will find that the first batch of j' programs will be stored on the $k^{th} - 1$ disk, and that *at least* the last $j'+1, j'+2, \ldots \alpha' - 1$ will be stored on the k^{th} disk. So for each *pair* of disks, space for one program is lost. Therefore,

$$C^* \leq C + k - 1$$

For the case of $k = 3$ disks, space for one program is lost between disk 1 and 2, and another space is lost between disk 2 and 3, and the approximation algorithm PStore obtains a performance ratio of

$$\frac{C^*}{C} \leq 1 + \frac{2}{C}$$

(b)

Let $L = 8$, $n = 9$, and $s[1], s[2], \ldots, s[n] = \{2, 2, 2, 3, 3, 3, 3, 4, 4\}$
If we use one disk of size 24L, then all programs will fit, however using 3 disks of size 8 each, there will be no room for s[8] and s[9] under the PStore routine.