CS-5300-01

Date: 10/19/2023

Advanced Algorithm Design and Analysis

① @ The largest integer of all 2n combined elements.

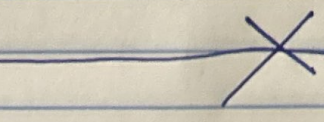
→ Best Case

As the list are already sorted in nondecreasing order or ascending order, the largest element will be at the end of the two lists. So for finding it we need to compare last element of both the list and pick the bigger one. The time complexity would be-

Time Complexity = O(1) [constant time], since it requires only constant no of steps which do not depend on the size of input lists Another reason, is also because we don't need to traverse all element to check as list is already sorted in ~~oo~~ ascending order.

→ Worst Case

The time complexity would be O(n) as we would need to go through both the list until getting the last element



(b) The second largest integer of all 2n combined elements.

→ Best Case

To find the second largest integer of all 2n combined elements in the 2 sorted list, we will compare the last elements of both list.

Steps

① Initialize two variables, 'i = n-1' 'j = n-1'. They both point to last element of X, Y list

② If X[i] > Y[j], then largest element is X[i]. so to get second biggest, we compare X[i-1] and Y[j]

③ Same step but we do check if Y[j] > X[i], then to get 2nd largest, we compare X[i] and Y[j-1]

Then we get the 2nd largest element. The Time complexity is O(1)

→ Worst Case

- This would happen if the second largest element is at the front of either X or Y list. Then, we would have to iterate through both list until we get second largest element.

- The Time Complexity = O(n)

(3) (c) the Median of all 2n combined elements

For Instance,

X = (4, 7, 8, 9, 12) and Y = (1, 2, 5, 7, 10), then median = 7, the n^{th} smallest, in the combined list (1, 2, 4, 5, 7, 8, 9, 9, 10, 12)

Solution:

Best Case

The best case time complexity here would be $O(\log(\min(n, m)))$ time, where n and m are the length of two lists

Steps: ① Finding the middle element of smaller list. Then we calculate the corresponding index in second list '$(n+m)/2 - i$'

Both these steps take constant time, $O(1)$

② In each iteration, we will compare elements at 'i' and '$(n+m)/2 - i$'

③ Then we repeat the above step until we get the median position. The size of search gets reduced by half [Binary Search]

So, no of steps required to get the median is logarithmic with respect to size of search so the binary search step has a

Time complexity of $O(\log(\min(n, m)))$.

P. T. O

1) Worst Case [Assuming when list are not sorted]

The Time complexity for worst case when using binary search is $O(\log n)$.

[n is size of list]

2) Worst Case [when list is sorted]

Time complexity: $O(\log(\min(n, m))$

It is same as best case. The reason being we are doing binary search on the smaller of two list. In each iteration, the size of search gets cut by half, leading to a logarithm time complexity.

(2) (a) Dynamic programming functional equation:

- ~~The~~ Let DP[i][j] is defined to check if there is a subset of the first 'i' elements of Z' in way that sum of numbers in Z is equal to J
- Framing the equation, dp(i,j) = dp(i-1,j) or dp(i-1)(j-zi), telling that either the current element is excluded or included in list.
- Base cases are: dp[0][0] = True
  dp[0][j] = False for j>0
  dp(i,0) = True for all i

(b) The Algorithm implementing the functional equation is as follows:

- (1) Calculate total sum in Z
- (2) Checking if total sum is ~~odd~~ even: if it is odd, the 1-2 Partition is not possible. RETURN FALSE.
- (3) Intialize the DP array. An empty subset has a sum of 0. So DP[i][0] true for all i
- (4) Filling up array: For each element Z[i] in set Z and for j:
  DP[i][j] is TRUE if:
  DP[i-1][j] is TRUE
- (5) Return result - DP[N-1][TOTAL_SUM/2]. If True, it tells there exists a subset sum equal to S/2.

(C) Let $Z = \{2, 5, 4, 3, 7\}$ with a sum of 21 as sum is 21 which is an odd number so there is no solution for this.

$2+5+4+3+7 = 21 = \text{Odd} \neq \text{No Solution}$

(D) Time Complexity

The Time Complexity is: $O(n * s)$ where

→ n is the numbers present in set Z
→ S is the total sum

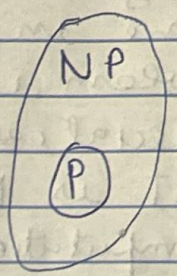
Space Complexity

The Space Complexity is: $O(n * s)$
Reason is the array of size $(n+1) * (s/2 + 1)$

③

ⓐ Ans: False

The above statement "If P ≠ NP, then no problem in NP can be solved in polynomial time deterministically" is not always true.
The relationship between P and NP remains ambiguous and even if there are NP problems that can be solved in polynomial time. Thus, even if P ≠ NP, there are still problems within NP that can be solved in polynomial time deterministically



Hence, the statement is <u>false</u>

ⓑ <u>Ans</u>: True

A problem is said to be NP-complete if it is in NP and if any problem in NP can be shortened to in polynomial time. This means that if we have a polynomial time algorithm for an NP complete problem we can use it to solve any problem in NP in polynomial time.
<u>Ex</u>: Suppose A is NP-complete problem and B is a problem that can be reduced to A in polynomial time. This indicates that a case of B can be changed into an case of A using a polynomial-time technique. The case of B can be solved if the case of A can be resolved

Since A is NP-complete, We know that there is a polynomial-time algorithm for A. So, we can use this algorithm to solve the case of B. This also means that B can be solved in polynomial time. SINCE B is in NP, it proves that B is NP complete

(C) Answer: False

"3SAT (all clauses have size 3) is NP-complete. 1SAT (all clauses have size 1) is also NP-complete" is False

3SAT is indeed NP-complete because it's hard to find solution, but easy to check them. On the ~~Bo~~ other hand, 1SAT is not NP-complete as it can be solved easily and quickly just by looking for any mismatch clauses. The 1SAT can be done in polynomial time But the 3SAT problem is NP-complete Also the 3SAT is a special case of SAT problem. In summary, 3SAT is NP-complete because it is computationally hard to find a solution but easy to verify one, while 1SAT can be solved easily and thus is not NP-complete

So the above statement is False

(4)

(a) A nondeterministic polynomial time algorithm for the SUM of subsets is done using the steps below:

→ for each element in let set A (suppose) non-deterministically choose if to include it in subset or not

→ If the sum of the elements which are selected equals to m, then we can accept or reject. It works recursively

→ For all possible subsets of input set A the algorithm works recursively. If the sum equal to target, we can accept or return True, else reject it or FALSE

→ This runs in polynomial time as the number of choices we have are finite and recursion depth is bounded by size of input set A.

(b)

The transformation from the partition problem to the sum of subsets problem is defined as -

→ We have a set of positive integers -

$Z = \{z_1, z_2, z_3, z_4, \cdots, z_n\}$

Now,

$Sum[Z'] = Sum[Z - Z']$, let us assume $sum\{A\} = S$

defining a new set with an additional element

$\{A'\} = \{A\} \cup \{2M - S\}$

$2N_r = \text{sum}\{\bar{A}\}$

$S(\bar{A}) = \{A'\}\{A - A'\}$ as output

We return which has $\{2M - S\}$ and a valid solution to subset set sum problem with sum M.

(c) If there exists a subset Zi of Z such that the sum of its element is equal to the sum of the remaining element in Z So if partition problem is having a ~~problem~~ solution then the subset sum also has the solution. It can be easily proved. We use the solution of partition problem — the above solution with the input sequence of the partition problem and sum = M/2.
The sum of Z would be half of M, which is the sum of the remaining element in Z. So there is 1:1 relationship between the Partition problem and solution of sum of subset problem. Also Partition problem is NP Complete which tells that SUM of subset is also NP-Complete

(5)

(a) The decision version of the 0/1 knapsack problem (DK) can be expressed as -

given a set of items, each having some weight and value, determine if there is a subset of items with a total weight less than or equal to the limit and the total value greater than or equal to a given threshold

n → items
w_i → weight
v_i → value
W → weight limit
V → Threshold Value

$\sum_{i \in S} w_i \leq W$ → Total Weight

$\sum_{i \in S} v_i \geq V$ for some subset S

(b) To prove DK is NP-Complete, DK is in NP and DK is NP-hard.

→ For DK to be in NP, the DK problem just needs a easy way to check if given answer is right. The solution is a list of chosen items called a subset S. We can easily add up the weights and values of items in S and see if it is within the limit. The total weight is less than or equal to W and total value ≥ V.

→ To show DK is NP-hard, we can do reduction from the partition problem which is a well known NP-complete problem. A given set B of integers can be partitioned into two subsets such that sum of numbers in each subset is equal.

Hence DK is NP-complete by satisfying both conditions.

© By proving the decision version DK of the 0/1 KNAPSACK problem as NP-complete is enough to establish NP-Hardness for the entire optimization problem. An NP-complete problem is one that is both in NP and as hard as any problem in NP. If a problem is NP-Complete, it is also NP-hard by definition. If a polynomial-time algorithm exists for the optimization problem, it would contradict the NP completeness of DK as solving it would indirectly solve the decision problem. Thus if we ~~solve~~ could solve 0/1 Knapsack problem in polynomial time, we could all instances of DK which is not possible unless P = NP. So just showing that DK is NP complete is enough to prove the 0/1 Knapsack problem is NP hard.

⑥ (a) Let C^* be the optimal number of programs that can be stored.

Also, let C be the number of programs the approximation algorithm pack

→ We have to show that performance ratio of

$C^* \leq (C+2)$ OR $C^*/C \leq 1+2/C$

$\Rightarrow C^*/C \leq 1+2/C$

$C^* \leq C+2$

In each step, the next problem is placed on the disk that has minimum load

So the load on each disk is at max twice the average load

→ Let OPT be the average optimal load on the disk

Total storage: OPT * C^*

C^* is the optimal number of programs which can be stored with optimal average load.

Now, we have: OPT x C^* <= 2 x OPT x C — ①

Now dividing both sides of ①

$$\frac{OPT \times C^*}{OPT \times C} <= \frac{2 \times OPT \times C}{OPT \times C}$$

$$\boxed{C^*/C <= 2}$$

$\Rightarrow C^*/C <= 2$

$\Rightarrow$ Adding 2 to both sides

$C^* <= 2C$

$C^*+2 <= 2C+2$

$C^*+2 <= 2(C+2)$

$\therefore$ $\boxed{C^* <= (C+2)}$

(6) (b)

$\boxed{C^* = C + 2}$ → Performance Ratio

We will construct 3 storage devices with capacity L each and set of programs with varying s_i. We have to leave 1 unit of storage space

Example

Let $L = 9$

s_i are as follows:

$s_1 = s_2 = s_3 = 8$

$s_4 = s_5 = s_6 = 9$

$s_7 = s_8 = 1$

→ Using the approximation algorithm, we allocate s_1, s_2, s_3 to 3 devices, each having 8 units and leaving 1 unit on each disk. Thus, $C = 3$

$s_1 = 8, s_2 = 8, s_3 = 8 \Rightarrow C = 3$

→ Optimal solution stores s_4, s_5, s_6, s_7, s_8 fully making the space and $c^* = 5$

So Now $c^* = 5, C = 3 \Rightarrow c^* = C + 2$ [Not Matching $c^* = C + 2$]

To fullfil, we add s_7, s_8. On adding, the optimal solution yeilds $c^* = 5$, while the approximation algorithm gives $C = 3$

AA $\Rightarrow C^* = C + 2 \Rightarrow c^* = 3 + 2 = 5$

$\boxed{5 = 5}$ Hence Proved