4.

c)

```
void addBook(string title, string publisher)
{
    dllNode* p = dll.head;
    dllNode* q = new dllNode(title, publisher, NULL, NULL);
    while(p->next!= NULL && p->next->publisher != publisher)
    {
        p = p ->next;
    }
    q->prev = p;
    q->next = p->next;
    if(p->next != NULL){
        p->next->prev = q;
        p->next = q;
    }
}
```

5. a)

5(a)

Greedy:

itemNo	w	v	v/w
1	3	15	5
2	2	20	10
3	10	30	3
4	2	14	7

After sorting: 2, 4, 1, 3

item	x_i	$x_i w_i$
2	1	20
4	1	14
1	0	0
3	0	0

$\therefore$ 20 + 14 = 34

DP:

item No / capacity	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	15	15	15	15
2	0	0	20	20	20	35	35
3	0	0	20	20	20	35	35
4	0	0	14	14	34	35	35

$\therefore$ The items taken: 1 and 2

w : 3 + 2 = 5 < 6

v : 15 + 20 = 35

7.a)

```
Initially let R be the set of all requests, and let A be empty
While R is not yet empty
  Choose a request i ∈ R that has the smallest finishing time
  Add request i to A
  Delete all requests from R that are not compatible with request i
EndWhile
Return the set A as the set of accepted requests
```

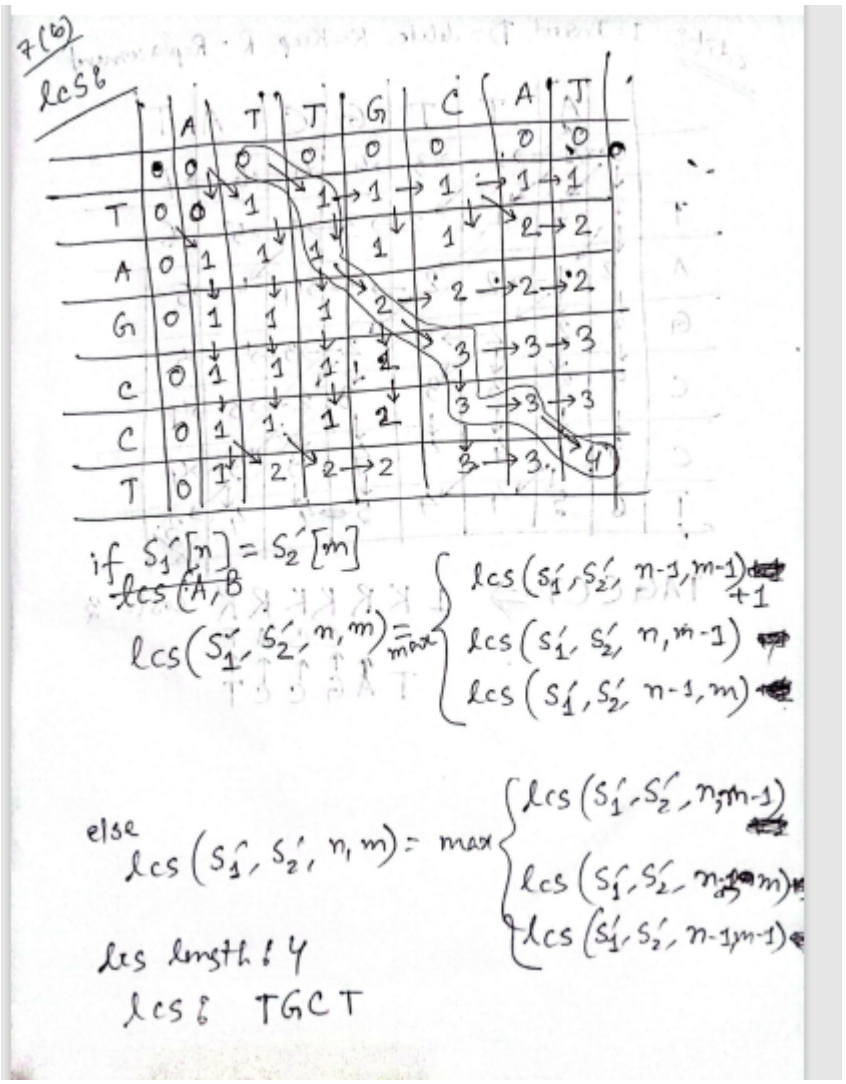
(4.3) *The greedy algorithm returns an optimal set A.*

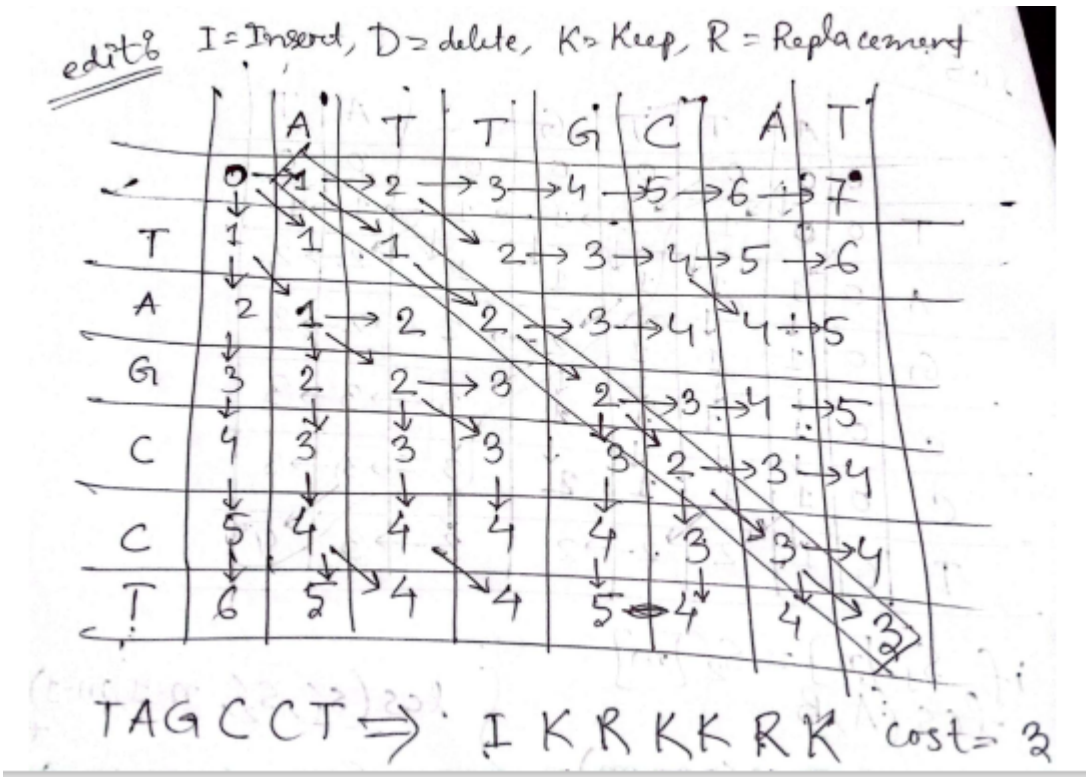
Proof. We will prove the statement by contradiction. If A is not optimal, then an optimal set $\mathcal{O}$ must have more requests, that is, we must have $m > k$. Applying (4.2) with $r = k$, we get that $f(i_k) \leq f(j_k)$. Since $m > k$, there is a request j_{k+1} in $\mathcal{O}$. This request starts after request j_k ends, and hence after i_k ends. So after deleting all requests that are not compatible with requests $i_1, \ldots, i_k$, the set of possible requests R still contains j_{k+1}. But the greedy algorithm stops with request i_k, and it is only supposed to stop when R is empty—a contradiction. ■

Implementation and Running Time We can make our algorithm run in time $O(n \log n)$ as follows. We begin by sorting the n requests in order of finishing time and labeling them in this order; that is, we will assume that $f(i) \leq f(j)$ when $i < j$. This takes time $O(n \log n)$. In an additional $O(n)$ time, we construct an array $S[1 \ldots n]$ with the property that $S[i]$ contains the value $s(i)$.

We now select requests by processing the intervals in order of increasing $f(i)$. We always select the first interval; we then iterate through the intervals in order until reaching the first interval j for which $s(j) \geq f(1)$; we then select this one as well. More generally, if the most recent interval we've selected ends at time f, we continue iterating through subsequent intervals until we reach the first j for which $s(j) \geq f$. In this way, we implement the greedy algorithm analyzed above in one pass through the intervals, spending constant time per interval. Thus this part of the algorithm takes time $O(n)$.

7.b)


7(b)
lcs &
if $S_1'[n] = S_2'[m]$
$lcs(S_1', S_2', n, m) = \max \begin{cases} lcs(S_1', S_2', n-1, m-1) + 1 \\ lcs(S_1', S_2', n, m-1) \\ lcs(S_1', S_2', n-1, m) \end{cases}$
else
$lcs(S_1', S_2', n, m) = \max \begin{cases} lcs(S_1', S_2', n, m-1) \\ lcs(S_1', S_2', n-1, m) \\ lcs(S_1', S_2', n-1, m-1) \end{cases}$
lcs length : 4
lcs : TGCT



8.a) i) + ii)

```
Algorithm MinDist(P)
   if |P| ≤ 3 then
      try all pairs
   else
      find x median of the points
      split P into L and R of about the same size using x

      δ = min {MinDist(L), MinDist(R)}
      B_L = points in L within δ of x median
      B_R = points in R within δ of x median

      sort B_R by y-coordinates
      for p ∈ B_L do
         binary search for highest q ∈ B_R in a δ × 2δ box of p
         compare at most 6 points starting with q to p
```

The algorithm is dominated by sorting and recursing, so the run-time recurrence is $T(n) = 2T(n/2) + \Theta(n \log n)$. This is case 2 of the Master Theorem in the Goodrich and Tamassia textbook. Hence, the run-time is $\Theta(n \log^2 n)$. We note that the algorithm in CLRS does some clever bookkeeping to cut this down to $\Theta(n \log n)$.

b)

```
Merge-and-Count(A,B)
  Maintain a Current pointer into each list, initialized to
    point to the front elements
  Maintain a variable Count for the number of inversions,
    initialized to 0
  While both lists are nonempty:
    Let a_i and b_j be the elements pointed to by the Current pointer
    Append the smaller of these two to the output list
    If b_j is the smaller element then
      Increment Count by the number of elements remaining in A
    Endif
    Advance the Current pointer in the list from which the
      smaller element was selected.
  EndWhile
```

```
Once one list is empty, append the remainder of the other list
    to the output
Return Count and the merged list
```

```
Sort-and-Count(L)
  If the list has one element then
      there are no inversions
  Else
      Divide the list into two halves:
         A contains the first ⌈n/2⌉ elements
         B contains the remaining ⌊n/2⌋ elements
      (r_A, A) = Sort-and-Count(A)
      (r_B, B) = Sort-and-Count(B)
      (r, L) = Merge-and-Count(A, B)
   Endif
   Return r = r_A + r_B + r, and the sorted list L
```

Since our `Merge-and-Count` procedure takes $O(n)$ time, the running time $T(n)$ of the full `Sort-and-Count` procedure satisfies the recurrence (5.1). By (5.2), we have

(5.7) *The* `Sort-and-Count` *algorithm correctly sorts the input list and counts the number of inversions; it runs in $O(n \log n)$ time for a list with n elements.*

PSEUDOCODE:

```
Merge-And-Count(A, B):
current1 = 1
current2 = 1
count = 0
L = empty list
index = 0
while current1 <= A.length and current2 <= B.length
    if A[current1] > 2 x B[current2]
        index = index + 1
        L[index] = B[current2]
        count = count + A.length - current1 + 1
        current2 = current2 + 1
    else
        index = index + 1
        L[index] = A[current1]
        current1 = current1 + 1
while current1 <= A.length
    index = index + 1
    L[index] = A[current1]
while current2 <= B.length
    index = index + 1
    L[index] = B[current2]
return count and L
```

(The Sort-And-Count is rather easy, same as the algorithm part, just use equal sign instead of description :"))

8.c)
Yes, adopting memoization over dynamic programming can result in faster solution.
Example, in finding the nth fibonacci number the recursion is as follows:

$$F(n) = F(n-1) + F(n-2), \text{ for } n >= 2$$
$$F(0) = F(1) = 1$$

Here, to calculate each term, we have to call two recursive call for previous two terms.
So, in the recursion tree there is about exponential number of node, i.e. recursive calls which produces a exponential time complexity (very slow).

But if we observe carefully that, there is some unnecessary calls which are already calculated such as F(5) = F(4) + F(3) and F(4) = F(3) + F(2), In this example, F(3) is called twice which is unnecessary. So, if we already have known the value of F(3) by the first call from F(4), we can use this result in constant time saving time by maintaining a memo for previous result. This method is called memoization. **So… yeah, it's good to memoize**!!