22nd SCSE – Past Year Paper Solution (2020 – 2021 Semester 2)
CE/CZ 2101 – Algorithm Design and Analysis

1 (a) Assume the array is 0-th indexed.

Mergesort(0,5) [5 ,2 ,8, 5 ,3 ,4]
Mergesort(0,2) [5 ,2, 8] Mergesort(3,5) [5,3,4]
Mergesort(0,1) [5,2] Mergesort(2,2) [8] Mergesort(3,4) [5,3] Mergesort(4,5) [4]

Merge(0,1) is called:

Initial Array	Initial MergeList	Final Array	Final MergeList	# of Comparison
[5] [2]	[ ]	[] [2]	[2]	1
[] [2]	[2]	[] []	[2, 5]	0 (1st half is empty)

Merge(2,2) is call

Initial Array	Initial MergeList	Final Array	Final MergeList	# of Comparison
[8] []	[ ]	[] []	[8]	0 (2nd half is empty)

Merge(3,4) is call

Initial Array	Initial MergeList	Final Array	Final MergeList	# of Comparison
[5] [3]	[ ]	[5] []	[3]	1
[5] []	[3]	[] []	[3, 5]	0 (2nd half is empty)

Merge(4,5) is call

Initial Array	Initial MergeList	Final Array	Final MergeList	# of Comparison
[4] []	[ ]	[] []	[4]	0 (2nd half is empty)

After all the merge is called
Return to mergesort(0,2), merge(0,2) is called

Initial Array	Initial MergeList	Final Array	Final MergeList	# of Comparison
[2,5] [8]	[ ]	[5] [8]	[2]	1 (2<5)
[5] [8]	[2]	[] [8]	[2, 5]	1 (5<8)
[] [8]	[2, 5]	[] []	[2, 5,8]	0 (1st half is empty)

Return to mergesort(3,5), merge(3,5) is called

Initial Array	Initial MergeList	Final Array	Final MergeList	# of Comparison
[3,5] [4]	[ ]	[5] [4]	[3]	1 (3<4)
[5] [4]	[3]	[5] []	[3, 4]	1 (5>4)
[5] []	[3, 4]	[] []	[3, 4 ,5]	0 (2nd half is empty)

After all the merge is called
Return to mergesort(0,5), merge(0,5) is called

Initial Array	Initial MergeList	Final Array	Final MergeList	# of Comparison	# of key shift
[2, 5,8] [3, 4 ,5]	[ ]	[5,8] [3,4,5]	[2]	1 (2<3)	1
[5,8] [3,4,5]	[2]	[5,8] [4,5]	[2,3]	1 (5>3)	1
[5,8] [4,5]	[2,3]	[5,8] [5]	[2,3,4]	1 (5>4)	1

[5,8] [5]	[2,3,4]	[8] []	[2,3,4,5,5]	1 (5==5)	2 (Both array are shift)
[8] []	[2,3,4,5,5]	[] []	[2,3,4,5,5,8]	0 (2nd half is empty)	0 (This is block shift)

In Total there is 10 key comparison and 5 key shift

(b) The worst case scenario for insertion Sort is when the array is reversely sorted (16,15,…,3,2,1).
In this case, the number of key comparisons is $1 + 2 + 3 + 4 + 5 + \ldots + (n-1) = \frac{n(n-1)}{2}$, $n\ is\ size\ of\ subarray$

The worst-case scenario for Quicksort is when every pivot that is chosen is the smallest or the largest element in the array which causes array is splited into size 0 and n-1 (exclude pivot).
The number of key comparisons is $(n-1) + (n-2) + \ldots + 1 = \frac{n(n\text{-}1)}{2}$, $n\ is\ size\ of\ subarray$

When the size of subarray is less than or equal to 16, Quick Sort will be changed to Insertion Sort is called.
Therefore, the number of key comparison for $n \geq 16$ is equal to
number of key comparisons by Quicksort on size n
– number of key comparisons by Quicksort on size 16
+ number of key comparisons by Insertion Sort

$$= \frac{n(n-1)}{2} - \frac{16(16-1)}{2} + \frac{16(16-1)}{2}$$
$$= \frac{n(n-1)}{2}$$

(c) Suppose the array is 0-indexed.
Denote Maj(A) as majority element of array A.
Note that Maj(A) may be NIL
Denote Count(number, A) as the number of occurrences of the number appear in array A

A divide and conquer algorithm would be divide an array A of size n by half into two subarray denote A1 and A2 with size n/2.
Now suppose we already have the Majority element for subarray A1, denote Maj(A1) and Majority element for subarray A2, denote Maj(A2), now we want to find the Majority element for the original array A.

Case 1:
If Maj(A1) = Maj(A2), then Maj(A) = Maj(A1)
In other word, if majority element of both subarray is the same then the majority element of subarray is the majority element of original array even if Maj(A1) and Maj(A2) is NIL. (Prove see below)

Case 2:
If Maj(A1) ≠ Maj(A2), then Maj(A1), Maj(A2) and NIL are the only candidates Majority element for Maj(A)
In other word, if majority element of both subarray are not the same then the only possible majority element of the original array is either Maj(A1) or Maj(A2). (Prove see below)

At such point,
If Count(Maj(A1), A) > n/2, then Maj(A) = Maj(A1)
else if Count (Maj(A2), A) >n/2, then Maj(A) = Maj(A2)
else Maj(A) = NIL
In other word, simply scan through the original array and check the number of occurrences of Maj(A1) and Maj(A2) in the array A. If the number of occurrences exceeds half of the array then the majority element is it. Simply ignore Maj(A1) if it is NIL, this applies to Maj(A2).

At this moment we know how to find Maj(A) from its subarray. So we divide the array into half until we reach the base case (array size of 1). The majority element of the array of size 1 is simply the only element in the array.

The pseudo-code for the algorithm is as such

```
int Maj (int A[], int left, int right)
      // Can be NIL;
      // This denote the subarray of A from index left to index right
      int n = right-left+1;
      if(left == right)
            // subarray of size 1
            Return A[left];

      int mid = (left+right)/2;
      int maj_A1 = Maj(A,left,mid);
      int maj_A2 = Maj(A,mid+1,right);

      if(maj_A1 == maj_A2) return maj_A1;

      if(maj_A1 != NIL && Count(maj_A1, A, left,right) > n/2) return maj_A1;
      elif(maj_A2 != NIL && Count(maj_A2, A,left,right)> n/2) return maj_A2;
      else return NIL;
```

```
int Count (int maj_A1, int A[], int left, int right)
      int cnt = 0;
      for(int i=left;i<=right;i++)
            if(maj_A1 == A[i]) cnt++;
      return cnt;
```

Complexity Analysis

We can build recurrences formula based on our algorithm.
First we divide our array into half and conquer both subarray. After we conquer both subarray (subproblems), we solve the problem by at most calling two Count (number, A). Count(number,A) is a linear scan therefore its equality Test is $O(N)$. Therefore with such, we come out with following recurrences formula.

$$W(N) = 2W(N/2) + 2N$$
$$By\ using\ master\ method,$$
$$a = 2, b = 2, n^{\log_a b} = n$$
$$Since\ f(n) = 2n = \theta(n) = \theta\left(n^{\log_a b}\right)$$
$$By\ using\ the\ second\ case,$$
$$W(N) = \theta(N\ lg\ N)$$

Editor's Note: In this solution, I do complexity analysis with master method, however you may use iterative method or substitution method as well. Some tips to thinking this solution in exam, since you know the solution you want to have is N lg N and you have to use divide and conquer. By the master method you can see that after each divide and conquer, you can only do linear scan to combine the result of both subproblems. With this, the key point is the observe that majority element of both subproblem is the only candidates answer to the current problem. Then slowly figure the answer from this.

Next I will be giving the prove that why my statement given in Case 1 and case 2 is correct.

Prove for Case 1:

Suppose that Maj(A1) and Maj(A2) is NIL. The largest number of count of a number occur in the array A is at most N/2 which is not the majority element.
Suppose that the count of a number (let say x) occur in array A is N/2 +1. Then x can occur in A1 (N/2)/2 times (since Maj(A1) is Nil). With such, the count of x in A2 is N/2 +1 – N/4 = N/2+1. However Maj(A2) is Nil. This lead to a contradiction.

Suppose that Maj(A1) and Maj(A2) is not NIL. The count of Maj(A1) in A1 is at least N/4 +1 and the count of Maj(A2) in A2 is also N/4 +1. Therefore the count of Maj(A1) in A is N/4+1 + N/4 +1 = N/2+2 (>N/2).

Prove for Case 2:

This is to prove that only Maj(A1) and Maj(A2) can be candidates for Maj(A) only.
Suppose an other number (let say x) that is not Maj(A1) or Maj(A2) is Maj(A). Then it would have to occur in array A at least N/2 +1. Suppose in Maj(A1) is NIL, x occur in A1 N/4 times. If so, x will occur in A2 N/2 + 1 – N/4 = N/2 +1. However x is not Maj(A2). This lead to a contradiction.

2 (a) Let initialize 4 data structures based on the lecture note.
Array d: distance of a fringe vertex from the tree
Array pi: vertex connecting a fringe vertex to a tree vertex
Array S: whether a vertex is in the minimum spanning tree being built
Priority queue pq: queue of fringe vertices in the order of the distances from the tree

Initial State:

Array d:

A	B	C	D	E	F	G
∞	∞	∞	∞	∞	∞	∞

Array pi:

A	B	C	D	E	F	G
na	na	na	na	na	na	na

Array S

A	B	C	D	E	F	G
0	0	0	0	0	0	0

Priority queue pq (dist, vertex) : {Empty}

After 1st Iteration
Start from vertex B, Select it as tree vertex and push all the fringe vertex to the queue

Array d:

A	B	C	D	E	F	G
7	0	8	9	7	∞	∞

Array pi:

A	B	C	D	E	F	G
B	na	B	B	B	na	na

Array S

A	B	C	D	E	F	G
0	1	0	0	0	0	0

Priority queue pq (dist, vertex) : { (7,A), (7,E), (8,C), (9,D) }
Note: A,C,D,E are the Fringe vertex that is being checked in this iteration

After 2nd Iteration
Vertex A is chosen (Smallest dist and vertex A and E tie is break by alphabetical order.)

Array d:

A	B	C	D	E	F	G
7	0	8	5	7	∞	∞

Array pi:

A	B	C	D	E	F	G
B	na	B	A	B	na	na

Array S

A	B	C	D	E	F	G
1	1	0	0	0	0	0

Priority queue pq (dist, vertex) : { (5,D) , (7,E), (8,C), }
Note: D are the Fringe vertex that is being checked in this iteration

After 3rd Iteration
Vertex D is chosen (Smallest dist in pq)

Array d:

A	B	C	D	E	F	G
7	0	8	5	7	6	∞

Array pi:

A	B	C	D	E	F	G
B	na	B	A	B	D	na

Array S

A	B	C	D	E	F	G
1	1	0	1	0	0	0

Priority queue pq (dist, vertex) : { (6,F) ,(7,E), (8,C), }
Note: E and F are the Fringe vertex that is being checked in this iteration but E is not update because weight DE (15) > current dist[E] (7)

After 4th Iteration
Vertex F is chosen (Smallest dist in pq)

Array d:

A	B	C	D	E	F	G
7	0	8	5	7	6	11

Array pi:

A	B	C	D	E	F	G
B	na	B	A	B	D	F

Array S

A	B	C	D	E	F	G
1	1	0	1	0	1	0

Priority queue pq (dist, vertex) : { (7,E), (8,C),(11,G) }
Note: E and G are the Fringe vertex that is being checked in this iteration but E is not update because weight FE (8) > current dist[E] (7)

After 5th Iteration
Vertex F is chosen (Smallest dist in pq)

Array d:

A	B	C	D	E	F	G
7	0	8	5	7	6	11

Array pi:

A	B	C	D	E	F	G
B	na	B	A	B	D	F

Array S

A	B	C	D	E	F	G
1	1	0	1	0	1	0

Priority queue pq (dist, vertex) : { (7,E), (8,C),(11,G) }
Note: E and G are the Fringe vertex that is being checked in this iteration but E is not update because weight FE (8) > current dist[E] (7)

After 6th Iteration
Vertex E is chosen (Smallest dist in pq)

Array d:

A	B	C	D	E	F	G
7	0	5	5	7	6	9

Array pi:

A	B	C	D	E	F	G
B	na	E	A	B	D	E

Array S

A	B	C	D	E	F	G
1	1	0	1	1	1	0

Priority queue pq (dist, vertex) : { (5,C),(9,G) }
Note: C and G are the Fringe vertex that is being checked in this iteration

After 7th Iteration
Vertex E is chosen (Smallest dist in pq)

Array d:

A	B	C	D	E	F	G
7	0	5	5	7	6	9

Array pi:

A	B	C	D	E	F	G
B	na	E	A	B	D	E

Array S

A	B	C	D	E	F	G
1	1	1	1	1	1	0

Priority queue pq (dist, vertex) : { (9,G) }
Note: No Fringe vertex that is being checked in this iteration

After 8th Iteration (Last Iteration)
Vertex G is chosen (Smallest dist in pq)

Array d:

A	B	C	D	E	F	G
7	0	5	5	7	6	9

Array pi:

A	B	C	D	E	F	G
B	na	E	A	B	D	E

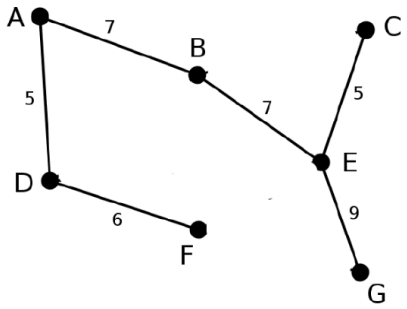
Array S

A	B	C	D	E	F	G
1	1	1	1	1	1	1

Priority queue pq (dist, vertex) : { Empty }
Note: No Fringe vertex that is being checked in this iteration

Based on the final array of d, pi, S,
We Stop the iteration because all the element in S is 1 meaning all vertex is tree vertex already.

The summation of the value in array d is the total weight of MST and
Array pi is the MST itself.



Total weight: 7+0+5+5+7+6+9 = 39

Editor's note: I explained using the changes in the data structures, alternately, you may explain it by drawing out graph after each iteration

(b) The answer is **YES**

I explain a few terminology first. W(P) denote total weight of the path P.

Suppose G is the weighted graph with non-negative edge weights before multiplying by 5
Path P: s -> … -> t is the shortest path from vertex s to vertex t. This implies that for any other possible path Q, $W(Q) >= W(P)$

Here I use Proof by contradiction,
Suppose a graph G' is the weighted graph after every edge is multiply by 5.
Let the Path P' as the same path as P: s-> …. -> t. Notice that W(P') = 5* W(P).

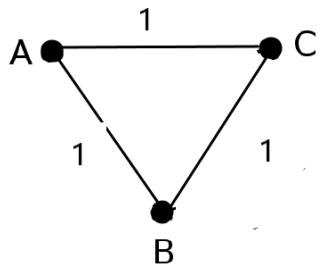
Assume that there is another path Q' such that W(Q') < W(P'), meaning there exists another path that is shorter than path P'

$$W(Q') < W(P')$$
$$5 \times W(Q) < 5 \times W(P)$$
$$W(Q) < W(P)$$

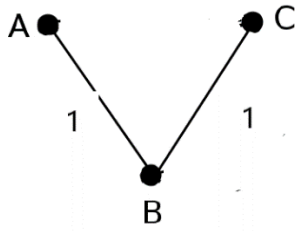
Here lead to a contradiction that it contradicts the fact path P is the shortest path of graph G, $W(Q) >= W(P)$.
Therefore the assumption is wrong, path P' is still the shortest path on the new graph

Editor's note: Here I use proof by contradiction. However you may think logically and say that the new graph with edge multiplied by a positive integer will causes all the weight of the path multiplied by same positive integer regardless of their number of edges. Since the same effect is applied to all the path, so there is no changes to the fact that path P is the shortest path.

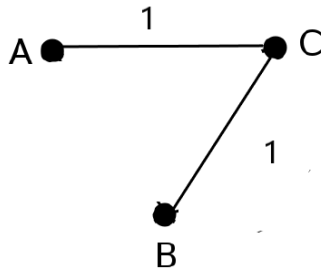
(c) The answer is **NO**
CounterExample,



Assume Prim's Algorithm is executed starting at vertex A and alphabetical order to break a tie in choosing the next vertex, this is the MST you will get



Assume Kruskal's Algorithm is executed and reverse alphabetical order of the vertex connected to to break a tie in choosing the next edge, This is the MST you will get



When there is no unique MST and when Prim and Kruskal algorithm treat the edges with same weight differently, then both algorithm may yield a difference MST. The answer of MST depends on the rule by which how the algorithms break ties.

3 (a) (i) $W(n) = W(n/2) + n\ lg\ n$

By Comparing with the master method formula, $W(n) = aW(n/b) + f(n)$

$a = 1$

$b = 2$

$f(n) = n\ lg\ n$

$n^{\log_b a} = n^{\log_2 1} = n^0 = 1$

We can assume $\varepsilon = 1$,

$f(n) = \Omega(n) = \Omega(n^{0+\varepsilon})$

and

$af(n/b) \leq cf(n)$

$\mathrm{f(n/2)} \leq cf(n)$

$\frac{n}{2}\lg\left(\frac{n}{2}\right) \leq \mathrm{c\ nlg(n)}$

$\frac{\left(lg(n) - lg(2)\right)}{2lg(n)} \leq \mathrm{c}$

$c \geq \frac{1}{2} - \frac{lg(2)}{2lg(n)}$

Since when n is sufficiently large, $\lim_{\mathrm{n}\to\infty} \frac{lg(2)}{2lg(n)} = 0$,

so we can let $\mathrm{c} = 3/4$ and so the inequality is valid

Since there exists a $\varepsilon, f(n) = \Omega(n^{0+\varepsilon})$

And a c such that $af(n/b) \leq cf(n)$.

By using third case of Master Method,

$\mathrm{W(n)} = \theta\big(f(n)\big) = \theta(n\ lg\ n)$

(ii) $W(n) = 2\ W(n/2) + n\ lg\ n$

By Comparing with the master method formula, $W(n) = aW(n/b) + f(n)$

$a = 2$

$b = 2$

$f(n) = n\ lg\ n$

$n^{\log_b a} = n^{\log_2 2} = n^1 = \mathrm{n}$

$f(n) = \theta(n\ log\ n) = \theta\big(n^{\log_b a}\ log\ n\big)$

Since $f(n) = \theta\big(n^{\log_b a}\ log\ n\big)$

By using second case of Master Method,

$\mathrm{W(n)} = \theta\big(n^{\log_b a}\ log^2\ n\big) = \theta(n\ log^2\ n)$

(b) (i) Assume the string is 1-indexed

M = 6

Pattern	P	A	P	A	Y	A
position	1	2	3	4	5	6

The position of rightmost P is 3, charJump ['P] = 6 – 3 = 3
The position of rightmost A is 4, charJump ['A'] = 6 – 6 = 0
The position of rightmost Y is 5, charJump ['Y'] = 6 – 5 = 1
charJump[Other Char] = m = 6

Slide Pattern		P	A	P	A	Y	A					
Pattern	P	A	P	A	Y	A						

Slide[6] = 1

Slide Pattern			P	A	P	A	Y	A				
Pattern	P	A	P	A	Y	A						

Slide[5] = 2

Slide Pattern							P	A	P	A	Y	A
Pattern	P	A	P	A	Y	A						

(No occurrence of matching suffix in the pattern)
Slide[4] = 6

Slide Pattern							P	A	P	A	Y	A
Pattern	P	A	P	A	Y	A						

(No occurrence of matching suffix in the pattern)
Slide[3] = 6

Slide Pattern							P	A	P	A	Y	A
Pattern	P	A	P	A	Y	A						

(No occurrence of matching suffix in the pattern)
Slide[2] = 6

Slide Pattern							P	A	P	A	Y	A
Pattern	P	A	P	A	Y	A						

(No occurrence of matching suffix in the pattern)
Slide[1] = 6

Matched	5	4	3	2	1	0
Slide	6	6	6	6	2	1
MatchJump	11	10	9	8	3	1
Pattern	P	A	P	A	Y	A

(ii) Recall that from the lecture notes
J is the index of the element on the text
K is the index of the element on the pattern.

Simple Boyer-Moore algorithm
When there is a mismatch, the following shift will occur.

```
j+=max(charJump[T[j]],m-k+1);
```

P5:													P	A	Y	A	Y	A
P4:											P	A	Y	A	Y	A		
P3:									P	A	Y	A	Y	A				
P2:						P	A	Y	A	Y	A							
P1:	P	A	P	A	Y	A												
T:	B	A	N	A	N	A		K	A	Y	A		P	A	P	A	Y	A
j	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Noted: The highlighted is the mismatch, PX meaning which times the pattern is checked again after shift.

Mismatch on P1 is found on 5th element of Pattern. So 2 character comparisons are done
charJump[T[5]] = charJump[N] = 6
m – k + 1 = 6 – 5 + 1 = 2
max(charJump[T[5]] , m-k+1) = max(6,2) = 6

Mismatch on P2 is found on 3rd element of Pattern. So 4 character comparisons are done
charJump[T[8]] = charJump[K] = 6
m – k + 1 = 6 – 3 + 1 = 4
max(charJump[T[8]] , m-k+1) = max(6,4) = 6

Mismatch on P3 is found on 5th element of Pattern. So 2 character comparisons are done
charJump[T[13]] = charJump[P] = 3
m – k + 1 = 6 – 5 + 1 = 2
max(charJump[T[5]] , m-k+1) = max(3,2) = 3

Mismatch on P4 is found on 5th element of Pattern. So 2 character comparisons are done
charJump[T[15]] = charJump[P] = 3
m – k + 1 = 6 – 5 + 1 = 2
max(charJump[T[5]] , m-k+1) = max(3,2) = 3

No Mismatch on P5. The pattern is found on text. So 6 character comparisons are done

In Total: **16 character Comparisons**

Boyer-Moore algorithm

When there is a mismatch, the following shift will occur.

```
j+=max(charJump[T[j]],matchJump[k]);
```

P4:													P	A	Y	A	Y	A
P3:												P	A	Y	A	Y	A	
P2:						P	A	Y	A	Y	A							
P1:	P	A	P	A	Y	A												
T:	B	A	N	A	N	A		K	A	Y	A		P	A	P	A	Y	A
j	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Noted: The highlighted is the mismatch

Mismatch on P1 is found on 5th element of Pattern. So 2 character comparisons are done
charJump[T[5]] = charJump[N] = 6
matchJump[k] = matchJump[5] = 3
max(charJump[T[5]] , matchJump[k]) = max(6,3) = 6

Mismatch on P2 is found on 3rd element of Pattern. So 4 character comparisons are done
charJump[T[8]] = charJump[K] = 6
matchJump[k] = matchJump[3] = 9
max(charJump[T[8]] , matchJump[k]) = max(6,9) = 9

Mismatch on P3 is found on 6th element of Pattern. So 1 character comparisons are done
charJump[T[17]] = charJump[Y] = 1
matchJump[k] = matchJump[6] = 1
max(charJump[T[17]] , matchJump[k])) = max(1,1) = 1

No Mismatch on P4. The pattern is found on text. So 6 character comparisons are done

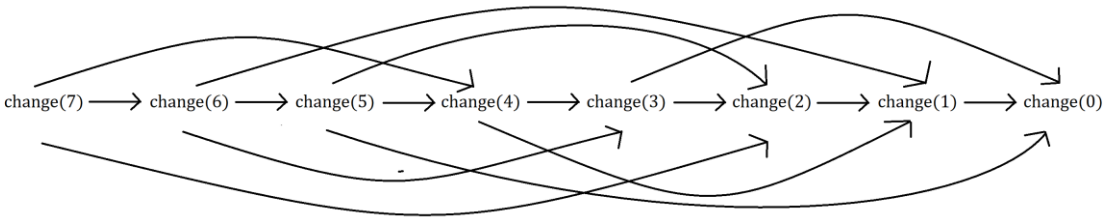
In Total: **13 character comparisons**

4 (a) (i)

$$change(n) = \begin{cases} 0, \forall\, i \in \{1, \ldots, m\}, n - d_i < 0 \\ min_{1 \le i \le m,\ n - d_i >= 0} (change(n - d_i) + 1), Otherwise \end{cases}$$

Editor's Note: The second line simply means that it take minimum of all the $change(n - d_i) + 1, provided\ that\ n - d_i >= 0$.

(ii)



(iii) The required algorithm is as below:

```
int coinchange (int n):
      int change[n+1]; // initialize dp array
      for (int j=0; j<=n; j++){
           change[j] = MAXINT  // MAXINT is the maximum integer value
            for(int i=0;i<m;i++)
                  if(j-val[i]>=0)
                        change[j] = min(change[j],change[j-val[i]]+1)
            if(change[j]== MAXINT)
                  change[j]=0; // If no changes means base case 0
      }
      return change[n]
```

(b) P problem is a class of decision problems that are Polynomially bounded
NP problem is the class of decision problems for which there is a polynomially bounded nondeterministic algorithm
NP-complete problem is a problem that is in NP and every problem Q in NP is reducible to D in polynomial time.

Longest common subsequence of 2 strings of length n **is a P problem**
There exists a dynamic programming solution define by LCS(n,n), where the optimal subsolution is from LCS(i-1, j-1), LCS(i-1,j) or LCS(i,j-1). The time complexity of the dynamic programming solution is O(N^2). Therefore it is polynomially bounded.

0/1 knapsack with n objects **is NOT a P problem**
There is no known polynomial time algorithm for this problem. The best algorithm for this is a dynamic programming algorithm which has a time complexity of O(nC) where C is the maximum capacity. The time complexity is in term of the value of input and not the size of the input. This is indeed a pseudo-polynomial.

Solver: Lee Zong Yu (ZLEE043@e.ntu.edu.sg)