Egg-dropping problem:
Sorting problem:

loop invariant (insertion sort):
We state these properties of A [j – 1] formally as a loop invariant:
At the start of each iteration of the for loop of lines 1–8, the subarray A [j – 1] consists of the elements originally in A [j – 1], but in sorted order.
Initialization: It is true prior to the first iteration of the loop. State why the loop is true before first loop.
Maintenance: If it is true before an iteration of the loop, it remains true before the
next iteration.
Termination: When the loop terminates, the invariant gives us a useful property
that helps show that the algorithm is correct.

INSERTION-SORT(A)

```
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1..j-1].
4      i = j - 1
5      while i > 0 and A[i] > key
6          A[i+1] = A[i]
7          i = i - 1
8      A[i+1] = key
```

Initialization: We start by showing that the loop invariant holds before the first loop iteration, when $j = 2$.[1] The subarray $A[1..j-1]$, therefore, consists of just the single element $A[1]$, which is in fact the original element in $A[1]$. Moreover, this subarray is sorted (trivially, of course), which shows that the loop invariant holds prior to the first iteration of the loop.

Maintenance: Next, we tackle the second property: showing that each iteration maintains the loop invariant. Informally, the body of the **for** loop works by moving $A[j-1]$, $A[j-2]$, $A[j-3]$, and so on by one position to the right until it finds the proper position for $A[j]$ (lines 4–7), at which point it inserts the value of $A[j]$ (line 8). The subarray $A[1..j]$ then consists of the elements originally in $A[1..j]$, but in sorted order. Incrementing j for the next iteration of the **for** loop then preserves the loop invariant.

A more formal treatment of the second property would require us to state and show a loop invariant for the **while** loop of lines 5–7. At this point, however, we prefer not to get bogged down in such formalism, and so we rely on our informal analysis to show that the second property holds for the outer loop.

Termination: Finally, we examine what happens when the loop terminates. The condition causing the **for** loop to terminate is that $j > A.length = n$. Because each loop iteration increases j by 1, we must have $j = n + 1$ at that time. Substituting $n + 1$ for j in the wording of loop invariant, we have that the subarray $A[1..n]$ consists of the elements originally in $A[1..n]$, but in sorted order. Observing that the subarray $A[1..n]$ is the entire array, we conclude that the entire array is sorted. Hence, the algorithm is correct.

We shall use this method of loop invariants to show correctness later in this chapter and in other chapters as well.

selection sort

b. selection sort

```
1,        for i =0 to n-1 do :
2,            minIndex = i
3,                for j =i+1 to n do:
4,                    if A[j] < A[minIndex]:
5,                        minIndex = j
6,                    end if
7,                end for
8,                if list[minIndex] < list[i]:
9,                    exchange A[minIndex] with
10,           end for
```

loop invariants:

Initialization:
We start by showing that the loop invariant holds before the first loop iteration, when i=0, we assume the index of minimum element is 0, then we go into the second for loop starts at line 4,

1

Name: Shuo WU   Student ID: 45266   Wustl key: wushuo

in this for loop, we iterate all the element from index 1(i+1) to the last one to check if the smallest element in the subarray A[1....last one], if the smallest element of subarray smaller than list [i], then exchange the value of these two, that move the smallest element of the subarray to the current index.

Maintain:
After finding the smallest element of the subarray and exchanging the smallest element and current element, the algorithm increase i which makes i= i+1, and then find the smallest element of the subarray from list[i+2] to list[list.length-1], then it compares the smallest element of the subarray with the current element which is A[i+1] to make sure it put the smallest element from subarray into A[i+1], so the algorithm remains true for the next iterations.

Termination:
When the loop terminates, the element in the last position (at the end of the array) is the biggest element of the whole array- list. Which i = list.length-1. Now we have the subarray list[0..list.length-2] sorted and list[list.length-1] which is the biggest element of the array list. Since it doesn't been selected from the previous iterations. Therefore, we conclude that the entire array is sorted. Hence the algorithm is correct

master method:

T(n) = aT(n/b)+f(n):
$= n^{\log_b a}$

a, linear search:

```
1,      for i =0 to A.length-1
2,              now = A[i]
3,              if v = now
4,                      return i
5,              end if
6,      end for
7,      return NIL
```

Loop invariants:
Initialization:
We start by showing that the loop invariant holds before the first loop iteration, when i=0. The now =A[0] which is the first item. If the v = A[0], the algorithm will return 0.
Else, if the array length = 1, it will return NIL, or it will go to the first iteration of the loop. So that shows the loop invariant holds prior to the first iteration of the loop.
Maintenance:
The body of the for loop will go through A[1], A[2]...A[A.length-3], A[A.length-2], A[A.length-1]. If A[x] != v, then the algorithm will check if A[x+1] =v or not. At any point, if A[x]= v(x< A.length-1), the algorithm will return the value of x as index, or it will go to the next iteration of loop. Therefore, the loop variant holds until the algorithm goes to the last position of the array which is A[A.length-1].
Termination:
Finally, if the algorithm finds any x (x means the index, and only the first x) that makes A[x]= v, it will terminate and return the value of x. Or after comparing A[A.length-1] with v, if the algorithm go through all the position of the array and doesn't find any x that holds A[x] = v, it will return NIL as line 7, we can say there isn't any elements that equals to v in the array A. Hence, the algorithm is correct.

一般情况下: initialization only talk about the first element in the array, then A.length =1, so sorted.

c. bubble-sort

```
1,      for i = 0 to A.length-1
2,              for j= A.length downto i+1
3,                      if A[j]<A[j-1]:
4,                              exchange A[j] with A[j-1]
```

Loop invariants:
Initialization:
For the first iteration, i=0, then we go into the second loop in lines 2 through 4 find the smallest element of the whole array A, it kinds of like make the smaller of two elements stand out, and then move the smaller element to first position and the bigger one to the later position, after first iteration, we get the smallest element of the whole array.
Maintain:
The for loop in lines 2 through 4 maintains the following loop invariant: At the start of each iteration, the position of the smallest element of A[i..n] is at most j. This is clearly true prior to the first iteration because the position of any element is at most A.length. To see that each iteration maintains the loop invariant, suppose that j = k and the position of the smallest element of A[i..n] is at most k. Then we compare A[k] to A[k − 1]. If A[k] < A[k − 1] then A[k − 1] is not the smallest element of A[i..n], so when we swap A[k] and A[k − 1] we know that the smallest element of A[i..n] must occur in the first k − 1 positions of the subarray, the maintaining the invariant. On the other hand, if A[k] ≥ A[k − 1] then the smallest element can't be A[k]. Since we do nothing, we conclude that the smallest element has position at most k − 1. Upon termination, the smallest element of A[i..n] is in position i.
Termination:
The for loop in lines 1 through 4 maintain the following loop invariant: At the start of each iteration the subarray A[1..i − 1] contains the i − 1 smallest elements of A in sorted order. Prior to the first iteration i = 1, and the first 0 elements of A are trivially sorted. To see that each iteration maintains the loop invariant, fix i and suppose that A[1..i − 1] contains the i − 1 smalle

Name: Shuo WU   Student ID: 45266   Wustl key: wushu

elements of A in sorted order. Then we run the loop in lines 2 through 4. We showed in part b that when this loop terminates, the smallest element of A[i..n] is in position i. Since the i − 1 smallest elements of A are already in A[1..i − 1], A[i] must be the i th smallest element of A. Therefore A[1..i] contains the i smallest elements of A in sorted order, maintaining the loop invariant. Upon termination, A[1..n] contains the n elements of A in sorted order as desired.

function loop invariant (need to state invariant):

d. Horner's rule

```
1, y=0
2, for i= n downto 0
3       y=ai +x*y
```

Initialization:
When i=n, (the first loop iteration). The value of y, therefore, equals to a_n, which is in fact the last element of polynomial.
For example if n=0, from the algorithm, we will get that y = a_0
And the polynomial gives us that y =a_0, therefore, they have the same result.
Showing that the loop invariant holds prior to the first iteration of the loop.
Maintain:
The for loop then makes y equals y = a_i+ x*(a_{i+1}+x*y') =

$$= a_i + x \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^k = a_i + x \sum_{k=1}^{n-i} a_{k+i} x^{k-1} = \sum_{k=0}^{n-i} a_{k+i} x^k$$

which shows that the loop invariant holds prior to terminate.
Termination:
At termination, i=0, so is summing up to n-1, therefore, the last iteration give us the correct fi output as shown before. We can see that the algorithm evaluated to the function. This is the v of the polynomial evaluated at x.

pseudo code for selection-sort and binary search:

Question 2,
a.
Pseudo-code:

```
def selection_sort(A, index):
    if index >= len(A):
        print(A)
        return
    minIndex = index
    for i in range (index+1,len(A)):
        if A[minIndex] > A[i]:
            minIndex = i
    if minIndex !=index:
        exchange A[index] with A[minIndex]
    return selection_sort(A, index+1)
```

T(n) = T(n-1)+ O(n)
The running time for worst-case = $\mathbf{\Theta(n^2)}$

b.
Pseudo-code:

```
def binary_search(A, target, start_index, end_index):
    if end_index < start_index:
        return -1
    else :
        mid = (end_index+start_index)/2
        if A[mid] == target:
            return mid
        elif A[mid] > target:
            return binary_search(A, target,start_index, mid-1)
        else:
            return binary_search(A ,target, mid+1, end_index)
```

T(n) = T(n/2)+ O(1)
T(n)= Θ (lg (n))

MERGE(A, p, q, r)

```
1   n1 = q - p + 1
2   n2 = r - q
3   let L[1..n1 + 1] and R[1..n2 + 1] be new arı
4   for i = 1 to n1
5       L[i] = A[p + i - 1]
6   for j = 1 to n2
7       R[j] = A[q + j]
8   L[n1 + 1] = ∞
9   R[n2 + 1] = ∞
10  i = 1
11  j = 1
12  for k = p to r
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
```

MERGE-SORT(A, p, r)

```
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

time complexity: T(n) = nlogn

trated in Figure 2.3, perform the $r - p + 1$ basic steps by maintaining the following loop invariant:

At the start of each iteration of the **for** loop of lines 12–17, the subarray $A[p..k-1]$ contains the $k - p$ smallest elements of $L[1..n_1+1]$ and $R[1..n_2+1]$, in sorted order. Moreover, $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

We must show that this loop invariant holds prior to the first iteration of the **for** loop of lines 12–17, that each iteration of the loop maintains the invariant, and that the invariant provides a useful property to show correctness when the loop terminates.

Initialization: Prior to the first iteration of the loop, we have $k = p$, so that the subarray $A[p..k-1]$ is empty. This empty subarray contains the $k - p = 0$ smallest elements of L and R, and since $i = j = 1$, both $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

Maintenance: To see that each iteration maintains the loop invariant, let us first suppose that $L[i] \le R[j]$. Then $L[i]$ is the smallest element not yet copied back into A. Because $A[p..k-1]$ contains the $k - p$ smallest elements, after line 14 copies $L[i]$ into $A[k]$, the subarray $A[p..k]$ will contain the $k - p + 1$ smallest elements. Incrementing k (in the **for** loop update) and i (in line 15) reestablishes the loop invariant for the next iteration. If instead $L[i] > R[j]$, then lines 16–17 perform the appropriate action to maintain the loop invariant.

Termination: At termination, $k = r + 1$. By the loop invariant, the subarray $A[p..k-1]$, which is $A[p..r]$, contains the $k - p = r - p + 1$ smallest elements of $L[1..n_1+1]$ and $R[1..n_2+1]$, in sorted order. The arrays L and R together contain $n_1 + n_2 + 2 = r - p + 3$ elements. All but the two largest have been copied back into A, and these two largest elements are the sentinels.

Find-max-subarray☹: find maximum subs array sum: T(n)=2T(n/2)+n

FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)

```
1   left-sum = -∞
2   sum = 0
3   for i = mid downto low
4       sum = sum + A[i]
5       if sum > left-sum
6           left-sum = sum
7           max-left = i
8   right-sum = -∞
9   sum = 0
10  for j = mid + 1 to high
11      sum = sum + A[j]
12      if sum > right-sum
13          right-sum = sum
14          max-right = j
15  return (max-left, max-right, left-sum + right-sum)
```

FIND-MAXIMUM-SUBARRAY(A, low, high)

```
1   if high == low
2       return (low, high, A[low])          // base case: only one e
3   else mid = ⌊(low + high)/2⌋
4       (left-low, left-high, left-sum) =
            FIND-MAXIMUM-SUBARRAY(A, low, mid)
5       (right-low, right-high, right-sum) =
            FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
6       (cross-low, cross-high, cross-sum) =
            FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
7       if left-sum ≥ right-sum and left-sum ≥ cross-sum
8           return (left-low, left-high, left-sum)
9       elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
10          return (right-low, right-high, right-sum)
11      else return (cross-low, cross-high, cross-sum)
```

how to deal with: T(n)= T(n-1) +T(n/2) +n?
assume T(n)≤S(n) = 2T(n-1) +n = 2^n
substitution:
T(n)= T(n-1) + T(n/2)+n ≤ 2^{n-1} +$2^{n/2}$ +n≤ 2*2^{n-1} =2^n
Selection in worst-case linear time:

A.
1. Divide the n elements of the input array into ⌈n/9⌉ groups of 9 elements each and at most one group made up of the remaining n mod 9 elements.
2. Find the median of each of the ⌈n/9⌉ groups by first insertion-sorting the elements of each group (of which there are at most 9) and then picking the median from the sorted list of group elements.
3. Use SELECT recursively to find the median x of the ⌈n/9⌉ medians found in step 2. (If there are an even number of medians, then by our convention, x is the lower median.)
4. Partition the input array around the median-of-medians x using the modified version of PARTITION. Let k be one more than the number of elements on the low side of the partition, so that x is the kth smallest element and there are n-k elements on the high side (or we can say, right side)of the partition.
5. If i = k, then return x. Otherwise, use SELECT recursively to find the ith smallest element on the low side if i < k, or the (i-k)th smallest element on the high side if i > k.

B.
For groups size of 9, we know at least half of medians found in step 2 are greater than or equal to the median-of-medians x. Thus, at least half of the ⌈n/9⌉ groups contribute at least 5 elements that are greater than x, except for the one group that has fewer than 9 elements if 9 does not divide n exactly, and the one group containing x itself. Discounting these two groups, it follows that the number of elements greater than x is at least:
5*(⌈1/2*⌈n/9⌉⌉-2) ≥ 5/18*n -10= ((5n/18)-10)
Similarly, at least 5/18*n-10 elements are less than x. thus, in the worst case, step 5 calls select recursively on at most 13/18*n+10 elements.

C.

$$T(n) \le \begin{cases} O(1) & \text{if } n < \text{constant (like: 140)} \\ T(n/9) + T(13n/18+10) + O(n) & \text{if } n \ge \text{constant (like: 140)} \end{cases}$$

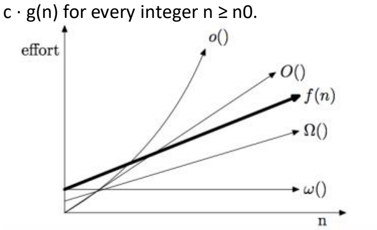
pseudo-code for finding element in matrix

pseudo-code:

```
def quarter_part( matrix, target):
    if (len(matrix)== 0 or len(matrix[0]) ==0 ):
        return False
    return helper(matrix,0,len(matrix)-1,0,len(matrix[0])-1,target)

def helper(matrix,row_first,row_end,col_start,col_end,target):
    if(row_end<row_first or col_end<col_start):
        return False
    row_mid = row_first+ (row_end-row_first)/2
    col_mid = col_start+ (col_end-col_start)/2
    if(matrix[row_mid][col_mid]==target):
        return True
    elif (matrix[row_mid][col_mid] >target):
        return  (helper(matrix,row_first,row_mid-1,col_start,col_mid-1,target)
          or helper(matrix, row_first, row_mid-1, col_mid, col_end, target)
          or  helper(matrix, row_mid, row_end, col_start, col_mid-1, target))
```

Definition (Little–o, o()), We say that f(n) is o(g(n)) if for any real constant c > 0, there exists an integer constant n0 ≥ 1 such that f(n) < c ∗ g(n) for every integer n ≥ n0.
Definition (Little–Omega, ω()):
We say that f(n) is ω(g(n)) if for any real constant c > 0, there exists
an integer constant n0 ≥ 1 such that f(n) > c · g(n) for every integer n ≥ n0.



quick sort :

QUICKSORT(A, p, r)

```
1  if p < r
2      q = PARTITION(A, p, r)
3      QUICKSORT(A, p, q - 1)
4      QUICKSORT(A, q + 1, r)
```

To sort an entire array A, the initial call is QUICKSORT(A, 1, $A.length$).

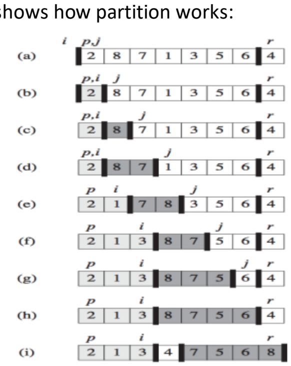
Partitioning the array

The key to the algorithm is the PARTITION procedure, which rearranges the subarray $A[p..r]$ in place.

PARTITION(A, p, r)

```
1  x = A[r]
2  i = p - 1
3  for j = p to r - 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

shows how partition works:



At the beginning of each iteration of the loop of lines 3–6, for any array index k,

1. If $p \le k \le i$, then $A[k] \le x$.
2. If $i + 1 \le k \le j - 1$, then $A[k] > x$.
3. If $k = r$, then $A[k] = x$.

Initialization: Prior to the first iteration of the loop, $i = p - 1$ and $j = p$. Because no values lie between p and i and no values lie between $i + 1$ and $j - 1$, the first two conditions of the loop invariant are trivially satisfied. The assignment in line 1 satisfies the third condition.

Maintenance: As Figure 7.3 shows, we consider two cases, depending on the outcome of the test in line 4. Figure 7.3(a) shows what happens when $A[j] > x$; the only action in the loop is to increment j. After j is incremented, condition 2 holds for $A[j-1]$ and all other entries remain unchanged. Figure 7.3(b) shows what happens when $A[j] \le x$; the loop increments i, swaps $A[i]$ and $A[j]$, and then increments j. Because of the swap, we now have that $A[i] \le x$, and condition 1 is satisfied. Similarly, we also have that $A[j-1] > x$, since the item that was swapped into $A[j-1]$ is, by the loop invariant, greater than x.

Termination: At termination, $j = r$. Therefore, every entry in the array is in one of the three sets described by the invariant, and we have partitioned the values in the array into three sets: those less than or equal to x, those greater than x, and a singleton set containing x.

Dynamic-programming:

BOTTOM-UP-CUT-ROD(p, n)

1 let $r[0..n]$ be a new array
2 $r[0] = 0$
3 **for** $j = 1$ **to** n
4 $\quad q = -\infty$
5 $\quad$ **for** $i = 1$ **to** j
6 $\qquad q = \max(q, p[i] + r[j-i])$
7 $\quad r[j] = q$
8 **return** $r[n]$

time complexity is $O(n^2)$, when each cut incurs a fixed cost of c, we need to change line 6 into: q= max(q,p[i]+r[j-1]-c) and line 5, for i=1 to j-1, since we might make no cuts, and modify line 4, q = p[j]

Egg-dropping:

```
# Function to get minimum number of trials needed in worst
# case with n eggs and k floors
def eggDrop(n, k):
    # A 2D table where entery eggFloor[i][j] will represent minimum
    # number of trials needed for i eggs and j floors.
    eggFloor = [[0 for x in range(k+1)] for x in range(n+1)]

    # We need one trial for one floor and0 trials for 0 floors
    for i in range(1, n+1):
        eggFloor[i][1] = 1
        eggFloor[i][0] = 0

    # We always need j trials for one egg and j floors.
    for j in range(1, k+1):
        eggFloor[1][j] = j

    # Fill rest of the entries in table using optimal substructure
    # property
    for i in range(2, n+1):
        for j in range(2, k+1):
            eggFloor[i][j] = INT_MAX
            for x in range(1, j+1):
                res = 1 + max(eggFloor[i-1][x-1], eggFloor[i][j-x])
                if res < eggFloor[i][j]:
                    eggFloor[i][j] = res

    # eggFloor[n][k] holds the result
    return eggFloor[n][k]
```

time: $T(n) = O(nk^2)$ space = $O(nk)$

longest increasing subsequence:

```
# Dynamic programming Python implementation of LIS problem

# lis returns length of the longest increasing subsequence
# in arr of size n
def lis(arr):
    n = len(arr)

    # Declare the list (array) for LIS and initialize LIS
    # values for all indexes
    lis = [1]*n

    # Compute optimized LIS values in bottom up manner
    for i in range (1 , n):
        for j in range(0 , i):
            if arr[i] > arr[j] and lis[i]< lis[j] + 1 :
                lis[i] = lis[j]+1

    # Initialize maximum to 0 to get the maximum of all
    # LIS
    maximum = 0

    # Pick maximum of all LIS values
    for i in range(n):
        maximum = max(maximum , lis[i])

    return maximum
# end of lis function
```

time $T(n) = O(n^2)$

knapsack problem:

```
# A Dynamic Programming based Python Program for 0-1 Knapsack problem
# Returns the maximum value that can be put in a knapsack of capacity W
def knapSack(W, wt, val, n):
    K = [[0 for x in range(W+1)] for x in range(n+1)]

    # Build table K[][] in bottom up manner
    for i in range(n+1):
        for w in range(W+1):
            if i==0 or w==0:
                K[i][w] = 0
            elif wt[i-1] <= w:
                K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]],  K[i-1][w])
            else:
                K[i][w] = K[i-1][w]

    return K[n][W]
```

time T(n)=O(nW) n: the number of items and W is the capacity of knapsack.

Find sum in a given set:

```
def isSubSum(st, n, sm):
    # The value of subset[i][j] will be
    # true if there is a subset of
    # set[0..j-1] with sum equal to i

    boolean[][] solution = new boolean[A.length + 1][sum + 1]

    # If sum is 0, then answer is true
    for i in range(0, n + 1):
        subset[i][0] = True

    # If sum is not 0 and set is empty,
    # then answer is false
    for i in range(1, sm + 1):
        subset[0][i] = False

    # Fill the subset table in botton
    # up manner
    for i in range(1, n + 1):
        for j in range(1, sm + 1):
            if (j < st[i - 1]):
                subset[i][j] = subset[i - 1][j]
            if (j >= st[i - 1] and subset[i][j] == False):
                subset[i][j] = subset[i - 1][j] or subset[i - 1][j - st[i - 1]]

    return subset[n][sm];
```

Elements \ Sum	0	1	2	3	4	5	6
0	T	F	F	F	F	F	F
3	T	F	F	T	F	F	F
2	T	F	T	T	F	T	F
7	T	F	T	T	F	T	F
1	T	T	T	T	T	T	T

Time T(n) = O(SUM*n) where n is elements.

Longest common subsequence:

LCS-LENGTH(X, Y)

$m = X.length$
$n = Y.length$
let $b[1..m, 1..n]$ and $c[0..m, 0..n]$ be new tables
for $i = 1$ **to** m
$\quad c[i, 0] = 0$
for $j = 0$ **to** n
$\quad c[0, j] = 0$
for $i = 1$ **to** m
$\quad$ **for** $j = 1$ **to** n
$\qquad$ **if** $x_i == y_j$
$\qquad\quad c[i, j] = c[i-1, j-1] + 1$
$\qquad\quad b[i, j] =$ "↖"
$\qquad$ **elseif** $c[i-1, j] \ge c[i, j-1]$
$\qquad\quad c[i, j] = c[i-1, j]$
$\qquad\quad b[i, j] =$ "↑"
$\qquad$ **else** $c[i, j] = c[i, j-1]$
$\qquad\quad b[i, j] =$ "←"
return c and b

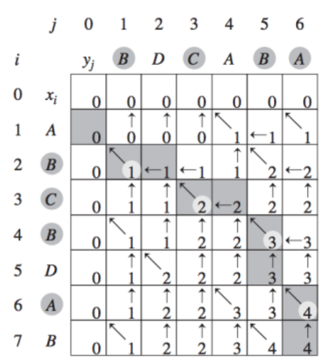
Theorem 15.1 (Optimal substructure of an LCS)
Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \ne y_n$, then $z_k \ne x_m$ implies that Z is an LCS of X_{m-1} and Y.
3. If $x_m \ne y_n$, then $z_k \ne y_n$ implies that Z is an LCS of X and Y_{n-1}.

Proof (1) If $z_k \ne x_m$, then we could append $x_m = y_n$ to Z to obtain a common subsequence of X and Y of length $k + 1$, contradicting the supposition that Z is a *longest* common subsequence of X and Y. Thus, we must have $z_k = x_m = y_n$. Now, the prefix Z_{k-1} is a length-$(k-1)$ common subsequence of X_{m-1} and Y_{n-1}. We wish to show that it is an LCS. Suppose for the purpose of contradiction that there exists a common subsequence W of X_{m-1} and Y_{n-1} with length greater than $k - 1$. Then, appending $x_m = y_n$ to W produces a common subsequence of X and Y whose length is greater than k, which is a contradiction.

(2) If $z_k \ne x_m$, then Z is a common subsequence of X_{m-1} and Y. If there were a common subsequence W of X_{m-1} and Y with length greater than k, then W would also be a common subsequence of X_m and Y, contradicting the assumption that Z is an LCS of X and Y.

(3) The proof is symmetric to (2). ■



DYNAMIC_ACTIVITY_SELECTOR(S):
initialize c[i,j] = 0
for i <- 1 to n
do for j <- 2 to n
do if i >= j
then c[i,j] <- 0
else
for k <- i+1 to j-1
do if c[i,j] < c[i,k] + c[k,j] + 1
then c[i,j] <- c[i,k] + c[k,j] + 1
s[i,j] <- k

T(n)= $O(n^3)$

A,
Let $X = \{x_i, x_{i+1}, \ldots, x_j\}$, we sort and re-label the points so that $x_i \le x_{i+1} \le \cdots \le x_j$.
Let S_{ij} be the smallest set of intervals we build (the solution), contains some intervals $I_k = [x_k, x_k + 1]$, where $i \le k \le j$.
Same definition for S_{ik} and S_{kj}.
$X_{ik} = S_{ik} \cap X$
$X_{kj} = S_{kj} \cap X$
1, S_{ik} is the optimal solution for sub-problem X_{ik}
2, S_{kj} is the optimal solution for sub-problem X_{kj}

B,
Consider any nonempty sub-problem X_k, and let x_1 be a point in X_k with the smallest position. Then interval $I_1 = [x_1, x_1+1]$ is included in some of smallest set of intervals of X_k.

C,
Let S_k be the smallest set of unit-length closed intervals of X_k, and let $I'_1 = [x'_1, x'_1+1]$ be the first interval in S_k. If $I'_1 = I_1$, then we are done, since we have shown that I_1 is in some smallest set of intervals of X_k.
$I'_1 = [x'_1, x'_1+1]$, and $I_1 = [x_1, x_1+1]$.
If $x'_1 \ne x_1$:
If $x'_1 > x_1$, we can say S_k is not a solution for the question since the first point x_1 isn't included in S_k.
If $x'_1 < x_1$, as x_1 is the leftmost point, there are no points from X_k contained in the interval $[x'_1, x_1]$. Therefore, we could simply replace the interval $[x'_1, x'_1+1]$ in S_k (which is S_{opt}) with the interval $[x_1, x_1+1]$ such that the new set of intervals $S_{k\text{-revise}}$ is still optimal (as $|S_k| = |S_{k\text{-revise}}|$ and all points are covered).

```
def Unit_length_closed_intervals(s, X):
    new_x = list()
    if len(X) ==0:
        return s
    else:
        X.sort()
        first_point = X[0]
        #print(first_point)
        answer = [first_point,first_point+1]
        s.append(answer)

        for i in range(0,len(X)):
            if X[i] <= (first_point+1):
                continue
            else:
                new_x.append(X[i])
        return Unit_length_closed_intervals(s,new_x)

X=[6,1,12,3,14,5,5.3,4,5.5,1.3,2.1]

s = []
Unit_length_closed_intervals(s,X)
print(s)
```

```
def Unit_length_closed_intervals_Iterative(s, X):
    X.sort()

    while len(X) !=0:
            #print (i)
            first_point = X[0]
            new_x = list()
            for i in range(0, len(X)):
                if X[i] <= (first_point + 1):
                    continue
                else:
                    new_x.append(X[i])
            X = new_x
            answer = [first_point,first_point+1]
            s.append(answer)

    return s

X=[6,1,12,3,14,5,5.3,4,5.5,1.3,2.1]

s = []
s2 = []
Unit_length_closed_intervals(s2,X)
print(s2)
Unit_length_closed_intervals_Iterative(s,X)
print(s)
```

Unit_length_clo... › while len(X) !=0

RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)

$m = k + 1$
while $m \le n$ and $s[m] < f[k]$ // find the first activity in S_k to
$\quad m = m + 1$
if $m \le n$
$\quad$ **return** $\{a_m\} \cup$ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
else return $\emptyset$

Huffman-coding:

Steps to build Huffman Tree

Input is array of unique characters along with their frequency of occurrences and output is Huffman Tree.

1. Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root)

2. Extract two nodes with the minimum frequency from the min heap.

3. Create a new internal node with frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.

4. Repeat steps#2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete.

HUFFMAN(C)

$n = |C|$
$Q = C$
for $i = 1$ **to** $n - 1$
$\quad$ allocate a new node z
$\quad z.left = x =$ EXTRACT-MIN(Q)
$\quad z.right = y =$ EXTRACT-MIN(Q)
$\quad z.freq = x.freq + y.freq$
$\quad$ INSERT(Q, z)
return EXTRACT-MIN(Q) // return

Bellman:

BELLMAN-FORD(G, w, s)

INITIALIZE-SINGLE-SOURCE(G, s)
for $i = 1$ **to** $|G.V| - 1$
$\quad$ **for** each edge $(u, v) \in G.E$
$\qquad$ RELAX(u, v, w)
for each edge $(u, v) \in G.E$
$\quad$ **if** $v.d > u.d + w(u, v)$
$\qquad$ **return** FALSE
return TRUE

RELAX(u, v, w)

if $v.d > u.d + w(u, v)$
$\quad v.d = u.d + w(u, v)$
$\quad v.\pi = u$

DIJKSTRA(G, w, s)

INITIALIZE-SINGLE-SOURCE(G, s)
$S = \emptyset$
$Q = G.V$
while $Q \ne \emptyset$
$\quad u =$ EXTRACT-MIN(Q)
$\quad S = S \cup \{u\}$
$\quad$ **for** each vertex $v \in G.Adj[u]$
$\qquad$ RELAX(u, v, w)

Suppose that we have a set of activities to schedule among a large number of lecture halls, where any activity can take place in any lecture hall. We wish to schedule all the activities using as few lecture halls as possible. Give an efficient greedy algorithm to determine which activity should use which lecture hall.

Maintain a set of free (but already used) lecture halls F and currently busy lecture halls B. Sort the classes by start time. For each new start time which you encounter, remove a lecture hall from F, schedule the class in that room, and add the lecture hall to B. If F is empty, add a new, unused lecture hall to F. When a class finishes, remove its lecture hall from B and add it to F. Why this is optimal: Suppose we have just started using the m^{th} lecture hall for the first time. This only happens when ever classroom ever used before is in B. But this means that there are m classes occurring simultaneously, so it is necessary to have m distinct lecture halls in use.

16.1-5
Consider a modification to the activity-selection problem in which each activity a_i has, in addition to a start and finish time, a value v_i. The objective is no longer to maximize the number of activities scheduled, but instead to maximize the total value of the activities scheduled. That is, we wish to choose a set A of compatible activities such that $\sum_{a_k \in A} v_k$ is maximized. Give a polynomial-time algorithm for this problem.

$$c[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset , \\ \max\limits_{a_k \in S_{ij}} \{c[i,k] + c[k,j] + 1\} & \text{if } S_{ij} \ne \emptyset . \end{cases} \qquad (16.2)$$

Run a dynamic programming solution based off of the equation (16.2) where the second case has "1" replaced with "v_k". Since the subproblems are still indexed by a pair of activities, and each calculation requires taking the minimum over some set of size $\le |S_{ij}| \in O(n)$. The total runtime is bounded by $O(n^3)$.

just change value of 1 into V_k in the left corner.

```
DYNAMIC-ACTIVITY-SELECTOR(s , f , n)
        let c[0..n+1, 0..n+1] and act[0..n+1, 0..n+1]
    for i = 0 to n
        c[i , i] = 0



        c[i, i+1] = 0
    c[n+1, n+1] = 0
    for l = 2 to n+1
        for i = 0 to n-l+1
            j = i+l
            c[i, j] = 0
            k = j -1
            while f[i] < f[k]
                if f[i] <= s[k] and f[k] <= s[j]
                    and c[i, k] + c[k, j] + 1 > c[i, j] do
                    c[i, j] = c[i, k] + c[k, j] + 1
                    act[i, j] = k
                k = k -1
     print "A max size set of mutually compatible activities "
     print c[0, n+1]
     print "The set contains "
     PRINT-ACTIVITIES(c, act, 0, n+1)

  PRINT-ACTIVITIES(c, act, i, j)
        if c[i, j] > 0
        k = act[i, j]
        print k
        PRINT-ACTIVITIES(c, act, i, k)
        PRINT-ACTIVITIES(c, act, k, j)
```

We create two fictitious activities, a_0 with $f_0 = 0$ and a_{n+1} with $s_{n+1} = \infty$. We are interested in a maximum-size set $A_{0,n+1}$ of mutually compatible activities in $S_{0,n+1}$. We'll use tables $c[0..n+1, 0..n+1]$ as in recurrence (16.2) (so that $c[i,j] = |A_{ij}|$, and $act[0..n+1, 0..n+1]$, where $act[i,j]$ is the activity k that we choose to put into A_{ij}.

We fill the tables in according to increasing difference $j - i$, which we denote by l in the pseudocode. Since $S_{ij} = \emptyset$ if $j - i < 2$, we initialize $c[i,i] = 0$ for all i and $c[i, i+1] = 0$ for $0 \le i \le n$. As in RECURSIVE-ACTIVITY-SELECTOR and GREEDY-ACTIVITY-SELECTOR, the start and finish times are given as arrays s and f, where we assume that the arrays already include the two fictitious activities and that the activities are sorted by monotonically increasing finish time.

ds