CS325 MT1

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Big O – g(n) is an asymptotic upper bound for f(n). “Less than” f(n) =< g(n)  Big Omega - g(n) is an … lower bound for f(n). “greater than” f(n) >= g(n)  Big Theta - g(n) is an asymptotic tight bound for f(n). “equal to” f(n) = g(n)  Iterative – translate loops in code to summations  Recursive – translate recursion into summations | |  |  |  | | --- | --- | --- | | Insertion Sort | Builds 1x a time | O(n^2) | | MergeSort | Divide/Conquer | O(n log n) | | Binary Search | Tree | O(log n) |   Mergesort asymptotically beats sort in worst case  O(n log n) grows more slowly than O(n^2) |

Method for solving recurrences

1. Iteration – convert recurrence into a summation & bind it using a known series
2. Substitution – T(n) = O(g(n)). Induction hypo
3. Recursion-tree – convert each recurrence into a tree; each node is cost incurred
4. Master Method – applied to T(n) = a\*T(n/b) + f(n), where a, b >1 & f(n) > 0. Leaves are n^logb(a)
   1. Case 1: If f(n) = O(n^logb(a)-e), for e > 0, then T(N) = Theta(n^logb(a))
   2. Case 2: If f(n) = Theta(n^logb(a), then T(N) = Theta(n^logb(a) \* logn)
   3. Case 3: If f(n) = Omega(n^logb(a)+e), for e > 0, and if af(n/b) =< cf(n) for some c < 1, then T(N) = Theta(f(n))
5. Muster Method – T(n) = aT(n-b) + f(n). If O or T(n^d), then O(n^d) if a < 1, O(n^(d+1)) if a = 1, O(n^d\*a^(n/b)) if a > 1.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | O(n) | Constant | | O(log n) | Logarithmic | | O(n) | Linear | | O(n^2) | Quadratic | | O(n^3) | Cubic | | O(n^k) | Polynomical | | O(k^n) | Exponential | | O(n!) | Factorial | | Elements of DP: Optimal substructure, overlapping subproblems  Find optimal formula to fill in table/array. Base cases, then to fill table, use looping structure/formula.  Greedy Algo – at each phase, you take the optimal w/o regard for future consequences, local optimum vs global  Sort by finishing times (or start times in reverse order). Show an optimal solution w/ a greedy choice  Both – Greedy usually more efficient than DP. But greedy doesn’t guarantee an optimal  Greedy Examples: knapsack, coin change, data compression, scheduling (activity, task, minimizing time, deadline), graph algos like breadth first search, dijkstras (shortest path), and minimum spanning trees |
| **Last-To-Start Activities Greedy (O(n log n))**  function greedy (a, s,f)  n = length of A  A = results array  index = n  for m = n-1 down to 1  If f[m] <= s[index]  add a[m] to results array  index = m  return A | **Scheduling (with penalty) – Greedy.**  Arrange the jobs in decreasing order of the penalties 𝑝1 ≥ 𝑝2 ≥ ⋯ \_≥ 𝑝𝑛 \_and add them in this order  To add job 𝑗𝑖 \_,  if any time interval 𝑀𝑙 \_is available for 1 ≤ 𝑙 \_≤ 𝑑𝑖 \_, then schedule 𝑗𝑖 \_in the last such available interval.  else schedule 𝑗𝑖 \_in the first available interval starting backwards from 𝑀𝑛 \_  Sorting by penalties – O(nlogn). To find spot in order (insertion/quick sorts): O(n^2) |

|  |  |
| --- | --- |
| **Binary Search – Div & Conquer**  def binarySearch (arr, l, r, x):      if r >= l:          mid = l + (r - l)/2          if arr[mid] == x: # If element is present at the middle itself              return mid          elif arr[mid] > x: #if elem is smaller than mid, then in left              return binarySearch(arr, l, mid-1, x)          else: #can only be present in right subarray              return binarySearch(arr, mid+1, r, x)      else:          return -1 # Element is not present in the array  Time: T(n) = T(n/2) + c or **O(log n)**  Space: Iterative: O(1); Recursion O(log n) | **0-1 Knapsack DP (Greedy is not optimal)**  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] (max value knapsack can hold)  O(nW); n is the number of items,W is the capacity |
| **Huffman code - O(nlogn)** | **Canoe Cost + Sequence O(n^2)**  CanoeSequence(R)  n <- #rows[R]  C[1]=0 , P[1]=0  for i<-2 to n  4min <- R[1,i]  P[i]<-1  for k <-2 to i -1  if C[k]+ R[k, i] < min  min <-C[k]+ R[k,i]  P[i]<-k  C[i], min  return P (P is array of locations); C[n] (minimal cost) |
| **MergeSort O(n log n)**  def mergeSort(listToMerge):  # Console output to confirm our inputs  # print("MergeSorting: ",listToMerge)  if len(listToMerge)>1:  mid = len(listToMerge)//2  lefthalf = listToMerge[:mid]  righthalf = listToMerge[mid:]  mergeSort(lefthalf)  mergeSort(righthalf)  i=0, j = 0, k = 0  # Sorting Each Half of the List, to be then mergesorted again  while i < len(lefthalf) and j < len(righthalf):  if lefthalf[i] < righthalf[j]:  listToMerge[k]=lefthalf[i]  i=i+1  else:  listToMerge[k]=righthalf[j]  j=j+1  k=k+1  while i < len(lefthalf):  listToMerge[k]=lefthalf[i]  i=i+1  k=k+1  while j < len(righthalf):  listToMerge[k]=righthalf[j]  j=j+1  k=k+1 | **Insertion Sort O(n^2)**  def insertionSort(arr):  for i in range(1, len(arr)):  curr = arr[i]  # As you go through the array, find elements that are creater than curr, then move to one position ahead of curr's position  j = i-1  while j >= 0 and curr < arr[j]:  arr[j+1] = arr[j]  j -= 1  arr[j+1] = curr  **Find Min-Max of unsorted array / Divide&conquer**  If |A|=1 then return min=max=A[0]  Divide A into two equal subsets A1 and A2  (min1, max1) = MIN-MAX(A1)  (min2, max2) = MIN-MAX(A2)  If min1 <= min2 then min = min1  Else min = min2  If max1 >= max2 then max = max1  Else max = max2  Return (min,max)  Recurrence: T(n) = 2T(n/2) + 2 **or** T(n) = 2T(n/2) + c  Solution: T(n) is O(n)  Master method: a=2, b=2, f(n) = c , logba = 1, f(n) = O(n) so case 1. |
| **LCS – Longest Common Subsequence - O(mn)**  def lcs(X , Y):      m = len(X)      n = len(Y)      L = [[None]\*(n+1) for i in xrange(m+1)]      for i in range(m+1):          for j in range(n+1):              if i == 0 or j == 0 :                  L[i][j] = 0              elif X[i-1] == Y[j-1]:                  L[i][j] = L[i-1][j-1]+1              else:                  L[i][j] = max(L[i-1][j] , L[i][j-1])      return L[m][n] | **LIS – Longest Increasing Subsequence – O(n^2)**  def lis(arr):      n = len(arr)      lis = [1]\*n      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      maximum = 0      for i in range(n):          maximum = max(maximum , lis[i])      return maximum |
| **Coin Change - Iterative**  def count(S, m, n ):      # If n is 0 then there is 1      if (n == 0):          return 1      # If n is less than 0 then no solution      if (n < 0):          return 0;      # If there are no coins and n is greater than 0, then no solution      if (m <=0 and n >= 1):          return 0      # count is sum of solutions (i)      # including S[m-1] (ii) excluding S[m-1]      return count( S, m - 1, n ) + count( S, m, n-S[m-1] ); | **DP Way: Time: O(mn); Space: O(n)**  def count(S, m, n):      # table[i] will be storing the number of solutions for      # value i. We need n+1 rows as the table is constructed      # in bottom up manner using the base case (n = 0)      # Initialize all table values as 0      table = [0 for k in range(n+1)]      table[0] = 1 # Base case (If given value is 0)      # Pick all coins one by one and update the table[] values      # after the index greater than or equal to the value of the      # picked coin      for i in range(0,m):          for j in range(S[i],n+1):              table[j] += table[j-S[i]]      return table[n] |
| **Hotel DP**  Let S[j] be the minimum total penalty when you stop at hotel j.  Let S[0] = 0  For j >= 1, j <= n  S[j] = inf  For i = 0, i < j  S[j] = min { S[j], S[i] + (200- (aj – ai))2 }  Return: S[n] | **Consulting – Max Fees – DP**  Let OPT(i,d) be the maximum fee collected for considering projects 1,…, i with d days available. The base cases are OPT(i, 0) = 0 for i = 1, …, n and OPT(0, d) = 0 for d = 1, …, D.  For i = 1 to n {  For j = 1 to D { //updated from d to j  If (di > j ) { OPT(i, j) = OPT( i-1, j) // not enough time to complete projecti}  Else { OPT(i, j) = max ( OPT( i-1, j), // Don’t complete project i OPT( i-1, j-di) + fi ) // Complete project I and earn fee fi ) } } }  O(nD) |

The running time of a dynamic programming algorithm is always Θ(P) where P is the number of subproblems. -False

If f(n) = O(g(n)) and g(n) = O(h(n)), then f(n) = O(h(n)) - True