TEST 2 Guide

**SPLAY TREES** = BST not necessarily balance so height can be > log n

Depth of node v = d(v) = distance from root to v by counting edges

Height of node v = h(v) = distance from v to farthest leaf in subtree rooted at v by counting edges

Height of overall tree is height of root

Size of node v = |v| = number of nodes in subtree rooted at v inclusive of v

N = size of whole tree

Minimum height of any BST = [log n] + 1 with 1 adjusting for N

Operations: search, insert and delete require splaying operation

* Search uses standard BST search but then splays found node
* Insert uses standard BST insert but then splays node inserted
* Delete finds node, splays it, then deletes it from root position which creates 2 sub-trees which are joined by finding largest node in left sub-tree, splaying it, then making the right subtree its child

Rotations are used to splay – 2 kinds which start by first rotating the splayed nodes’ parent but then leaving the depth of that parent unchanged when competed after the second rotation which is at the splayed node – rotations run in constant time

1. Roller-Coaster or Zig-Zig: used when splayed node is a left child of a left child or right child of a right child – “straight path”
2. Zig-Zag: used when splayed node is a right child of a left child or a left child of a right child

Splaying a node v requires time proportional to d(v) which is v’s depth before splaying

Running time of splaying: (via Potential Method) by adding up the amortized costs of all the rotations (based on r(v) being the rank of a node which = the log of the number of nodes in the subtree rooted at v inclusive so a leaf has rank 0) we find that the total amortized cost of splaying a node v is at most 1 + 3rfinal(v) - 3rstart(v) [because you have this equation for all the double rotations along the way to the root, but all but the first and last term cancel – can get as tight as 1 + rfinal(v) - rstart(v)] but of course after the splay v becomes the root hence rfinal(v) = └log n┘ therefore the amortized cost of a splay is at most 3 log n + 1 = O(log n) (took floor out as this is worst bound), SO ULTIMATELY, any operation on a splay tree requires “walking down” to the node which is doe in constant time + the splay operation which is done in O(log n) time so all splay tree operations run in an amortized time of O(log n)

**DIVIDE AND CONQUER** = break problem into non-overlapping subproblems then combine solutions into answer

Mergesort – runs in divide O(1) \* 2T(n/2) which is O(log n) \* merge which is O(n) for total of O(n log n)

Closest Pair of Points = divide in half, find closest pair of points in each half which = delta then find distances of points delta away from division line – has recurrence relation T(n) ≤ 2T(n/2) + O(n log n) => T(n) = O(n log2 n) BUT if we keep two sorted lists, one by x and one by y we get to T(n) ≤ 2T(n/2) + O(n) => T(n) = O(n log n)

Brute force running time = check all pairs = n (n – 1) = Θ(n2) comparisons

1-D version (when all points on a line) = O(n log n) as you can sort all points and go left-to-right on line which takes linear time

2-D running time raw: T(n) ≤ 2T(n/2) + (n log n) = O(n log2 n) but we can get to O(n log n) by keeping 2 sorted lists of points (x, y)

Integer Multiplication:

old-school = T(n) = 4T(n/2) + Θ(n) => Θ(n2) but

Karatsuba’s = T(n) ≤ T(⌊n/2⌋) + T(|n/2|) + T(1 + |n/2|) + Θ(n) => T(n) = O(nlog23) = O(n1.585)

Matrix Multiplication: note here we use n to denote the dimension of the matrix not the input size which for square = n2

M(X) \* M(Y) = M(Z) is solved by Zij = ith row of X \* jth column of Y for example: dot products =

X row 1/Y col 1 = (1\*5)+(2\*7) = Z 1/1 = 19 and X row 2/Y col 2 = (3\*6)+(4\*8) = Z 2/2 =50

naïve algorithm involves 3 loops and non-Strassen divide algorithm = T(n) = 8T(n/2) + O(n2) thus O(n3) via Master Method Case 1 – YouTube says Zij = Σ[k, 1-n] XikYkj which is Θ(n) so Θ(n) for Zij \* n2 entries for i and j so = O(n3)

NOTE: each n/2 sub-matrix is 25% of the original matrix since the square matrix is n x n in dimension

Strassen = T(n) = 7T(n/2) + O(n2) = O(nlogba) = O(nlog27) = O(n2.8074) by doing the following on 2x2 block matrices:

* + 1. Divide: partition A and B into n/2 x n/2 blocks
    2. Compute: 14 n/2 x n/2 matrices via 10 matrix additions (14 comes from finding B12 – B22 and A11 + A12 shown in above diagram before making recursive calls)
    3. Conquer: perform 7 n/2 x n/2 matrix multiplications recursively (7 recursive calls result in 7 products)
    4. Combine: 7 products into 4 terms using 8 matrix additions

The 7 Products that are each derived from a recursive call instead of 8 for a straight recursive algorithm:

P1 = A(F-H), P2 = (A+B)H, P3 = (C+D)E, P4 = D(G-E), P5 = (A+D)(E+H), P6 = (B-D)(G+H), P7 = (A-C)(E+F)

7 multiplications total instead of 8 so we get T(n) = 7T(n/2) + O(n2) = O(nlogba) = O(nlog27) = O(n2.8074) instead of O(n3)

**DYNAMIC PROGRAMMING** = break problem up into a series of overlapping sub-problems and build up solutions to larger and larger sub-problems

Weighted Interval Scheduling = finds max weight subset of mutually compatible jobs and returns total weight of subset of jobs

1. Sort jobs by finish time and assign them a number starting at 1 for job with earliest finish time (these are jobs j)
2. Define p(j) which is all jobs i compatible with job j (all jobs above the p value are compatible due to being sorted)
3. Define OPT(j) = value of optimal solution to problem consisting of job requests 1, 2, . . . , j
4. CASE 1: OPT selects job j - Cannot use incompatible jobs { p(j) + 1, p(j) + 2, . . . , j – 1 } BUT Must include optimal solution to the problem consisting of remaining compatible jobs 1, 2, . . . , p(j)
5. CASE 2: OPT does not select job j - Must include optimal solution to the problem consisting of remaining compatible jobs 1, 2, . . . , j – 1

**pseudocode: takes(number, start, finish, weight) first Sort jobs by finish // sorting take O(n log n) time**

**Compute p(1), p(2), . . . , p(n) // the job w/latest but previous finishing time that does not overlap**

**// computing p requires O(n) time**

**For j = 1 to n { // Difference from brute force is this MATRIX of values that are referred**

**M[j] = empty } // to instead of recalculating the same OPT over and over**

**M[0] = 0 }**

**Compute-Opt(n) // she says this is main call to function**

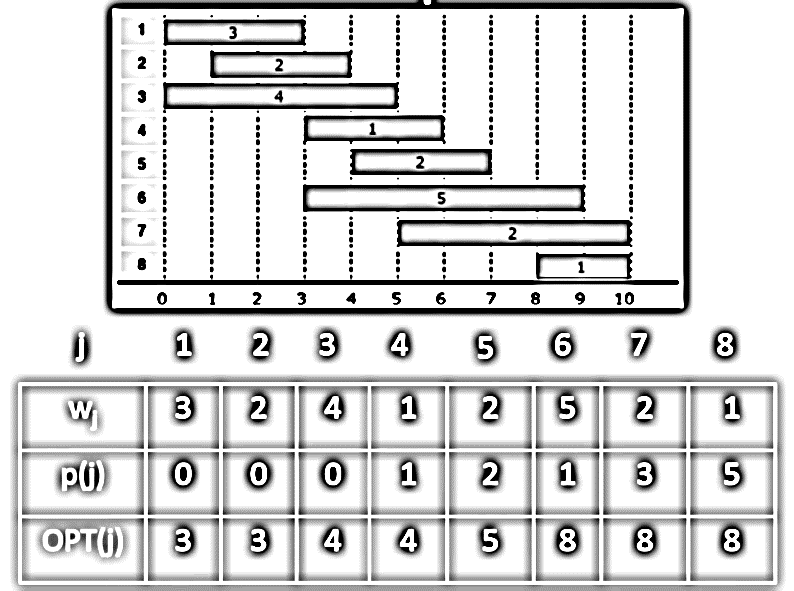
**Compute-Opt(j){ // Computng OPT(j) takes O(1) time**

**if(M[j} is empty) // if we have not previously solved the OPT before then do so**

**M[j] = max(vj + M-Compute-Opt(p(j)), M-Compute-Opt(j – 1))**

**Return M[j] // if already solved return old solution otherwise return new solution**

**} // overall run time of M-Compute-Opt(n) is O(n)**



**How it works:** wj = weight then Compute-Opt(p(j)) = the table entry for OPT(j) for the p(j) value so for example when j = 5, wj + Compute-Opt(p(5)) = 2 + {the value of OPT[p(j)]} and here p(j) = 2 so the OPT(2) was 3 so ultimately you add 2 to 3 = 5 by referring back to OPT(1) value and adding that to w5

Once Opt(j) does not grow pick first value – here 6

Weighted Interval Scheduling

Running Time = O(n log n) or if jobs are presorted O(n)

Weighted Interval Scheduling: Finding a Solution – dynamic programming algorithms compute optimal final value, but do not give you the solution itself. Here is routine for pulling actual solution out – **pseudocode**

**Run M-Computer-Opt(n)**

**Run Find-Solution(n)**

**Find-Solution(j) {**

**if ( j == 0 )**

**Output nothing**

**else if ( vj + M[p(j)] > M[j – 1]**

**Print j**

**Find solution(p(j))**

**else**

**Find-Solution(j – 1)**

**}**

**KNAPSACK** = fill weight limited knapsack in a way that maximizes total value loaded

Create Matrix that has i ordinal values as the x axis and weight and value as the y axis with each axis starting at 0

Variable OPT(i, w) = max profit subset of items 1 through i with weight limit w

CASE 1: OPT(i, w) does not select item i buth rather selects best of {1, 2, . . . , i-1} using weight limit w

CASE 2: OPT(i, w) selects item i and new weight limit = w – wi and OPT selects best of {1, 2, . . . , i-1} using new weight limit

**KNAPSACK Method = draw table with weight 0 to (W+1) across top and number i and value down left side in parallel**

Bottom-Up – can be considered as filling up an n-by-W array using following **pseudocode**

**Input: n, W, wi, . . . , wn, vi, . . . , vn  // bottom-up = recurses to base case (leaf) then works way up (tree)**

**for w = 0 to W // O(W) where w = weight count and W = weight maximum**

**M[0, w] = 0**

**for i = 1 to n // i stands for item number // O(n W) for nested**

**for w = 1 to W // iterates 1-to-total not via weights of items**

**if (wi > w) // compare item weight to weight count// O(1)**

**M[i, w] = M[i – 1, w] // M[i – 1, w] = value above // O(1)**

**Else**

**M[i, w] = max {M[i – 1, w], vi + M[i – 1, w – wi]} // w iteration count - wi item weight //up to 2 x O(1) so O(1)**

**return M[n, W] // then you look up value at M[i -1, w iteration count - wi item weight]**

Knapsack Algorithm

if i or w == 0 which means if weight of current i == running total weight, then M[i, w] // diff than above

if wi > w then M[i, w] = M[i-1, w}

else M[i, w] = max {M[i – 1, w], vi + M[i – 1, w – wi]} // w – wi = move left running weight count – weight of i

// so you go up 1 level (i – 1) then left w – wi

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | 0 base -> Weight = W = 6 | | | | | | |
| i |  | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| wt | val | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | 3 | 0 | 1 | 3 | 4 | 4 | 4 | 4 |
| 3 | 4 | 0 | 1 | 3 | 4 | 5 | 7 | 8 |
| 4 | 6 | 0 | 1 | 3 | 4 | 6 | 7 | 9 |

NOTE: in W = 3 and wt = 2 the term vi + M[i – 1, w – wi] => 3 + M[1, 3-2] as w = 3 and wi = 2 so add M[1, 1] = 1

NOTE: in W = 1-2 and wt = 3 the weight of i = 3 so w is less than wi so we just bring M[i-1, w] down for value

Knapsack Running Time: Running Time = Θ(n W)

Not polynomial on input size! **Psuedo-polynomial means dependent on the numeric value which is also the length of input in binary** – looks like only primality testing and knapsack are pseudo-polynomial

* + 1. An algorithm whose worst-case time complexity depends on numeric value of input (**not number of inputs**) is called Pseudo-polynomial algorithm.
    2. E.G. primality testing with this simple algorithm. It looks like O(n) but is really O(2logn) due the length of the inputs because the length of the input in bits is going to grow exponentially:

**for (i = 2; i < n – 1; i++) {**

**if n % i == 0 return false**

**} return true**

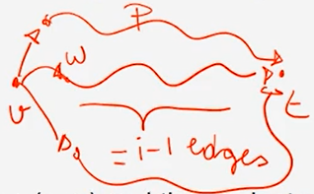
“Pseudo-polynomial” because max weight W in O(n W) = O(2logW n) is represented by 2logW in bits in binary

* + 1. W is exponential in the size of W - 2logW = W also Wlog2 = W

Decision version of Knapsack is NP-complete as there is no polynomial algorithm to solve, HOWEVER

* + 1. If the capacity of the knapsack can be shown to be a constant or multiple of 2 then polynomial available as asymptotically the running times depends on n not W

There exists a knapsack approximation algorithm that produces a feasible solution that has a value within 0.01% of optimum

**BELLMAN-FORD** – shortest path = find shortest path from s-t – Dynamic Programming: find the path from every node v to the destination node t instead of from the start node to every other node like Dijkstra

OPT(i, v) is the sub-problem that gives shortest v-t path using at most i edges (Dijkstra = OPT(v) only)

CASE 1: P uses at most i-1 edges thus OPT(i, v) = OPT(i-1, v)

* + 1. A path with i edges is simply a path with i-1 edges plus one edge. Call that one more edge w

CASE 2: P uses exactly i edges thus if (v, w) is the first edge then OPT uses (v, w) and then selects best w-t path using at most i-1 edges

cvw = cost of one edge

|Case 1 |Case 2 – (v,w) ϵ E means try every possible see diagram above|

|try every node adjacent to v as candidate for node w which requires many i – 1

|this second case catches the cases when you are using exactly i edges

Remark: by previous observation, if there are no negative cycles then OPT(n-1, v) = length of shortest v-t path because if there is a positive weight cycle it will simply not be taken, i.e. simple path dominates path w/cycle. Assuming no negative cycles, once we get to an i that is = to n – 1 we know we have found a shortest path, i.e. M[s, n-1] = the shortest path

**BELLMAN-FORD Method = draw table with edges from s->t across top and number of edges i 0 to n-1 down left side**

**Pseudocode for bottom-up: // Prof says come up w/memorized version for practice – see slides 12 and 16**

**Shortest-Path(G, t) { // bottom-up = recurses to base case (leaf) then works its way back up (tree)**

**foreach node v ϵ V**

**M[0, v] <- ∞**

**M[0, t] <- 0 // table M = space complexity n2 = n rows and n columns = can be large**

**for i = 1 to n-1 // look at sub-problems in order of increasing complexity = increasing # edges**

**foreach node v ϵ V // once # of edges fixed go through every node of the graph**

**M[i, v] <- M[i-1, v] // Case 1 – n2 iterations recall m = n – 1 = number of edges**

**foreach edge (v, w) ϵ E // not necessarily following the same order as recurrence relation**

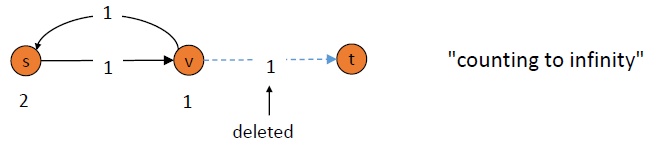
**M[i, v] <- min {M[i, v], M[i-1, w] + cvw } // can Case 2 improve on what we had before (ellvw = cvw)**

**} // may not occur in an organized order – m \* n iterations**

**// M[s, n-1] = the shortest path**

Bellman-Ford Runs in Θ(mn) or closer to Θ(n2) as graph => complete– if no save sub-problems runs Θ(n3)

Counting to Infinite: when v loses the v – t edge its asks its neighbors for help trying to get to t and it finds that the routing table of s says that s knows a 2-hop route (now out of date) to t and gets into a loop where based on s having a 2-hop route v mistakenly tells s it has a route with 3 hops and s updates its routing table and an endless cycle begins – protocols have mechanism to ID this issue and abort



Detecting Negative Cycles: a B-F shortest path has n-1 edges. If you one run more iterations of B-F and get a lower value with a path with n edges instead of n-1 edges there MUST be a negative cycle. This can be shown in O(mn) by adding a new sink node to that connects to every node with a 0-cost edge and if OPT(n, v) = OPT(n – 1, v) then no negative edges (this is faster than running B\_F for n iterations each time with a different destination node)

1) iff OPT(n, v) = OPT(n – 1, v) for all v, then there are no negative cycles on paths to node t. This means that if you are done running Bellman-Ford using n – 1 edges, and you run one more iteration so that you are now up to n edges, if the values do not change then there is no negative cycle – proof is via Bellman-Ford; 2) iff OPT(n, v) < OPT(n – 1, v) for some node v, then (any) shortest path from v to t contains cycle W. Moreover W has a negative costs - Proof: a) Since OPT(n, v) < OPT(n – 1, v) we know P has exactly n edges; b) By pigeonhole principle P must contain a directed cycle W; c) Deleted W yields a v-t path with < n edges => W has negative cost

MASTER METHOD: k = power on n, p = power on logn in f(n) if form T(n) = aT(n/b) + f(nk logpn)

**CASE 1:** if f(n) ≤ (nlogba-∊)then T(n) = Θ(nlogba) EX: T(n) = 9T(n/3) + f(n), nlogba = Θ(n2) so f(n) = *O*(n2-∊) so T(n) = Θ(n2).

**Example:** T(n) = 4T(n/2) + n here log24 = 2 which is larger than the exponent on n1 so **Case 1** = Θ(nlog24) = Θ(n2)

**CASE 2**: if logba = k then: **a**) if p > -1 then Θ(nk logp+1n); **b**) if p = -1 then Θ([nk or nlogba] logn); **c**) if p < -1 then Θ(nk log n).

**Example:** T(n) = 4T(n/2) + n2 here log24 = 2 which is = to the exponent on n2 so **Case 2** = Θ(nlogba log n) = Θ(n2 log n)

**CASE 3**: if logba < k or the traditional way to write it is if f(n) = O(nlogba + ɛ) then: **a**) if p ≥ 0 then Θ(nk logpn); **b**) if p < 0 Θ(nk)

**Example:** T(n) = 4T(n/2) + n3 here log24 = 2 which is < the exponent on n3 so **Case 3** = Θ(n3)