**Divide and Conquer: Median and Order Statistics**

Lower median is the overall median

Medians occur at: i = floor[(n+1)/2] and

i = ceiling[(n+1)/2]

**StraightMaxMin(a, n, max, min) {**

1. Max = a[1]
2. Min = a[1]
3. For i = 2 to n do
4. If a[i] > max then
5. Max = a[i]
6. End if
7. If a[i] < min then Min = a[i]
8. End if
9. End for

}

Time Complexity: best: n-1 comparisons

Worst: 2(n-1) Average: < 2(n-1)

**MaxMin: Divide and Conquer:**

MaxMin(i,j,max,min)

A[1:n] is a global array Parameters i and j are integers

1 <= i <= j <= n.

{

1. If (i = j) then max = min = a[i] small(P)
2. Else if (i = j - 1) then //small(P) # 2
3. If (a[i] < a[j]) then
4. Max = a[j]; min = a[i];
5. Else then
6. Max = a[i]; min = a[j];
7. End if
8. Else then
9. //If P not small find where to split P
10. Mid = floor[(i+j)/2]
11. MaxMin[i, mid, max, min]
12. MaxMin[mid + 1, j, max1, min1]
13. If (max < max1) then max = max1
14. If (min > min1) then min = min1
15. End if

}

Recurrence for MaxMin

{ T[ceil(n/2)] + T[floor(n/2)+2] if n > 2

T(n) = { 1 if n = 2

{ 0 if n = 1

Assume n = 2^k for k is + integer

T(n) = 2T(n/2) + 2

= 2(2T(n/4) + 2) + 2 = 4T(n/4) + 6

= 2^(k-1)T(2) + sum {i=1} to (k-1) of 2^i

T(n) = 2^(k-1) + sum {i=1} to (k-1) of 2^i

T(n) = 2^(k-1) + 2^k - 2

T(n) = n/2 + n-2 = 3n/2 - 2 <----- TIme Complexity

**Finding the n-th Smallest Number**

Partition(array, low, high) divides

**QuickSelect (a, n, k)** {

1. Low = 1, up = n+1
2. a[n+1] = inf
3. Repeat
4. J = Partition(a, low, up)
5. If k = j then
6. Return
7. Else if k < j then
8. Up = j
9. Else then Low = j + 1
10. End if
11. Until false

}

A = [65, 70, 75, 80, 85, 60, 55, 50, 45]

Partition(6,10) = [80,85,75,70, inf]

Partition(7,8) = [75,70] after [70, 75]

7th smallest is 70

Time Complexity: Worst: O(n^2)

Average: O(n)

Partition is O(p-m) where p and m are the current bounds

**Matrix Multiplication**

Given matrices A and B of size n x n, the product is C=A x B

Each element C(i,j) = sum {k=1} to n of [A(i,k) \* B(k,j)]

This takes theta(n^3) time cuz of 3 nested loops

**Divide and Conquer: 2x2 Matrix**

C = A x B = [C11,C12] [C21,C22]

C11 = A11B11 + A12B21  C12 = A11B12 + A12B22

C21 = A21B11 + A22B21 C22 = A21B12 + A22B22

{ b if n <= 2

T(n) = { 8T(n/2) + cn^2 if n > 2

T(n)= O(n^3)

**Strassen’s Algorithm:**

Introduces 7 matrix multiplications instead of 8

P = (A11 + A22)(B11 + B22) Q = (A21 + A22)B11

R = (B12 - B22)A11 S = (B21 - B11)A22 T = (A11 + A12)B22

U = (A21 - A11)(B11 + B12) U = (A12 - A22)(B21 + B22)

C11 = P+S-T+V C12 = R+T C21 = Q+S C22 = P+R-Q+U

{ b if n <= 2

T(n) = { 7T(n/2)+an^2 if n > 2

T(n) = O[n^(log{2}(7))] ~= O(n^2.81) a little faster than O(n^3)

Smaller matrices are worse since more additions and subtractions

**Greedy Algorithms**

MakeChange, SJF, Deadline, GreedyKnapsack, Huffman Code, Kruskal, Prim, and Dijkstra’s are all greedy algorithms

MakeChange(n) {

1. C = {100,25,10,5,1}; S = empty set(solution)
2. s = 0 sum of coins
3. While s != n do
4. X = Largest item in C such that s + x <= n
5. If x does not exist then
6. Return “no solution found”
7. End if
8. S = S U {x} //Add to solution set
9. s = s + x //Update sum
10. End while
11. Return S

}

**NOT ALWAYS OPTIMAL** if coins are 1, 3, 4, to find 6

Greedy would give 4+1 +1 (3 coins) Optimal: 3+3 (2)

Greedy(a,n) {

1. Solution = empty set
2. For i = 1 to n do: x = Select(a)
3. If Feasible(solution, x) then
4. Solution = Union(solution, x)
5. End if
6. End for
7. Return solution

}

Select: Selects an input from a and removes it

Feasible: Boolean that checks if x can add to solution

Union: Combines x to current solution, update objective function

**SJF:** Time Complexity: Sort: O(nlogn) remaining tasks: O(n)

Result: O(nlogn)

**Deadline** d(i) >= 0 profit p(i) > 0

(p(1), p(2), p(3), p(4)) = (100, 10, 15, 27) d(i) = (2,1,2,1)

Greedy: sort profits in non-increasing order. Pick the first that works

J = {1,4} is solution and optimal

J(1,2,3,4): d(2,1,2,1) p(60,30,40,80)

Solution (4,1)

**Fractional Knapsack**

Object i has weight w(i) and knapsack (bag) has a capacity m

If fraction x(i) of w(i), 0 <= x(i) <= 1 then profit of p(i)x(i)

n=3, m=20 p(1,2,3)=(25,24,15) w(1,2,3) = (18,15,10)

Profit to weight ratio: p(i)/w(i)

Of total weight optimal is (0, 1, 1/2)

sum(w(i)x(i)) = 20 sum(p(i)x(i)) = 31.5

GreedyKnapsack(p,w,m) {

1. Sort objects by p(i),w(i) in descending order
2. Profit = 0, capacity = m
3. For each object i in sorted order do
4. If w(i) <= capacity then
5. Take full object i
6. profit += p(i)
7. Capacity -= w(i)
8. Else then
9. Take fraction of i to fill capacity
10. Profit += p(i) \* (capacity/w(i))
11. Capacity = 0
12. Break
13. End if
14. End for

}

w(1,2,3) = (14,18,10) p(1,2,3) = (24,20,16)

p/w(1,2,3) = (1.71,1.11,1.6)

Order is 1,3,2

All item 1 and .6 of item 3 Total profit = 33.6

**Huffman Code:** Sort symbols by frequency, take the two lowest and combine lowest on left gets 0 and right gets 1. The total of the numbers is the new frequency. Repeat until all symbols are used

Time complexity: Construct priority queue: O(n)

Extraction and merging operations: O(nlogn)

Total: O(nlogn)

**Minimum Spanning Tree**

MakeSet(x) {

1. Set parent(x) = x
2. Set rank(x) = 0
3. return

}

Find(x) {

1. If x != parent(x) then
2. parent(x) = find(parent(x))
3. End if
4. Return parent(x)

}

Union(x,y) {

1. rootX = Find(x)
2. rootY = Find(y)
3. If rootX != rootY then
4. If rank(rootX) > rank(rootY) then
5. parent(rootY) = rootX
6. Else if rank(rootX) < rank(rootY) then
7. parent(rootX) = rootY
8. Else then parent(rootY) = rootX
9. rank(rootX) += 1
10. End if
11. endif

}

**Kruskal(G)** {

1. For each vertex n subset N do
2. MakeSet(n)
3. End for
4. Sort edges A in non-decreasing order by weight
5. T = empty set
6. For each edge (u,v) subset A sorted order do
7. If Find(u) != Find(v) then
8. Union(u,v)
9. T = T U u,v
10. End if
11. End for
12. Return T

} Time complexity: theta(aloga)

Since n-1 <= a <= n(n-1)/2 approximately theta(alogn)

If initially each vertex placed in its own set theta(n)

Basis: The empty set is promising because G is connected, and a solution must exist

Induction: Assume T is promising just before adding a new edge e = {u,v}

- The edges in T divide G into two or more connected components; u in one and v in another

- Let B be the set of nodes in the component containing u

- B is a strict subset of nodes of G

- T is promising, with no edge in T leaving B

- e is one a shortest edge leaving B, satisfying Lemma

- By Lemma, T U {e} is also promising

Conclusion: When the algorithm stops, T is a solution and is promising, hence optimal

**Prim (G, length) {**

1. T = empty set
2. Choose an arbitrary node u and initialize B = {u}
3. While B != N do
4. Find the edge {u,v} of minimum length such that u subset B and v subset (N but not B)
5. Add v to B and {u,v} to T
6. End while
7. Return T as the minimum spanning tree

}

Kruskal: (1,2),(2,3),(4,5),(5,6),(1,4),(7,1) in order when found

Prim: (1,2), (2,3), (1,4), (4,5), (4,7), (7,6) with start at 1

**Single Source Shortest Path**

**Dijkstra’s:**

PERM cannot be changed, TEMP can

Path is the length to get to a specific node

Pred is the previous node

Dijkstra (Graph G, Adjacency Matrix adj, Source s) {

1. Initialize path[] with inf for all nodes
2. Path[s] = 0
3. Initialize status[] with TEMP for all nodes
4. Initialize pred[] with NIL
5. While True do
6. Set current = node of min temp distance from s
7. If current == NIL then
8. Return {All reachable nodes process;}
9. End if
10. Set status[current] = PERM
11. For each node i in graph G do
12. If adj[current][i] != 0 and status[i] == TEMP do
13. If path[current] + adj[current][i] < path[i] do
14. path[i] = path[current] + adj[current][i]
15. Pred[i] = current
16. End if
17. End if
18. End for
19. End while

}

Base: Only source node s is marked PERM and path[s] = 0

- For all other nodes, path[i] is set to inf, as no paths are

defined yet, aligning with the algorithms initialization

Thus, both conditions hold at the start of the algorithm

Induction Hypothesis:

- Assume that: (a) For every node currently marked as

PERM, path[i] is the shortest path from s to i

- (b) For every node currently marked as TEMP, path[i]

represents the shortest special path from s to i

This is true just before a new node v is added to the set of

PERM nodes

Induction:

Objective: Show that when a new node v is marked as

PERM, path[v] is the shortest bath from s to v

- When v is chosen, it has the smallest path value among

all nodes marked as TEMP

- According to the induction hypothesis, path[v] is the

shortest special path to v

Contradiction Proof:

- Suppose there’s a shorter path to v passing through a

node x marked as TEMP

- Since all edge weights are nonnegative, this alternative

path isn’t shorter, as v was chosen as smallest path value

This contradiction confirms that path[v] is the shortest path

Conclusion:

When Dijskstra’s algorithm completes, All nodes are

marked as PERM meaning each path[i] is the shortest

path from so to i

This concludes the proof of Dijkstra’s algorithm is optimal

**Calculating Binomial Coefficients**

C(n,k) = C(n-1, k-1) + C(n-1, k)

Space Complexity: O(k) Time Complexity: Theta(nk)

**Dynamic Programming**

A method for solving complex problems by breaking them

down into simpler, smaller problems to avoid calculating the

same thing twice

MakeChangeDP, 0/1 Knapsack (2 types), Subset Sum, Equal Sum, Count of Subsets, LCS

c[i,j ] is the minimum number of coins need to make an amount j using

the first i denominations

n coins denoted by d(1), d(2) … d(n)

There’s an unlimited supply of coins

C[i,j] is filled row by row: Start from 0 up to the target amount N

- For each denomination, decide whether to include or exclude it in

the current amount calculation

**MakeChangeDP(Amount N, denominations d (**d(1), d(2) … d(n))**)** {

1. Make table c[i,j] with size n x (N + 1)
2. For i = 1 to n do
3. For j = 0 to N do
4. If i = 1 and j < d[i] then
5. C[i, j] = inf
6. Else if i = 1 and j >= d[i] then
7. C[i,j] = 1 + c[i, j - d[i]]
8. Else if j < d[i] then
9. C[i,j] = c[i - 1, j]
10. Else then C[i, j] = min(c[i-1,j], 1 + c[i, j - d[i]])
11. End if
12. End for
13. End for
14. Return c[n, N]

} Time Complexity: Theta(nN)

Combination:

* C[i,j] = min(c[i-1,j], 1 + c[i,j-d(i)])

{ inf if i = 1 and j < d(i)

C[i,j] = { 1 + c[i, j - d(i)] if i = 1 and j >= d(i)

{ c[i-1, j] if j < d(i)

{ min(c[i-1,j], 1 + c[i,j-d(i)]) otherwise

Filling the table: Make 8 using as little of 1,4, and 6 coins

Row i = 1: c[1,j] = [0,1,2,3,4,5,6,7,8]

Row i = 2 (Using coins of 1 and 4 Units):

- For j < 4: We can only use the 1-unit coin, c[2,j] = c[1,j]

- For j >= 4: 2 choices: Exclude 4-unit coin: Use c[1,j]

- Include 4-unit coin: Use 1 + c[2,j-4]

-Take the min of the 2 choices

- c[2, j] = [0,1,2,3,1,2,3,2,3]]

Row i = 3: c[3, j] = [0,1,2,3,1,2,1,2,2]

Final Answer is c[3,8] = 2. The minimum # of coins for 8 units is 2

**0/1 Knapsack**

Objects in this type are not broken into pieces

Let i denote the objects with + weight w(i), and + value v(i)

The weight capacity is W

Goal: Maximize the total value held in the knapsack

Let x(i) = 0 if we do not take object i and x(i) = 1 if we do

Make 2D memorization array V[0…n][0…W] preset to -1 v[i][w] is the maximum value achievable with the first i items and weight w

**Knapsack01(i, w)** {

1. If i = 0 or w = 0 then
2. Return 0 //Base case: no items left to be held
3. End if
4. If V[i][w] != -1 then
5. Return V[i][w] //Return already computed value
6. End if
7. If w(i) <= w then
8. V[i][w] = max(Knapsack01(i - 1, w), Knapsack(i - 1, w - w[i] + v[i]))
9. Else then
10. V[i][W] = Knapsack(i - 1, w)
11. End if
12. Return V[i][w]

} Call Knapsack01(n, W) to get the max value (returns V[n][W])

**Bottom-Up 0/1 Knapsack: Same init criteria as above**

**Knapsack01BU() {**

1. Make 2D array V[0…n][0…W]
2. For i = 0 to n do
3. V[i][0] = 0
4. End for
5. For w = 0 to W do
6. V[0][w] = 0
7. End for
8. For i = 1 to n do
9. For w = 1 to W do
10. If w[i] <= w then
11. V[i][w] = max(V[i - 1][w], V[i-1][w - w[i] + v[i]])
12. Else then V[i][w] = V[i - 1][w]
13. End if
14. End for
15. End for
16. Return V[n][W]

} Time Complexity: O(nW) Space Complexity: O(nW)

Example: weights = 1,2,5,6,7; values = 1,6,18,22,28 cap = 11

0 1 1 1 1 1 1 1 1 1 1 1

0 1 2 6 7 7 7 7 7 7 7 7

0 1 6 7 7 18 19 24 25 25 25 25

0 1 6 7 7 18 22 23 28 29 29 40

0 1 6 7 7 18 22 28 29 34 35 40

Total is 40 and consists of objects 3 and 4

**Subset Sum**

Given array: {3, 34, 4, 12, 5, 2}

Target sum: 9

I 0 1 2 3 4 5 6 7 8 9

0 elements: T F F F F F F F F F

3: T F F T F F F F F F

3, 34: T F F T F F F F F F

3, 34, 4: T F F T T F F T F F

3, 34, 4, 12: T F F T T F F T F F

3, 34, 4, 12, 5: T F F T T T F T T T

3, 34, 4, 12, 5, 2: T F T T T T T T T T

i = row #, a[i - 1] = Last num in array for row i], j is column #

To make this, DP[i][j] = DP[i - 1][j] or DP[i - 1][j - a[i - 1]] if j - a[i - 1] >= 0

To find elements used start at j = 0

- check if DP[i - 1][j] is false, do DP[i - 1][j - sum] instead

- sum = values that previously returned false

- Stop once at j = 0

- In the problem above, {4, 5} was the answer

**Equal Sum Partition**

Purpose: Find if the array can be split into 2 equal sums

If odd, always return false, otherwise Target = Total Sum / 2

This is now a subset sum problem

**Count of Subsets**

Same as Subset Sum except:

DP[i][j] = DP[i - 1][j] + DP[i - 1][j - a[i - 1]]

**Longest Common Subsequence**

LCS Top-Down

Requires X: string of length m, Y: string of length n

1. Initialize a memoization table L[0…m][0…n] set to -1

LCS\_TD(i,j) {

1. If i = 0 or j = 0 then
2. Return 0
3. End if
4. If L[i][j] != -1 then
5. Return L[i][j]
6. End if
7. If x[i] = y[i] then
8. L[i][j] = LCS(i - 1, j - 1) + 1
9. Else then L[i][j] = max(LCS(i - 1, j), LCS(i, j - 1))
10. End if
11. Return L[i][j]

} Compute the solution with LCS(m, n)

**Floyd-Warshall**

1. cost [1…n,1…n] //cost adjacency matrix of the graph
2. Initialize A[i,j] = cost [i,j] for all 1 <= i <= n
3. For k = 1 to n do For i = 1 to n do For j = 1 to n do
4. A[i,j] = min(A[i, j], A[i, k] + A[k, j])
5. End fors
6. Return A

**Homework**

1. Let x[1] to x[n] be n elements with = weight w[i] = 1/n for all i = 1 to n

Ans: sum {i = 1} to n of (w[i]) = sum of (1/n) = n\*1/n = 1

Each element has an equal weight of w median = sum of 1/n <= ½

The weighted median is at x[n/2] if n is even as well as odd

Therefore, since the w median is at x[n/2] and the median is at x[n/2]

2. How to compute the w median of n elements in O(nlogn) use sorting

find\_w\_median(x,w) {

1. sort(x) //O(nlogn)
2. Total = 0
3. For i = 1 to n
4. If (total > 1/2) then Return x
5. Else then Total += w
6. End if
7. End for

} Time complexity = O(nlogn) from sorting algorithm

3. 2D matrix and an integer target and returns if target is present

M = matrix.length n = matrix[i].length l <=m

All integers sort in ascending order for both rows and columns

searchMatrix(matrix,target) {

1. Init from above, i = 0, j = n - 1
2. For (int k = 0; k < m + n; k++)
3. If (matrix[i][j] == target) {return false}
4. If (matrix[i][j] < target) {i++; if (i > m - 1) {return false}}
5. If (matrix[i][j] > target) {j–; if (j < 0) {return false}}

}

4. Prove prim’s algorithm using induction

Base case: Empty set promisin bc G connected and solution must exist

Induction: Assume T promising just before adding a new edge

-Let E be set of edges connecting a vertex in B to 1 not in B

-Find edge e that has least length in E. Let u be vertex in B and v be vertex in N not in B of which u and v are connected by e

Add v to B and e to T

Since this iteration is adding e which is connecting a new vertex with least length e to T it should be promising

Conclusion: When the sltn stops, T is the sltn & is promisin henc optimal

5. Schedule and tot prof p[1,2,3,4] = 10,75,15,40 d[1,2,3,4] = 1,2,3,4

Slots = empty \* 4; sort(): 2,4,3,1; slots = {1,2,3,4} order of deadline

Total profit = 10 + 75 + 15 + 40 = 140

6. Why greedy approach of job sel in non-inc profits & feasible effective

- Selecting jobs with highest profit ensures max profit at each step

- Feasibility ensures each job is scheduled by its deadline (validity)

- If high-profit job can be scheduled later, it leaves earlier slots open

7. Denominations are (30, 24, 12, 6, 3, 1) Unlimited supply. Show why greedy makeChange does not always give optimal solution

-Use 48 coins: Iter 1: S = {30}, s = 30 Iter 2: S = {30, 12}, s = 42

Iter 3: S = {30, 12, 6}, s = 48 Optimal = {24, 24} not equivalent

- Greedy only considers local max and the coin system are not multiples

8. Time complexity of Kruskal if user supplies matrix of distances

-Matrix representation is an n x n matrix M where M[i][j] represents weight of the edge between vertex i & j or inf if no edge exists

- Steps to adapt: 1. We need a sorted list of edges, so iterate the list for all non infinities. Step 2. Sort edges by weights. Step 3. Apply Kruskals

9. Suppose the graph is not connected, what affect on Prim and Kruskal

-Kruskal: It would find the MSF which is multiple MSPs

-Prim: It would find the MSP of all vertices connected to starting node

Both do not explicitly detect a disconnected graph

10. Can this have multiple spanning trees. If so show in prim and krusk

Yes, in Kruskal’s if (5,3) chosen b4 (7,6) and vice versa, 2 different MSP

In prim’s, same thing. Kruskal is due to sort order and prim is due to overall order

11. Dijkstra’s with new node v added to set S of PERM nodes, let w be a node not in S. Can the new shortest path first pass v then another in S b4 getting to w

- Properties: When a node added to S, shortest path is finalized

-w maintains known dist d[w] which updates when shorter path found

-Path structure: Once v added to S, considers all outgoing edges from v to update dist d[w] for all nodes w not subset S

-If shortest path from source to w passes through v then d[w] will be updated as d[v] + weight(v, w) in this step

-If w were to reach another node in S (u) after passing v, imply a detour, as direct shortest path to u is already known and short than path throu v

-Why not? When v added to S, only updates TEMP nodes, and u is not a TEMP node it cannot make a detour through v first then to u

12. Why Dijkstras fails with negative weights

-It assumes that once a node v is added to PERM set, shortest path is already found. If negative weights after v is PERM, path could be shorte  
-A cycle with a larger negative weight will cause a shorter path

13. W V V/W ratio

1 10 20 2.0

2 20 30 1.5

3 30 66 2.2

4 40 40 1.0

5 50 60 1.2

Determines optimal total profit: 3 full, 1 full, 2 full, 5 80%

14. Huffman code of A: 0.35 B: 0.1 C: 0.2 D: 0.2 E: 0.15

BE -> B = 0, E = 1 CD -> C = 0, D = 1 BEA -> B = 00 E = 01 A = 1

CDBEA -> C = 00 D = 01 B = 100 E = 101 A = 11

15. Kruskal’s algorithm on left

{(A,B),(A,C),(A,D)}

16. Dijkstra’s on right

A -> A : 0 A -> B : 5 A->C : 2 A->D : 3 A->E : 3

17. Given workload = [3,1,4,2,2] devise an approach to minimize distance between sum of two team workloads

- Use equal sum partitions where |sum(S[1]) - sum(S[2])| is lowest

18. Program to find minimum # of edits to word u to get word v

DP[i][j] = { DP[i - 1][j - 1] + 1 if u[i - 1] = v[j - 1]

{ max(DP[i - 1][j], DP[i][j - 1]) otherwise

Deletions = len(u) - len(LCS) Insertions = len(v) - len(LCS)

Return deletions, insertions

Time complexity: O(m \* n)

19. Write recurrence relation for Longest Repeating Subsequence of S

DP[i][j] = { 0 if i = 0 or j = 0

{ DP[i - 1][j - 1] + 1 if S[i - 1] = S[j - 1] and i != j

{ max(DP[i - 1][j], DP[i][j - 1]) otherwise

Time Complexity: O(n^2) Space Complexity: O(n^2)

20. Write recurrence relation for max profit achievable of the following:

w(1,2) = (3,2) v(1,2) = (6,5) capacity = 8

DP[c] = max { DP[c - w1] + v1 if c >= w1

{ DP[c - w2] + v2 if c >= w2

{ 0 if c = 0

**Sum Rules (sum of {i = 1} to n) bold is the sum**

**c** = cn **i** = n(n+1)/2 **i^2** = n(n+1)(2n+1)/6 **i^3** = n^2(n+1)^2/4