Design and Analysis of Algorithms - Final Overview

**You should be able to sufficiently justify why any algorithm or data structure operation has the time complexity it does.**

# Fundamental Techniques

* Greedy Method (greedy choice, greedy-choice property)

The greedy method is a general algorithm design paradigm, built on the following elements:

– configurations: different choices, collections, or values to find

– objective function: a score assigned to configurations, which we want to either maximize or minimize

Idea: make a greedy choice (locally optimal) in hopes it will eventually lead to a globally optimal solution

It works best when applied to problems with the greedy-choice property – a globally-optimal solution can always be found by a series of local improvements from a starting configuration.

* Divide and Conquer (divide, recur, conquer, recurrence equations, Master Theorem)

Divide-and conquer is a general algorithm design paradigm:

-Divide: divide the input data in two or more disjoint subsets S1, S2, …

– Recur: solve the subproblems recursively

– Conquer: combine the solutions for S1, S2, …, into a solution for S

The base case for the recursion are subproblems of constant size

Master Method:

Many divide-and-conquer recurrence equations have the form:

T(n)= c if n<d

aT(n/b)+f(n) if n>=d

The Master Theorem:

1. if f(n) is O(nlog b a), then T(n) is big-theta(nlog b a)

2. if f(n) is big-theta(nlog b a logkn), then T(n) is big-theta(nlog b a logk+1 n)

3. if f(n) is big-omega(nlog b a), then T(n) is big-theta(f(n))

* Dynamic Programming (define subproblems, subproblem optimality, subproblem overlap, bottom-up, table)

Since subproblems overlap, we don’t use recursion.

Instead, we construct optimal subproblems “bottom-up. ”

Ni,i ’s are easy, so start with them

Then do problems of “length” 2,3,… subproblems, and so on.

Running time: O(n3)

* Algorithms/Problems include:
* Fractional knapsack (greedy)

Algorithm fractionalKnapsack(S, W)

Input: set S of items w/ benefit bi and weight wi; max. weight W

Output: amount xi of each item i to maximize benefit with weight at most W

for each item i in S

xi <- 0

vi <- bi / wi {value}

w <- 0 {total weight}

while w < W

remove item i with highest vi

xi <- min{wi , W - w}

w <- w + xi

* Task Scheduling (greedy)

Algorithm taskSchedule(T)

Input: set T of tasks w/ start time si and finish time fi

Output: non-conflicting schedule with minimum number of machines

m <- 0 {no. of machines}

while T is not empty

remove task i w/ smallest si

if there’s a machine j for i then

schedule i on machine j

else

m <- m + 1 schedule i on machine m

* Merge sort (divide and conquer)

Algorithm mergeSort(S, C)

Input sequence S with n elements, comparator C

Output sequence S sorted according to C

if S.size() > 1

(S1, S2) <- partition(S, n/2)

mergeSort(S1, C)

mergeSort(S2, C)

S <- merge(S1, S2)

* I won’t ask you about integer multiplication (divide and conquer)
* Matrix Chain Multiplication (dynamic programming)

Algorithm matrixChain(S):

Input: sequence S of n matrices to be multiplied

Output: number of operations in an optimal parenthesization of S

for i <- 0 to n - 1 do

Ni,i <- 0

for length <- 1 to n - 1 do

{ length= j - i is the length of the chain }

for i <- 0 to n – 1 - length do

j <- i + length

Ni,j <- positive infinity

for k <- i to j - 1 do

Ni,j <- min{Ni,j, Ni,k + Nk+1,j + di dk+1 dj+1}

record k that produces minimum Ni,j

return N0,n-1

* 0-1 Knapsack Problem (dynamic programming)

Algorithm 01Knapsack(S, W):

Input: set S of n items with benefit bi and weight wi; maximum weight W

Output: benefit of best subset of S with weight at most W

let A and B be arrays of length W + 1

for w <- 0 to W do

B[w] <- 0

for k <- 1 to n do

copy array B into array A

for w <- wk to W do

if A[w-wk] + bk > A[w] then

B[w] <- A[w-wk] + bk

return B[W]

# Graphs

* Definitions: graph- A graph is a pair (V, E), where

V is a set of nodes, called vertices

E is a collection of pairs of vertices, called edges

Vertices and edges are positions and store elements

directed, weighted, vertex degree, adjacent, incident, path, simple path, cycle, simple cycle, subgraph, spanning subgraph, connected graph, connected components, spanning trees, forest, biconnected graph, biconnected components, separation vertex and separation edge, handshaking lemma, bound on number of edges

* Data structures include: edge list, adjacency list, adjacency matrix •

Algorithms/Problems include:

* Graph traversals DFS & BFS (discovery & non-discovery edges)
* Variations/applications of DFS & BFS to find connected components, spanning forest, path, shortest path, cycle, if graph is connected, biconnected components

# Digraphs

* Definitions: digraph, in-degree, out-degree, directed path, reachability, strong connectivity, directed cycle, DAG, topological order, transitive closure
* Algorithms/Problems include:
* Digraph traversals DFS & BFS
* Test for strong connectivity
* Find transitive closure using Floyd-Warshall’s algorithm
* Find a topological order

# Weighted Graphs

* Definitions: weighted graph, single source shortest path problem formulation, shortest path tree, all pairs shortest path, minimum spanning tree
* Algorithms/Problems include:
* Single source shortest path using Dijkstra’s, Bellman-Ford, and shortest path in DAGs
* All pairs shortest path using Floyd-Warshall’s algorithm
* Find minimum spanning tree using Prim-Jarnik’s, Kruskal’s, and Baruvka’s algorithm

# Maximum Flow

* Definitions: edge capacity, flow network, source, sink, flow, cut, flow over cut, capacity of a cut, maximum flow problem formulation, flow augmentation and augmenting path
* Max-Flow and Min-Cut Theorem
* Algorithms/Problems include:
* Find maximum flow using Ford-Fulkerson’s and Edmonds-Karp algorithm