Final Exam Study Guide

Chapter 5. Divide and Conquer

- Describe the divide-and-conquer approach in algorithm design
  - Break up problems into several parts
  - Solve each part recursively
  - Combine solutions to parts in overall solution
- State the approach to deriving the running time complexity of a divdie-and-conquer algorithm
  - We solve a recurrence relation to determine the running time complexity.

Outline the algorithm and derive it's complexity

Closest pair of points

```
Closest-Pair(p1, ..., pn) {
    If n <= 3 then compare all pair-wise distances, and return the smallest distance.

    Compute separation line L such that half the points are on one side and half on the other side.

    d1 = Closest-Pair(left half)
    d2 = Closest-pair(right half)
    d = min(d1, d2)

    Delete all points farther than d from L

    Sort remaining points by y-coordinate.

    Scan the points in y-order and compare distance between each point and the next 11 neighbors.
        If any of these distances is less than d, update d.

    return d
}
```

Running Time Complexity: $O(nlogn)$

- Separation line $= O(nlogn)$
- Deleting points $= O(n)$
- Sorting $= O(nlogn)$
- Scanning $= O(n)$

Recurrence Relation:

- Closest-Pair(Left half) $= T(n/2)$
- Closest-Pair(Right hal) $= T(n/2)$

Combining the *Recurrence Relation* with the other running time complexities gives us

$$T(n) <= 2T(n/2) + O(nlogn)$$
$$T(1) = 0$$

Integer Multiplication

```
karatsuba(num1, num2){
    If (num1 < 10) or (num2 < 10)
    return num1*num2

    /* calculates the size of the numbers */
    m = max(size_base10(num1), size_base10(num2))
    m2 = m/2

    /* split the digit sequences about the middle */
    high1, low1 = split_at(num1, m2)
    high2, low2 = split_at(num2, m2)

    /* 3 calls made to numbers approximately half the size */
    z0 = karatsuba(low1,low2)
    z1 = karatsuba((low1+high1),(low2+high2))
    z2 = karatsuba(high1,high2)

    return (z2*10^(2*m2))+((z1-z2-z0)*10^(m2))+(z0)
```

Running Time Complexity: $O(nlog3) = O(n1.585)$

Recurrence Relation:

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 3T(n/2) + n & \text{otherwise} \end{cases}$$

Dynamic Programming

- State the simliarities and dissimiliarities among the *greedy* approach, *divide-and-conquer*, and the *dynamic programming* approaches to algorithm design.
  - *Dynamic Programming* is similar to to *Divide and Conquer* in that we break the problem into sub-problems, but in *Dynamic Programming* we compute the sub-problem exactly once and store the result for later building up of a solution, thus culling the search space to a reasonable runtime. *Greedy* would inheriently have to explore the whole exponential search space and is more the opposite of *Dynamic Programming*.
- Describe the *dynamic programming* approach in algorithm design.
  - *Dynamic Programming* uses a global *memoization* table to store the result of each sub-problem exactly once so that we can consult already computed answers instead of recalculating them, giving us an efficient result.
- Explain the concept of memoization.
  - *Memoization* is the process of storing the result of previous computations and returning the cached result instead of recalculating the same value.
- Contrast the top-down dynamic programming and bottom-up dynamic programming.
  - *Top-Down* dynamic programming involved using recursion for the breaking up into sub-problems
  - *Bottom-Up* dynamic programming "unwinds" the recursion and iterates over the problem space instead.

Build optimal substructure, implement bottom-up, and derive time complexity

Weighted Interval Scheduling

Optimal Substructure:

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ max\ \{v_j + OPT(p(j)), OPT(j-1)\} & \text{otherwise} \end{cases}$$

Bottom-Up Algorithm:

```
Input: n; s1, ..., sn; f1, ..., fn; v1, ..., vn
Global Array: M[0...n]

Sort jobs by finishing time so that f1 <= f2 <= ... <= fn

# p(j) is the largest index i (i < j) of a job compatible with job j
Compute p(1), p(2), ..., p(n)
Run Iterative-Compute_Opt

Iterative-Compute-Opt {
    M[0] = 0
    for j = 1 to n
        M[j] = max{ vj + M[p(j)], M[j-1] }
}
```

Running Time Complexity: $O(nlogn)$

- Sorting $= O(nlogn)$
- Calculating p(j) $= O(nlogn)$ (binary search)
- Iterative-Compute-Opt $= O(n)$

Segmented Least Squares

Optimal Substructure:

$$SSE = e(i,j) = \sum_{i}^{j} (y_i - ax_i - b)^2$$

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \min\limits_{1<=i<=j} \{e(i,j) + c + OPT(i-1)\} & \text{otherwise} \end{cases}$$

```
Input: n; p1, ..., pn; c
Global Array: M[0 ... n] (M[j]: the result of computing Opt(j))

Segmented-Least-Squares() {
    for j = 1 to n
        for i = 1 to j
            compute the least SSE e(i,j) for the segment that fits pi, ..., pj

    M[0] = 0
    for j = 1 to n
        M[j] = min(1 <= i <= j) {e(i,j) + c + M[i - 1]}

    return M[n]
}
```

Running Time Complexity: $O(n^3)$

- Computing SSE $= O(n^2)$ pairs $+ O(n)$ opeartions per pair $= O(n^3)$

- Filling in the memoization table = $O(n^2)$

Knapsack Algorithm

Optimal Substructure:

$$OPT(j) \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1, w) & \text{if } w_i > w \\ max\{OPT(i-1, w),\ v_i + OPT(i-1, w-w_i)\} & \text{otherwise} \end{cases}$$

```
Input: W; w1, ..., wn; v1, ... vn
Global Array: M[0..n][0..W]

for w = 0 to W
    M[0, w] = 0

for i = 1 to n
    for w = 0 to W
        if (wi > w)
            M[i,w] = M[i-1, w]
        else
            M[i,w] = max{M[i-1, w], vi + M[i-1, w-wi]}

return M[n, W]
```

Running Time Complexity: $\theta(n\ 2^b)$;

- Looping over n: $O(n)$
- Looping over W: $O(W) = O(2^b)$

Sequence Alignment

Optimal Substructure:

$$OPT(i,j) = \begin{cases} j\delta & \text{if } i = 0 \\ min \begin{cases} \\ \alpha_{x_i, y_j} + OPT(i-1, j-1) \\ \delta + OPT(i-1, j) & \text{otherwise} \\ \delta + OPT(i, j-1) \end{cases} \\ i\delta & \text{if } j = 0 \end{cases}$$

```
Input: m; n; x1, ..., xm; y1, ..., yn, delta, alpha

for i = 0 to m
    M[i, 0] = idelta
for j = 0 to n
    M[0, j] = jdelta

for j = 1 to n
    for i = 1 to m
        M[i, j] = min(  alpha[xi, yj] + M[i-1, j-1],
                        delta + M[i-1, j],
                        delta + M[i, j-1])
```

```
return M[m, n]
```

Runtime Complexity: $\theta(mn)$

- Looping over n: $O(n)$
- Looping over m: $O(m)$

Bellman-Ford

Optimal Substructure:

$$OPT(i, v) = \begin{cases} 0 & \text{if } i = 0 \text{ and } v = t \\ \infty & \text{if } i = 0 \text{ and } v \neq t \\ \min\{OPT(i-1, v), \min\limits_{(v,w)\epsilon E}\{c_{vw} + OPT(i-1, w)\}\} & \text{otherwise} \end{cases}$$

```
Global array M[0..n-1, V]  # V=set of n nodes

Shortest-Paths(G, s, t)
    foreach node v in V
        M[0, v] = infinity
    M[0, t] = 0

    for i = 1 to n-1
        foreach node v in V
            M[i, v] = M[i-1, v]
            foreach edge (v, w) in E
                M[i, v] = min { M[i, v], cvw + M[i-1, w]}

    return M[n-1, s]
```

Runtime Complexity: $\theta(mn)$

Network Flow

- Greedy Ford-Fulkerson
- Max Flow/Min Cut Theorem:
  - Max Flow: Amount of flow passing from the source to the sink
  - Min Cut: The values of sets S and T that minimize the capacity of the S-T cut
  - The maximum possible flow from s to t must flow through the minimum cut that separates the source and the sink
- Choosing “poor” augmenting paths can result in exponential running times. “good” paths can give polynomial runtimes

Show how to reduce it to a max flow/min cut problem and prove the correctness of the reduction

Bipartite Matching

Reduction:

Find a *Max Matching*

```
Input: Undirected bipartite graph G= (L union R, E)
Set M is called matching if each node appears in at most one edge in M that is:
    * each node in L is mapped to at most one node in R
    * each node in R is mapped to at most one node in L

# The reduction
Create a directed graph G' = (L union R union {s, t}, E')
    * Make all edges form L to R directed, and assign capacity 1 to each edge.
    * Add source s, and add edge of capacity 1 from s to each node in L.
    * add sink t, and add edges of capacity 1 from each node in R to t.
```

Proof $\leq$:

Cardinality of max matching in G = value of max flow in G'

- Given max matching M of cardinality k in G
- Consider the flow f in G' such that 1 unit is sent a long each of the k paths from s to to
- Then, f is an s-t flow of value k, which is certainly no more than the value f max flow

Proof $\geq$:

- Let f be a max flow of value k in G'
- By the integrality assumption, k is an integer. Then, since every c(e) = 1, each f(e) is either 0 or 1
- Consider M = set of edges from L to R in G' with $f(e) = 1$, Then, $|M| = k$
- Consider the cut $(\{s\} \bigcup L,\ R \bigcup \{t\})$, then each node in L and R appears in at most one edge in M(because the edge capacity = 1 for all edges)
- So, M is a matching of cardinality k, which is certainly no more than the cardinality of max matching

Circulation

Reduction:

Convert to *Max Flow*

```
Directed graph G = (V, E)
Edge capacities c(e), e in E
Node demands d(v), v in V
    * demand if d(v) > 0
    * supply if d(v) < 0
    * trans-flow if d(v) = 0
```

Finding a circulation must satisfy:

- for each $e \epsilon E$: $\quad 0 \leq f(e) \leq c(e)$
- for each $v \epsilon V$: $\quad f_{in}(v) - f_{out}(v) = d(v)$

```
# Convert to Max Flow
* Add new source s and sink t
* For each v with d(v) < 0, add edges(s, v) with capacity -d(v)
* For each v with d(v) > 0, add edge(v, t) with capacity d(v)

G has a circulation iff G' has a max flow of value D
```

Proof $\implies$

- If there is a circulation f in G, then
  - $D = \sum_{v:d(v)>0} d(v) = \sum_{v:d(v)<0} -d(v)$
  - That is, the total supply of D from all supply nodes finds its way to be consumed by all demands nodes of total demand D

- The value of f in G' is D if we assugn $-d(v)$ to each edge (s, v) and assign $d(v)$ to each edge (v, t)
- Note the value of f in G' cannot be greater than D because $cap(\{s\},\ V - \{s\}) = D$
- So, D is the max flow value

Proof $\Longleftarrow$

- Assume the value of a max flow f in G' equals D
- Then, it must be that in G' all edges E_s out s and all edges E_t into t are saturated
- If we remove E_s and E_t from G' and add demand $d(v) = -f_{in}(v)$ on the node v for each edge $s, v)$ in E_s and demand $d(v) = f_{out}(v)$ on the node v for each edge (v, t) in E_t then we have a circulation in G (whever ever node in G satisfies the demand condition)

Image Segmentation

Reduction:

```
Input: grid G=(V,E) where V = set of pixels and E = set of pairs of neighboring pixels,
    horizontally or vertically
* For each pixel i
    * ai >= 0 is the likelihood that pixel i is in the foreground
    * bi >= 0 is the likelihood that pixel i is in the background
* For each pair of neighboring pixels i and j
    * pij >= is the separation penalty for labeling one of i and j as foreground
        and the other one as background
```

Find a partition $(A_{foreground}, B_{background})$ that maximizes

$$\underbrace{\sum_{j \epsilon A} a_j + \sum_{i \epsilon B} b_i}_{\text{Reward for high likelihood values}} - \underbrace{\sum_{(i,j) \epsilon E \text{ and } |A \bigcap \{i,j\}|=1} p_{ij}}_{\text{Penalty for neighboring pixels on different sides}}$$

Maximizing this is equivalent to Minimizing this:

$$\sum_{j \epsilon B} a_j + \sum_{i \epsilon A} b_i + \sum_{(i,j) \epsilon E \text{ and } |A \bigcap \{i,j\}|=1} p_{ij}$$

Now constructc a flow network G' that will give this amount as the capacity of a min cut:

- Add source to correspond to foreground
- Add sink to correspond to background
- Add an edge from source to each noe j with capacity a_j
- Add an edge from each node i to sink with capacity b_i

Convert Undirected graph to directed graph using equal anti-parallel directed edges with edge capcities for both being p_{ij}

The capacity of (A, B) is exactly the quantity we want to minimize

NP and Computationl Intractibility

- Reducing one algorithm to another can allow us to determine if a program is polynomial if the problem we reduce it to has been show to be polynomial.
- NP, NP-Complete, NP-Hard
    - NP: a problem is in NP if we can prove in polynomial time that any 'yes' instance is correct
    - NP-Hard: A problem at least as hard as ny NP problem

  - NP-Complete: A problem that belongs to both NP and NP-Hard. (as hard as ny problem in NP but can be verified quickly)

For each of the following problems, show its NP-Completeness through reduction from the suggested problem known to be in NP-Complete

Satisfiability problme from circuit satisfiability problem

Approximiation Algorithms

- In order to solve an NP-Hard problem in polynomial time we must accept a suboptimal solution. Approximation algorithms ask if there is a range of suboptimal sollutions around the optimal that we can accept as "good enough"

For each of the following problems, design an approximation algorithm and prove the approximation ratio

Load Balancing

Algorithm:

```
Input: M identical machines; n jobs, job j has processing time tj
* Job j must run contiguously on one machine
* A machine can processa t most on ejob at a time

# Makespan = the maximum of the loads on the machines, that is L = max{i} Li

Assign n jobs to m machines to minimize the makespan

LPT-Greedy-Balance(m; n; t1, ..., tn)
    sort jobs so that t1 >= t2 >= .. >= tn

    for i = 1 to m
        Li = 0
        J(i) = null

    for j = 1 to n
        i = argmin(k) Lk
        J(i) = J(i) union {j}
        Li = Li to tj
```

Runtime Complexity: $O(nlogn)$

Approximation Ratio Proof:

If there are more than m jobs, $L^* \geq 2t_{m+1}$

- Consider the firsts $m+1$ jobs $t_1, ..., t_{m+1}$
- Since the t_i's are in descending order, each takes at least t_{m+1} time
- There are $m+1$ jobs and m machines, so at least one machine gets two jobs, this machine will have $L \geq 2t_{m+1}$

LPT-Greedy-Balance is a 3/2 approximation algorithm

For a bottleneck machine $L_i = \underbrace{(L_i - t_p)}_{\leq L^*} + \underbrace{t_p}_{\leq t_{m+1} \leq \frac{1}{2} L^*} \leq \frac{3}{2} L^*$

Minimum weighted vertex cover

Algorithm:

Pricing Scheme. Each edge must be covered by some vertex, So we price the edges as follows:

- Each edge e pays price $p_e \geq 0$ in order to be covered by a vertex
- Each vertex i charges the price of as much as w_i in total to cover edges

```
Minimum-Weighted-Vertex-Cover-Approx(G, w) {
    foreach e in E
        pe = 0

    while (There exists an edge (i,j)) such that neither i nor j is tight)
        Select such an edge e
        Increase pe to the max possible without violating the fiarness
            (i or j becomes tight as a result)

    S = set of all tight nodes
    return S
}
```

Approximation Ratio Proof:

- At least one new vertex becomes tight after each iteration of the while loop
- Let S be the set of all tight vertexes up on termination of the algorithm. Then, S is a vertex cover
  - If some edge (i, j) were left uncovered, then neither i nor j$$ would be tight, but then while loop would not have terminate
- Let S^* be an optimal vertex cover then $w(S) \leq 2 * w(S^*)$

$$w(s) = \sum_{i \epsilon s} w_i = \sum_{i \epsilon S} \sum_{e=(i,j)} p_e \leq \sum_{i]epsilonV} \sum_{e=(i,j)} p_e = 2 * \sum_{e \epsilon E} p_e \leq 2w(S^*)$$