HKN CS/ECE 374 Midterm II Review

Nathaniel Bleier and Mahir Morshed

Outline

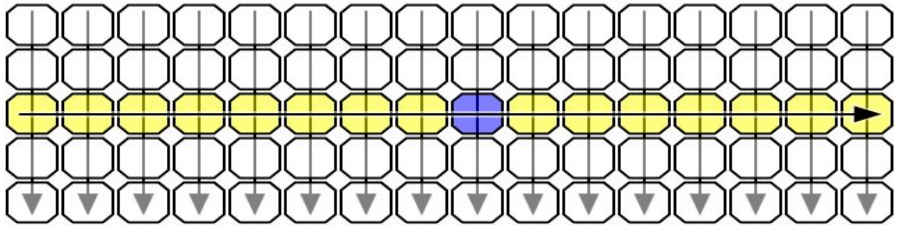
- Divide and Conquer Algorithms
  - Quick Sort (with Quick Select)
  - Karatsuba
  - (Binary Search, Mergesort if requested)
- Dynamic Programming
  - What is DP?
  - Outline of a dynamic programming solution
  - Example problem/solution
- Graphs
  - Review of definitions
  - Whatever-first-search
  - Topological ordering and topological sorting
  - Cycle Detection
- Shortest Path
  - Unweighted Graph (breadth-first search)
  - Djikstra's algorithm
  - Bellman-Ford
  - DAG's
- Potpourri
  - What is greedy?

Divide and Conquer - Quick [Search | Select]

- *Split your problem into smaller, complementary versions of the same problem!*

Input:	S	O	R	T	I	N	G	E	X	A	M	P	L
Choose a pivot:	S	O	R	T	I	N	G	E	X	A	M	P	L
Partition:	A	G	E	I	L	N	R	O	X	S	M	P	T
Recurse:	A	E	G	I	L	M	N	O	P	R	S	T	X

A quicksort example.



Visualizing the median of medians

```
MOM5SELECT(A[1..n], k):
  if n ≤ 25
      use brute force
  else
      m ← ⌈n/5⌉
      for i ← 1 to m
          M[i] ← MEDIANOFFIVE(A[5i−4..5i])   ⟨⟨Brute force!⟩⟩
      mom ← MOMSELECT(M[1..m], ⌊m/2⌋)        ⟨⟨Recursion!⟩⟩

      r ← PARTITION(A[1..n], mom)

      if k < r
          return MOMSELECT(A[1..r−1], k)     ⟨⟨Recursion!⟩⟩
      else if k > r
          return MOMSELECT(A[r+1..n], k−r)   ⟨⟨Recursion!⟩⟩
      else
          return mom
```

Divide and Conquer

- Karatsuba (Карацуба, not からつば): O(n^1.585) multiplication algorithm
  - Turns 4 multiplies and 1 add into 3 multiplies and 3 adds (cheaper to add)
- Binary searches
  - Should be familiar from a data structures course!
  - In practice, problems based on this require reconsideration of the bounds of the search space on each level of recursion
- Mergesort

Given an *m* row by *n* column array *A* in which, for all indices (h,k), *A*(h,k) >= *A*(j,i) where j < h *or* k < i *or* both, describe and analyze an algorithm that determines whether a value is present in *A*.

Dynamic Programming

- A problem has an *optimal substructure* if an optimal solution can be obtained from the optimal solutions of its sub-problems.
  - Shortest path from a start node to an end node in a graph
  - The basis for Bellman's equation and Principle of Optimality
- The subproblems of a problem are *overlapping* if, in solving the problem, the solutions to the subproblems are used more than once.
  - Fibonacci equation
  - This suggests the use of memoization

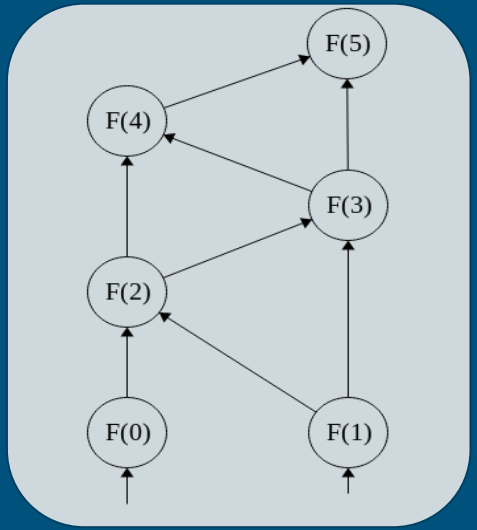
Dynamic Programming

Suppose a problem has both an optimal substructure and overlapping subproblems.

The natural approach to such a problem would be to recursively (iteratively) solve subproblems (smallest to largest) while memor?izing the results. This is called dynamic programming.

Dynamic Programming - Natural Structure

Since a dynamic programming problem has some number of initial smallest/base subproblems, and each subproblem must be solved before its parent problem can be solved, a dynamic programming problem has the structure of a directed acyclic graph.



Dynamic Programming - Efficient Structure

Due to the design of computers which execute dynamic programming algorithms (caches, TLBs), these algorithms run much faster (actual time) on data structures which are contained in contiguous memory than on node-pointer based data structures:

[F(0), F(1), F(2), F(3), F(4), F(5)]

Dynamic Programming - What Right Looks Like

1. **An English description of the underlying recurrence**
2. The details of that recurrence (as an equation)
3. The top level function call which will recursively solve the problem
4. The memoization data structure (typically as an *n*-dimensional array, or as a DAG)
5. An evaluation order (of the cells in the array or nodes in the DAG)
6. Runtime analysis

(See how pseudocode is absent from this list?)

Dynamic Programming - Maxmin Array Partition

Suppose we want to split an array of integers A[1..n] into *k* contiguous intervals that partition the sum of values in A as evenly as possible. Specifically, define the *quality* of such a partition as the minimum over all *k* intervals of the sum of the values in that interval; our goal is to maximize quality.

For example, given

A = [1, 6, -1, 8, 0, 3, 3, 9, 8, 8, 7, 4, 9, 8, 9, 4, 8, 4, 8, 2] and *k* = 3,

the max quality is 35 due to the optimal partition:

[1, 6, -1, 8, 0, 3, 3, 9, 8 | 8, 7, 4, 9, 8 | 9, 4, 8, 4, 8 2]

which has sums of 37, 36, and 35 respectively.

Dynamic Programming - Maxmin Array Partition

1. English Description:

   We define the function *Q* as follows:

   Let *Q(i, j)* := The max quality of the suffix A[j..n] using *i* sets in the partition.

Dynamic Programming - Maxmin Array Partition

2. The details of the recurrence:

Let

$$F(i,j) := \sum_{k=i}^{j} A[k]$$

Then

$$Q(i,j) = \begin{cases} -\infty & j > n \\ F(i,j) & i = 1 \\ \max_{j \le p \le n} \min\{F(j,p), Q(p+1, i-1)\} & \text{else} \end{cases}$$

3. Top level function call: *Q(k, 1)*.

Dynamic Programming - Maxmin Array Partition

4. The memoization data structure:

    Q is memoized as an array of *k* rows with *n* columns

5. An evaluation order:

    As pseudocode:

```
For col from n to 1:
    For row from 1 to k:
        Evaluate Q(row, col)
```

Dynamic Programming - Maxmin Array Partition

6. Run-time analysis

Memoizing *F* is O(*n^2*).

The *Q* array has *nk* entries, each of which looks up O(*n*) previous table entries. Thus filling out *Q* is O(*n^2k)*.

The overall runtime is O(*n^2k + n^2*) = O(*n^2k*).

Graphs - Terminology

- Sources
- Sinks
- Walks
- Paths
- Negative cycles
- Strongly connected components
- N-coloring
- Hamiltonian
- Ordering

Graphs

- Whatever (depth, breadth, best)-first search - O(|V|+|E|)
  - Should be familiar from a data structures course!
  - Stack -> depth, Queue -> breadth, Priority Queue -> best
- Dijkstra's algorithm - O(|E|+|V| log |V|)
  - Also should be familiar from a data structures course!
  - greedy, but no need to prove the correctness of its cost function
- Bellman-Ford-... algorithm - O(|E||V|)
  - Shortest-path algorithm relying on single source--uses DP
- Floyd-Warshall - O(|V|^3)
  - Shortest-path algorithm relying on producing all-pairs comparisons

Directed Acyclic Graphs

- Let [v1, v2, ... , vN] be an ordering of a graph's nodes such that for all $i < j$, there does not exist an edge vj -> vi. This ordering is called a *topological ordering*.
- A graph is a DAG if and only if it has a topological ordering.
- A DAG may have up to $|V|!$ topological orderings (when does this occur?)

Directed Acyclic Graphs

- Topological sort - O(|V|+|E|)
  - Not necessarily a unique ordering for a given DAG!
    - Consider the graph (a→b, c→d)
- Tarjan's algorithm - also O(|V|+|E|)
  - Find the DAG's SCCs
- Kosaraju-Sharir algorithm - also O(|V|+|E|)
  - Simpler conceptually than Tarjan's but not as practically efficient

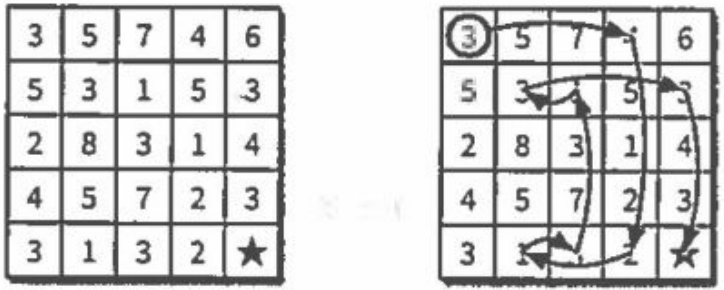
Graph Transformations

- How you transform a problem can determine the feasibility of approaching it
  - Add/delete edges?
  - Add/delete vertices?
  - Split undirected edge into two directed edges?
- (similar mindset to the transformation of DFAs/NFAs)

Describe and analyze an efficient algorithm to determine whether a given undirected graph is bipartite.

A ***number maze*** is an $n \times n$ grid of positive integers. A token starts in the upper left corner; your goal is to move the token to the lower-right corner. On each turn, you are allowed to move the token up, down, left, or right; the distance you may move the token is determined by the number on its current square. For example, if the token is on a square labeled 3, then you may move the token three steps up, three steps down, three steps left, or three steps right. However, you are never allowed to move the token off the edge of the board.

Describe and analyze an efficient algorithm that either returns the minimum number of moves required to solve a given number maze, or correctly reports that the maze has no solution.



A 5 × 5 number maze that can be solved in eight moves.

Describe and analyze an algorithm to find a path in an undirected graph with exclusively positive edge weights from vertex s to vertex t with minimum total weight.

Describe and analyze an algorithm to compute the shortest path in a directed graph with non-negative edge weights, built from a binary tree but adding edges from all leaves back to the root, that would be faster than Dijkstra's algorithm.

You are given a DAG G with n vertices and m edges (with $m >= n$), a parameter k, and a flag $r(v)$ which returns 1 if v is a ‘router’ and 0 otherwise. A ‘safe’ vertex can reach at least k distinct ‘routers’--note that k is small compared to n.

Describe an algorithm that computes the ‘safe’ vertices of G.

Greedy Algorithms

- Unless otherwise stated in the problem, do not use these *unless you are able to prove that the heuristic you're using is correct!*
- In Dynamic Programming, the algorithm itself enforces correctness. In a greedy algorithm, correctness is only guaranteed for certain subproblems, and correctness *must be proved*.