CS2223 in Four Pages*

1. Algorithmic Strategies. We study five main algorithmic strategies. These are ways to organize a computation that can be used across a variety of problem areas.

Breadth-First Traversal Given a starting point and some graph structure, we explore nodes in order of their distance from the starting point (counting by hops). The resulting data structure summarizes the distances, for nodes reachable from the starting point.

Depth-First Traversal Given a graph structure, we explore all nodes accessible from a given node before finishing with that node. Resulting data structure has start and finish "times." Key property is the *parenthesis property*: If node n_2 is discovered after n_1, but before n_1 is finished, then n_2 is finished before n_1: $d(n_1) < d(n_2) < f(n_1)$ implies $d(n_1) < d(n_2) < f(n_2) < f(n_1)$.

Greedy Algorithms In a greedy algorithm, an optimal solution to a problem is built from optimal solutions to its independent subproblems. After choosing an ordering of the subproblems, we simply choose the best permissible payoff for each subproblem as we encounter it.

Dynamic Programming To construct an optimal solution to a problem, we combine two or more optimal solutions to (possibly overlapping) subproblems. The main task is to choose an *operator* describing the optimal solution as a function of the optimal solutions to subproblems. This operator is a "knowledge extension operator," since it tells us how to extend our knowledge of how to solve some subproblems to learn how to solve others. Dynamic programming stores the successive optimal solutions into a *memo table*, from which the operator efficiently constructs later optimal values.

*Joshua Guttman, FL 137, `mailto:guttman@wpi.edu`. Include [cs2223] in the subject field. Version of October 25, 2012.

Divide and Conquer Given a large data object, we break it into smaller pieces and recursively solve the same problem on one or all of the pieces. We then combine the solutions for the pieces.

2. Problem Areas. These main areas give us examples:

Sorting *Insertion sort* and *bubble sort* are very simple. Their asymptotic complexity of $O(n^2)$ is worse than merge sort, but they are often quite useful for arrays that are small or already almost sorted.

Merge sort and *quicksort* are divide-and-conquer algorithms for sorting arrays. Merge sort has the better worst-case asymptotic complexity of $O(n \log n)$. However, quicksort at $O(n^2)$ is generally somewhat faster for most kinds of data, unless the data is already almost sorted (or in reverse order). *Heapsort* is asymptotically as good as merge sort, and is the simplest algorithm using a *priority queue*.

Graphs Breadth-first search finds *connected components* in an undirected graph, can check if it is *bipartite*. Depth-first search finds *cycles* and the *connected components* in a directed graph.

Dijkstra's algorithm finds shortest paths greedily from a given starting point in a graph with weighted edges. *Bellman-Ford* is a dynamic programming algorithm. It uses shortest paths with at most k edges to produce shortest paths with up to $k + 1$ edges. Unlike Dijkstra's algorithm, Bellman-Ford works when edges may have negative weights.

Prim's algorithm and *Kruskal's* algorithm are two greedy algorithms for finding a minimum spanning tree in a weighted graph.

Scheduling and Optimization Various small optimization problems illustrate greedy algorithms and dynamic programming.

Codes *Huffman codes* represent symbols by bitstrings of differing lengths depending on their frequencies. A classic greedy algorithm, it maintains a heap to examine the symbols in order of increasing frequency.

Preference and Choice The Gale-Shapley algorithm is a greedy algorithm that constructs matchings that are *stable*, meaning no pair of participants would agree on a change. Shapley just won the Nobel prize for economics.

Biology also provides some nice examples of dynamic programming.

3. Data Structures. We concentrate on a few data structures. A *table* is a data structure that associates *keys* or *tags* with *values*. An *array* k is a table in which the keys are integers, and form a sequence $1, \ldots, k$.

A *string* is really the same as an array, except that its values are characters or bytes that can be packed into contiguous memory.

In a *priority queue*, we can efficiently find and extract a minimal element relative to some ordering.[1] A priority queue is usually represented by a *heap*, namely a binary tree stored in an array, regarding entries $2i$ and $2i + 1$ as the left and right children of entry i. Many algorithms require that the ordering depend on information that will change as the algorithm runs; Prim's algorithm is an example.

We may briefly discuss *hash tables*, using chaining to resolve collisions, and *binary search trees*.

Graphs are represented using an array of *adjacency lists*, containing the edges from a node to all of its neighbors or successors. Graphs can also be represented by their adjacency matrices, a two-dimensional array defining the edges. A data structure is defined in terms of its:

Interface, a set of procedures that produce data of this kind, combine or modify them, and extract information from them;

Representation, the strategy for implementing the data structure in terms of more primitive objects; and

Invariants, the properties that are true of every instance of the data type, and are always preserved when data objects are modified or combined.

Taking priority queues as an example, its *interface* includes procedures to make an empty priority queue, to insert a new value, to extract that top value, and so on. It is *represented* as an array, with the left and right children of entry i stored at positions $2i$ and $2i + 1$. Its *invariant* is that the key of the parent is always less than or equal to the key of either child, using the heap's comparison function.

4. Evaluating Algorithms. We consider both experimental and analytic evaluations of algorithms.

Experimental Experimental evaluations include *measuring runtimes* for many algorithms and sizes of problems. In some cases, we get a clear indication of the algorithm's runtime across different problem sizes.

[1] It is sometimes regarded as a maximal element relative to the reverse ordering.

In other cases, it is more reliable to *count* the number of times that some *critical operation* is executed. In sorting algorithms and in algorithms using priority queues, we used the number of *comparisons* in the ordering of data objects as the events to count.

Analytical We also estimate the cost of an algorithm using big-O or Θ notation. When a function f is *asymptotically bounded* by a function g, to within some multiplicative constant c, we write $f \in O(g)$. When each of two functions f, g asymptotically bounds the other, to within multiplicative constants c_1, c_2, we write $f \in \Theta(g)$.

We classify algorithms by the function that describes *worst-case runtime* as a function of the input size x. An algorithm is *intractable* unless its worst-case runtime $f(x)$ is bounded by some *polynomial* $p(x)$, i.e. $f(x) \in O(p(x))$. The degree of the bounding polynomial is crucial, and should be very small. *Logarithms* give intermediate levels. Almost all the algorithms we discuss here fit in one of the classes:

$$\Theta(1) \quad \Theta(\log x) \quad \Theta(x) \quad \Theta(x \log x) \quad \Theta(x^2) \quad \Theta(x^2 \log x) \quad \Theta(x^3)$$

if there's a single measure x of the size of the problem instance. In some cases, we use two different factors for the input size, for instance, with graphs, the number of vertices V and the number of edges E.

When we can predict the structure of an algorithm's recursive calls, we can make a *recursion tree*. It represents the cost of running an algorithm in relation to all the costs of its recursive calls on subproblems. It also includes the cost of combining the answers to subproblems. The recursion tree suggests a *recurrence relation* or *summation* expression, which can be solved using a few tricks summarized in the *master theorem*.