CSC721 Spring 2013
Theory of Algorithms

Summary of Algorithm Strategies

Greedy A "greedy" algorithm makes a decision based on local knowledge (i.e. does not take a global view of all available information) that naively appears to be the best choice (i.e., greedy). A sequence of locally optimal choices leads to a globally correct solution. It's great when it works; we will see several examples where it fails miserably. Usually, a correctness proof is necessary to ensure the greedy approach actually yields a correct solution.

Divide and Conquer A "divide and conquer" algorithm breaks a problem into two or more sub-problems of the same type, solves the sub-problems, and then uses the solutions to the sub-problems to assemble a solution to the original problem. When the problem size is sufficiently, small, the solution is trivial or obvious. It is most often implemented recursively, using the trivial-sized problem as the base case.

Dynamic Programming A "dynamic programming" algorithm is similar to divide and conquer, but can be much more efficient, provided two conditions hold:

- the sub-problems overlap
- there is a straightforward way to index the sub-problems (i.e., uniquely identify a sub-problem by a small number of integers).

The key idea is that the algorithm keeps a table of the sub-problems which have already been solved. When that same sub-problem is encountered in the future, the solution is readily available in the table. There are two variations on this basic idea:

- The algorithm is implemented recursively. The table is consulted each time a solution to a sub-problem is needed. If the solution is available (in the table), that solution is used. Otherwise, a recursive function call is made to solve the sub-problem. Upon return from the recursive call, the returned solution is added to the table.
- Sometimes it is possible to re-order the sequence of operations such that the solution to each sub-problem will be placed in the table prior to needing that solution. This type of dynamic programming algorithm is most often implemented iteratively.

Backtracking A "backtracking" algorithm searches for a solution by making a sequence of local choices. But, unlike the "greedy" approach, there is no way of knowing whether the local decision will lead to a solution. Backtracking requires remembering the choices that have been made along the way, and systematically returning to (and reconsidering) earlier decisions. Backtracking algorithms are most often implemented recursively, but they may be implemented by explicitly managing their own stack.

Backtracking is computationally expensive, so its use is limited to problems in which the other approaches do not work. The basic idea is that at each recursive level, the algorithm makes a local decision, then recursively explores the consequences of that decision. If a solution is found, computation may be concluded. If a solution was not found, then at that point in the recursion, a different choice is made, and explored

recursively. When all choices are exhausted, the algorithm returns with an indication of failure from that point in the recursion.

Many backtracking algorithms may be thought of as a depth-first traversal of an appropriate decision tree.

Branch and Bound To describe "branch and bound", let us assume we are trying to minimize a function f over some set S. I.e., we seek a solution $\hat{x}$ such that:

$$\hat{x} = \text{argmin}_{x \in S} \, f(x) \tag{1}$$

A "branch and bound" algorithm starts with a set S of feasible solutions and divides it into two sets S_1 and S_2 such that $S = S_1 \cup S_2$. That's the "branching part. The approach requires a method to estimate a lower and an upper bound for the function values over sets S_1 and S_2. If the lower bound over S_1 exceeds the upper bound over S_2, then the branch represented by S_1 may be discarded (a.k.a, the *puning* step).

Naturally, the branch and bound approach can be generalized to split S into more than two sub-sets.