Basic Algorithms for Intractable Problems

Recursive Decomposition (Divide and Conquer)

Divide a problem into smaller problems of the exact same type, solving each recursively. Quicksort is a primary example.

Branch-and-Bound

Use this when a collection of decision variables must be examined. Visualize the problem as a tree in which each node represents a decision point, and the edges coming from the node represent the various choices being made. A *branch* is a choice point for one decision variable. At each branch, the *lower bound* of the solution is computed. If that bound is higher than other known solutions, then the branch can be pruned, or not visited.

The strength of this algorithm is in the ability to prune large portions of the decision tree. This depends on the quality of both the *selection function* for which branch to consider next, and the *bounding function* for evaluating that branch. The algorithm is efficient for average cases when the selection function makes an early choice of a branch which is close to optimal, and when the bounding function is sharp (providing exact bounds) and fast.

```
Branch-and-bound =def=
  current-best := anything;
  current-cost := infinite;
  decisions := s0;                  /* s0 is all possible unmade decisions
  while decisions =/= ( ) do
    select a decision  s;
    remove s from decisions;
    make a branch based on s, giving sequences s[i] from 1 to m;
    for i = 1 to m                  /* m=2 is a binary decision tree
      compute the lower bound b[i] of s[i];
      if b[i] >= current-cost
        then kill s[i];
        else if s[i] is a complete solution
          then current-best := s[i];
             current-cost := cost of s[i];
          else add s[i] to decisions.
```

Example: Selecting the shortest path from your house to the store. Each corner provides a decision point.

Dynamic Programming

Use this when the problem can be subdivided into subproblems, each of which allows a local optimal solution. Thus the best decision for each subproblem sums to the best decision for the entire problem. The key idea is to avoid solving the same problem multiple times by saving the solution to each subproblem. This is a bottom up approach. The Fibonacci recursion is a primary example: `F[n] = F[n-1] + F[n-2]`

Here is the dynamic programming algorithm applied to tree covering:

```
Tree-cover[T(V,E)] =def=
  Initialize cost of the internal vertices to -1;
  Initialize the cost of leaf vertices to 0;
  while some vertex has a negative weight do          /* bottom-up
    select a vertex v whose children have all nonnegative costs;
    M := set of all matching pattern trees at vertex v;
    L := set of leaves in T matching the leaves of pattern tree M;
    cost[v] := min[cost[M[i]] + sum-over-j[cost[L[j]]]].
```

Greedy Algorithm

This is a top-down approach. Use this when

1) the problem can be decomposed into local optimal decisions, and
2) making a particular locally best decision reduces the problem size.

Here is the greedy algorithm applied to task scheduling. We are given a set of tasks T, with each task having

a duration	`length[T[i]]`,
a release time when the task can begin	`release[T[i]]`, and
a deadline for completion	`deadline[T[i]]`

Find a schedule (an ordering of the tasks) which meets the release times and deadlines. The optimization problem is to find the minimal time schedule.

```
Greedy-scheduling[T] =def=
  i = 1;
  repeat
    while (Q := unscheduled tasks with release time < i)==0 do i++;
    if there is an unscheduled task p such that
      (i + length[p] > deadline[p])
      return FALSE;
    select Q[j] with smallest deadline;          /* greedy decision
    schedule Q[j] at time i;
    i := i + length[Q[j]];
  until all tasks are scheduled;
  return TRUE.
```

The greedy decision is to take the most constrained task when a task is selected. This may result in missing a possible solution, since selected tasks are not rescheduled by the algorithm (i.e. no search for the best solution).

E.g.: consider three tasks T = {a,b,c}, with length[1,2,3], release[1,1,3], deadline[4,inf,6].

Another example: what are the minimal number of coins that add to 63 cents? The greedy approach automatically uses the largest denomination coins first.

Backtracking

Sometimes there are no hints about the best choice. We simply must guess. Backtracking keeps track of that guess, and revokes it if necessary to solve the problem. Example is a complex game like chess, played by a non-expert.

There is a single generic search algorithm for tree and graph search:

```
Generic-search[current-state,search-list] =def=
  if current-state=nil or current-state=goal
    then done
    else search-list =
      special-merge[get-children[current-state],search-list];
      generic-search[first[search-list],rest[search-list]]
```

When the current-state is not the goal, generic-search puts the children of the current-state on the search-list and recurs. The `special-merge` function determines the type of search:

depth-first	add the new nodes on the front of the search-list
breadth-first	add the new nodes on the end of the search-list
iterative-deepening	like depth-first, but add new nodes to the front of the search-list only when their depth is not greater than the current cut-off, otherwise add new nodes to the end of the search-list
iterative-broadening	like breadth-first, but add new nodes to the front of the search-list only when their breadth is not greater than the current cut-off, otherwise add new nodes to the end of the search-list
hill-climbing	sort the new nodes by their estimated distance from the goal and add them to the front of the search-list
best-first	add the new nodes to the search-list, sort the entire list by estimated distance to the goal