## Greedy Algorithms

- Sometimes a problem is solved by making/testing a sequence of choices.
- In exhaustive search, we try all possibilities for each choice, backtracking if necessary.
- For greedy algorithms:
  - when presented a choice, select the possibility that "looks best" based on what has been done so far;
  - move on to the next choice and never backtrack.
- Often the choices are ordered (e.g. largest to smallest).

## Greedy Algorithms

- Generally very easy to come up with.
- Generally very fast.
- Very hard to prove correctness.
- As a result, many beginners try greedy algorithms on all “hard” problems and get “wrong answer”.

## Example: Coin Changing (Canadian)

- Denominations are \$2.00, \$1.00, \$0.25, \$0.10, \$0.05, \$0.01.
- Given an amount, achieve that amount using the fewest number of coins.
- Greedy algorithm: consider the denominations in decreasing order, use as many of a denomination as possible, and move on.
- Optimal for this set of denominations, not in general.

## 714: Copying Books (continued)

- Assuming that we are given the maximum number of pages a scribe can copy (as well as the pages of each book), can $k$ scribes copy all the books?
- Greedy algorithm: assign as many books to the first scribe as possible without exceeding the maximum.
- Continue until all books are assigned, or all scribes are used.
- Proof: exchange argument.

## Exchange argument

- We have some greedy solution which was obtained by making a sequence of choices $c_1, c_2, \ldots, c_n$.
- Suppose we have the "correct solution" which involves a possibly different sequence of choices $d_1, d_2, \ldots, d_n$.
- Suppose they are different. We may assume that $c_1 \neq d_1$ (otherwise, consider the first difference).
- Make some argument that by replacing $d_1$ by $c_1$ in the correct solution, we should still have a correct solution (or not make it worse for optimization problems).
- Recursively, we can make the correct solution the same as the greedy solution, so the greedy solution is correct (even if it may not be the same as the "correct" solution—there may be multiple solutions).

## 10656: Maximum Sum II

- Given a sequence of non-negative integers, find a subsequence whose summation is maximum.
- Basically just take all the integers—it does not help to not include one.
- Watch out for tie-breaking conditions.

## 10700: Camel Trading

- Given an arithmetic expression with no parentheses, find the maximum and minimum possible values when evaluated.
- If you want to find the minimum, multiply first.
- If you want to find the maximum, add first.
- Idea: $a * (b + c) = a * b + a * c \geq a * b + c$.
- The "choice" is when parsing an infix expression, whether to consider an operator high or low priority.
- This problem can be solved by modifying the "priority" table in `infix.cc`.

## 10026: Shoemaker's Problem

- $N$ jobs, each one takes $T_i$ days, and $S_i$ penalty for each day the $i$th job is delayed before starting.
- If $i$th job starts on day $d$, the penalty is $S_i \cdot d$.
- Want to do all jobs with minimum total penalty.
- If we choose a job $T_i$ first, then that job has no penalty. But all other jobs will have $T_i \cdot S_j$ added.
- Choose job $i$ over job $j$ if $T_i \cdot S_j < T_j \cdot S_i$.
- Sort jobs based on $T_i/S_i$, and choose the smallest job first.

## Dynamic Programming

- Sometimes if we are doing exhaustive search, we see the same subproblem being solved recursively over and over again.
- The parameters of this subproblem can be called a **state**.
- If there are not many states, but the same state is needed repeatedly, we may consider storing the result for the state so that it is not recomputed.
- Classic example: Fibonacci sequence.
- There is some recurrence that solves a problem from one or more subproblems. Don't forget about the base cases.

## Top-down vs. Bottom-up

- Top-down: use recursion. Check if a state has been computed first. If not, compute it and store the result. Use an array or a map to remember which state has been computed.
- Bottom-up: use iteration. Computes the base cases first, and order the computation of bigger states so that each computation only requires ones that have already been computed.
- Top-down computes only subproblems when necessary—saves space and sometimes faster.
- Bottom-up avoids overhead of recursion, useful if all table entries are needed.

## Example: Coin Changing (general)

- Set of denominations $d_1, \ldots, d_n$ (in cents).
- Total amount $T$.
- Let $f(t)$ be the minimum number of coins for amount $t$.
- $f(0) = 0$.
- $f(t) = 1 + \min\{f(t - d_i) : i = 1, \ldots, n, t \geq d_i\}$.
- Compute $f(T)$.
- Complexity: $O(nT)$ for time, $O(T)$ for space.

## Coin Changing (cont.)

- Can be extended: minimize weight of coins, count number of ways to form a sum (must order the coins).
- There is no known polynomial time algorithm (why is this not polynomial time)?
- State: the sum being formed. For counting, the sum and the "last" denomination used.

## Recovering Solutions

- Sometimes we want not only the optimal value (e.g. minimal weight of coins), but also the sequence of choices leading to that optimal value (which coin to use).
- When optimizing at each step of the recursion, also remember which choice leads to the optimal.
- Need to trace the choices **backward**.
- Sometimes we want to consider the problem backward so the trace can be done in the forward direction (e.g. lexicographical tie-breaking).

## Coin Changing (cont.)

- We had $f(t) = 1 + \min\{f\,(t - d_i) : i = 1, \ldots, n, t \geq d_i\}$.
- Also store $s(t) =$ the denomination $d_i$ that gives the minimum $f(t)$.
- Start from $s(T)$, then trace backward to the amount $T - s(T)$, and so on, until we get to 0.

## General Hints

- Start with exhaustive search (recursive version).
- Try to identify the states—what is different at each recursive call.
- Try to count the states—are some being computed repeatedly?

## Longest Increasing/Ascending Subsequence

- Given a sequence, remove as few of them as possible so that the remaining elements form an (strictly) increasing sequence.
- $O(n^2)$ algorithm: at each location, store the length of the longest subsequence ending there.
- $O(n \log n)$ algorithm: for each length, store the last element (smallest one if there is a tie) of the subsequence of that length. Scan from left to right.
- See `asc_subseq.cc`.

## Maximum Subvector Sum

- Given an array of integers, find a contiguous subarray that has the maximum total sum.
- $O(n)$ algorithm: scan from left to right. Keep track of the maximum sum found so far, as well as the maximum sum of a subarray that ends at the current location.
- See `vec_sum.cc`.

## Longest Common Subsequence

- Given two sequences, remove as few elements as possible from each so that the resulting two sequences are identical.
- $O(mn)$ algorithm: build a table $f$ so that $f(i,j)$ is the length of the longest common subsequence of `S1[0..i]` and `S2[0..j]`. The result is in $f(m,n)$.
- $f(i,j) = 1 + f(i-1, j-1)$ if $S1[i] == S2[j]$, and $f(i,j) = \max(f(i-1,j), f(i,j-1))$ otherwise.
- Base cases: $f(0,j) = f(i,0) = 0$.
- See `common_subseq.cc`.
- “Edit distance” is similar.

## Travelling Salesman (Advanced)

- Given a weighted graph with $n$ vertices, find the cheapest way to start from vertex 0, traverse all other vertices exactly once, and return.
- Brute force: $O(n \cdot n!)$.
- Dynamic programming: states are which vertices have not been visited (a subset) and which vertex you are currently at.
- Number of states: $O(n \cdot 2^n)$.
- Each state takes $O(n)$ operations to compute, total complexity $O(n^2 \cdot 2^n)$.