Implementing greedy algorithms

Our second programming exercise at this course asks you to write a solver subroutine that computes a low-cost set cover. (Click here to fast-forward to the problem statement and the submission dialog.)

One possible solution to the exercise is to prepare an implementation of the greedy set cover algorithm studied in the greedy algorithms lecture set.

Greedy algorithms are a subfamily of iterative algorithms for solving optimization problems by making the locally least-cost ("greedy") choice at every step of an iteration that builds a solution one choice at a time. For example, in the case of the set cover problem, one can build a set cover by choosing one set a time into a partial set cover, where the chosen set always covers the maximum possible number of points that yet remain to be covered.

Before we proceed with the formal problem statement, here are some general observations and guidance that you may find useful in implementing greedy algorithms:

- **Start with a correct implementation, then think about scalability.** As usual, rather than aim for two goals (correctness and scalability) at once, first prepare a correct implementation, and only then proceed to look at scalability.
- **The two modes of failure and working with unit tests.** When implementing a greedy algorithm, an implementation can fail to be correct in two possible ways: either

  > a. what is output is not a solution (in this exercise, not a set cover—there is at least one point among the n points $0, 1, \ldots, n-1$ that is not covered by the returned sets; or, there is in fact no solution and your implementation does not assert this correctly by giving an output with $k = 0$); or
  > b. what is output is a solution but it does not have a low-enough cost (in this exercise, there are too many sets in the cover; that is, the inequality $k \leq H_q \cdot \mathrm{OPT}$ does *not* hold).

  If your algorithm fails, the diagnostics associated with the unit test enable you to identify whether the error is due to (a) or (b). This in general requires one to study carefully both the input where the failure occurs as well as your source code—it may be a good idea to have your program print out what it is doing to the standard error stream. For example, in case of (b), are you actually making the greedy choice among all the available choices? Have you correctly identified all the available choices as well as their costs?
- **Finding and updating, fast**. When thinking about scalability, a greedy algorithm in most cases needs supporting data structures that enable one to quickly

  > 1. *find* the least-cost ("greedy") choice among the possible choices (for example, among the many sets that one can choose to extend a partial set cover), and
  > 2. *update* the consequences of a choice (for example, a set becomes less attractive to choose as more and more of its points get covered).

  For example, a *priority queue* that supports updates to priorities may be useful for keeping track of the costs of different choices as they evolve. Observe also that to be greedy it suffices to track only the distinct *costs* of the choices, one does not need to track all the possible *choices* with the priority queue.
- **Consequences of a choice are often local**. The effects of any one choice are in most cases *local* to the choice. For example, the choice of a set into a set cover affects only the sets that intersect the chosen set. For efficiency, it may be a good idea to to explicitly track the implications of each choice to avoid redundant work, such as scanning over all the sets. For example, one can track which sets contain which points, and *update this information as choices are made*, to avoid redundant work with subsequent choices. When a point is covered, one should be able to quickly access those sets, and *only* those sets, that contain the point. A *linked list* may be a useful data structure for maintaining and accessing such information fast under updates.
- **What would be a natural potential function?** As with iterative algorithms in general, to understand the scalability of a particular algorithm design and to produce a fast implementation, it is often useful to think in terms of a potential function. In this exercise, there are multiple natural parameters that one can look at, such as n (the size of the domain to be covered), m (the number of sets available for covering), q (the size of the largest set), and $p[m]$ (the sum of the sizes of the m sets). In particular, it takes $O(p[m])$ work just to read the input. What would it take from an implementation to achieve $O(p[m])$ worst-case scalability? Is this possible?
- **Greedy algorithms are not easy to parallelize**. As is the case with iterative algorithms in general, greedy algorithms are not particularly easy to parallelize due to the natural sequentiality of the successive greedy choices. A sequential implementation suffices for this exercise.
- **Beyond greedy algorithms**. In addition to being an iterative algorithm, a greedy algorithm is a special case of a local search algorithm where one always makes the least-cost choice among the possible local choices in a given state of the search. As such, it is often possible to turn an implementation of a greedy algorithm rather easily into a local search algorithm, which has similar needs for *enumerating* the available choices at a state (that is, enumerating all the possible choices, not just a least-cost one), and *updating* the state to reflect the consequences of a choice. If one also has the possibility to *undo* a choice, then an implementation of a *global search algorithm* such as backtrack search for exhaustively exploring a search space becomes available. However, such algorithms are not efficient in general. Yet such algorithms may be highly useful in finding exact and optimum solutions of small instances—for an example in the context of the exact cover problem, see here; this example also contains a data-structure observation that is useful to know.

Now we are ready for the problem statement and the submission dialog.