Problem Set 6

This problem set is due **in lecture** on **Wednesday, November 13.**

Reading: §33.3; Chapter 15; §16.1-16.3; Chapter 23

Both exercises and problems should be solved, but *only the problems* should be turned in. Exercises are intended to help you master the course material. Even though you should not turn in the exercise solutions, you are responsible for material covered by the exercises.

Mark the top of each sheet with your name, the course number, the problem number, your recitation section, the date, and the names of any students with whom you collaborated.

Each problem should be done on a separate sheet (or sheets) of three-hole punched paper.

You will often be called upon to "give an algorithm" to solve a certain problem. Your write-up should take the form of a short essay. A topic paragraph should summarize the problem you are solving and what your results are. The body of your essay should provide the following:

1. A description of the algorithm in English and, if helpful, pseudocode.
2. At least one worked example or diagram to show more precisely how your algorithm works.
3. A proof (or indication) of the correctness of the algorithm.
4. An analysis of the running time of the algorithm.

Remember, your goal is to communicate. Graders will be instructed to take off points for convoluted and obtuse descriptions.

Exercise 6-1. Do Exercise 33.3-3 on page 956 of CLRS.

Exercise 6-2. Do Exercise 15.4-4 on page 356 of CLRS.

Exercise 6-3. Do Exercise 15.5-3 on page 363 of CLRS.

Exercise 6-4. Do Exercise 16.2-5 on page 384 of CLRS.

Exercise 6-5. Do Exercise 16.3-4 on page 392 of CLRS.

Exercise 6-6. Do Exercise 23.1-3 on page 566 of CLRS.

Exercise 6-7. Do Exercise 23.2-3 on page 573 of CLRS.

Problem 6-1. In this problem, we will use hashing to modify the van Emde Boas data structure presented in lecture in order to reduce its space usage. The following material (through part **(a)**) is a review of the lecture on van Emde Boas structures; then we consider some changes.

Recall the problem statement: In the ***fixed-universe successor*** problem, a data structure must maintain a dynamic subset S of the universe $U = \{0, \ldots, u-1\}$. The data structure must support the operations of inserting elements into S, deleting elements from S, finding the successor (next element in S) from any element in U, and finding the predecessor (previous element in S) from any element in U.

Recall the outline of the van Emde Boas data structure: The universe $U = \{0, \ldots, u-1\}$ is represented by a ***widget*** of size u. Each widget W of size $|W|$ stores an array *sub*$[W]$ of $\sqrt{|W|}$ ***recursive subwidgets*** *sub*$[W][0]$, *sub*$[W][1], \ldots,$ *sub*$[W][\sqrt{|W|}-1]$ each of size $\sqrt{|W|}$. In addition, each widget W stores a ***summary widget*** *summary*$[W]$ of size $\sqrt{|W|}$, representing which subwidgets are nonempty. Each widget W also stores its minimum element *min*$[W]$ separately from all the subwidgets. Finally, each widget W maintains the value *max*$[W]$ of its maximum element.

For reference, the van Emde Boas algorithms for insertion and finding successors in $O(\lg\lg u)$ time are given as follows. For any widget W, and for any x in the universe of possible elements in W, define *high*(x) and *low*(x) to be nonnegative integers so that $x =$ *high*$(x)\sqrt{|W|}$ + *low*(x). Thus, *high*(x) and *low*(x) are both less than $\sqrt{|W|}$, and represent the high-order and low-order halves of the bits in the binary representation of x.

vEB-INSERT(x, W)
if $x <$ *min*$[W]$
  then exchange $x \leftrightarrow$ *min*$[W]$
if subwidget *sub*$[W][$*high*$(x)]$ is nonempty, that is, *min*$[$*sub*$[W][$*high*$(x)]] \neq$ NIL
  then vEB-INSERT(*low*(x), *sub*$[w][$*high*$(x)]$)
  else *min*$[$*sub*$[W][$*high*$(x)]] \leftarrow$ *low*(x)
    vEB-INSERT(*high*(x), *summary*$[W]$)
if $x >$ *max*$[W]$
  then *max*$[W] \leftarrow x$

vEB-SUCCESSOR(x, W)
if $x <$ *min*$[W]$
  then return *min*$[W]$
if *low*$(x) <$ *max*$[$*sub*$[W][$*high*$(x)]]$
  then $j \leftarrow$ vEB-SUCCESSOR(*low*(x), *sub*$[W][$*high*$(x)]$)
    return *high*$(x)\sqrt{|W|} + j$
  else $i \leftarrow$ vEB-SUCCESSOR(*high*(x), *summary*$[W]$)
    return $i\sqrt{|W|}$ + *min*$[$*sub*$[W][i]]$

(a) Argue that the van Emde Boas data structure uses $\Theta(u)$ space. (*Hint:* Derive a recurrence for the space $S(u)$ occupied by a widget of size u.)

Now consider the following modifications to the van Emde Boas data structure.

1. Empty widgets are represented by the value NIL instead of being explicitly represented by a recursive construction.
2. The structure *sub*$[W]$ containing the subwidgets

$$\textit{sub}[W][0], \textit{sub}[W][1], \ldots, \textit{sub}[W][\sqrt{|W|} - 1]$$

is stored as a dynamic hash table (as in Section 17.4 of CLRS) instead of an array. The key of a subwidget *sub*$[W][i]$ is i, so we can quickly find the ith subwidget *sub*$[W][i]$ by a single search in the hash table *sub*$[W]$.
3. As a consequence of the first two modifications, the hash table *sub*$[W]$ only stores the *nonempty* subwidgets. The NIL values of the empty subwidgets are not even stored in the hash table. Thus, the space occupied by the hash table *sub*$[W]$ is proportional to the number of nonempty subwidgets of W.

Whenever we insert an element into an empty (NIL) widget, we ***create*** a widget using the following procedure, which runs in $O(1)$ time:

```
CREATE-WIDGET(x)        ▷ Returns a new widget containing just the element x.
allocate a widget structure W
min[W] ← x
max[W] ← x
summary[W] ← NIL
sub[W] ← a new empty dynamic hash table
return W
```

In the next two problem parts, you will develop the insertion and successor operations for this modified van Emde Boas structure. It suffices to simply describe the necessary changes from the vEB-INSERT and vEB-SUCCESSOR operations detailed above. In any case, you should give special attention to the interaction with the hash table *sub*$[W]$.

(b) Give an efficient algorithm for inserting an element into the modified van Emde Boas structure, using CREATE-WIDGET as a subroutine.

(c) Give an efficient algorithm for finding the successor of an element in the modified van Emde Boas structure.

(d) Using known results, argue that the running time of your modified insertion and successor algorithms run in $O(\lg \lg u)$ expected time, under the assumption of simple uniform hashing.

(e) Prove that the space occupied by the modified data structure is $O(n)$. You may ignore the possibility of deletions, and assume that only insertions and successor operations are performed.

Problem 6-2. We are given a checkerboard with 4 rows and n columns, and a set of $2n$ pebbles; each pebble can be placed on exactly one square of the checkerboard. We define a ***placement*** to be a positioning of some or all of the pebbles on the board, such that no two pebbles are placed on horizontally or vertically adjacent squares. (Diagonal adjacencies are permitted.) On each square of the checkerboard is written an integer. The ***value*** of a placement is the sum of the integers in the squares that are covered by the pebbles of that placement.

(a) Determine the number of legal patterns that can occur in any column and describe these patterns.

We say that two patterns are ***compatible*** if they can be placed on adjacent columns to form a legal placement. Let us consider subproblems consisting of the first k columns $1 \leq k \leq n$. Each subproblem can be assigned a ***type***, which is the pattern occurring in the last column.

(b) Using the notions of compatibility and type, give an $O(n)$-time dynamic programming algorithm for computing a placement of maximum value.

Problem 6-3. Given a sequence of numbers $X = \langle x_1, x_2, \ldots, x_n \rangle$, a ***monotonically increasing subsequence*** is a subsequence $Z = \langle z_1, z_2, \ldots, z_k \rangle$ of X with the additional property that $z_1 \leq z_2 \leq \cdots \leq z_k$.

(a) Define $c[i]$ to be the length of a longest monotonically increasing subsequence that ends in x_i. Write a recurrence and initial conditions for $c[i]$, and prove them correct.

(b) Give an $O(n^2)$-time dynamic programming algorithm which, given a length-n sequence of numbers, finds a longest monotonically increasing subsequence.

(c) Improve your algorithm to work in time $O(n \lg n)$. *Hint:* Keep an augmented data structure of elements $x_1, \ldots, x_i$ that allows you to compute the recurrence for $c[i]$ faster.

Problem 6-4. You are an algorithms consultant for SipCar, which provides temporary vehicles to people at a variety of fixed locations across the Boston area. The day before a user needs a car, he goes to SipCar's website and chooses the time and location of the pick-up, and the time and location of the drop-off. SipCar needs to make sure that enough cars are at each location at the start of the day, so that there is always at least one car available for every pick-up. However, it is expensive to place cars (overnight) at the various locations, so minimizing the total number of cars in circulation is an important goal. SipCar wants an efficient algorithm to process the vehicle requests so as to minimize the total number of cars that must be initially placed at the locations.

Formally, the problem is the following: you are given m locations (numbered 1 through m) and n tuples (ℓ, t, ℓ', t') in some arbitrary order. Here ℓ and t are the location and time of the pick-up (respectively); ℓ' and t' are the location and time of the drop-off (respectively). When a user picks up a car from a location, the number of cars at that location decreases by 1; when she drops off a car to a location, the number of cars at that location increases by 1. Give an algorithm that outputs the number of cars that should initially be circulated to each location subject to two constraints: every location always has a non-negative number of cars, and the total number of cars in circulation is minimized. Make your algorithm as efficient as you can.