Arrays

The array (or vector of references, see http://www.standardml.org/Basis/array.html) is a very useful data structure, as one can access or update any element in constant time, provided its index is known. For example, if one has an array `a` of type `int array`, then one can use the expressions

\[
\text{Array.sub (a, 997)}
\]
\[
\text{Array.update (a, 3, x)}
\]

in order to access the value of element `a[997]` and update the value of element `a[3]` to the value of `x`, respectively. However, arrays require that the memory needed to store all the elements is allocated contiguously and at once, as well as (for static arrays) that the maximum array size is known in advance. A rather important consequence thereof is that if for some reason one uses only a few elements in a large array, then one wastes a lot of memory. For example, assume one declares an array of a million elements, but for some reason only the values at the indices 3 and 997 are used in a given run; the declaration

\[
\text{val a = Array.array (1000000, 5)}
\]

then allocates contiguous memory for an array `a` of one million integers (at indices 0 to 999,999) and initialises them all to the default value 5, but only a very tiny percentage of this memory is used, which is very wasteful.

Flexible Arrays

This is where the concept of flexible array comes in. A flexible array is like a normal array as far as how one uses it is concerned: one can access and update the element at a
given index as easily as in a normal array. Functionally, there is thus no difference, but computationally there are differences in resource consumption. The first difference is that if one needs an array of a million elements, say, but really uses only a small number thereof, then the flexible array requires a much smaller amount of memory. This flexibility comes at a price though: the second difference is that the access to an element of known index is not constant-time any more.

Representing Flexible Arrays

The size of a flexible array is not a defined concept. One can represent a flexible array as a list of chunks. Each chunk has a small fixed-size normal array of the same type of objects as the flexible array, as well as the starting index within the flexible array of the chunk. The polymorphic 'b flexArray datatype has the following definition:

```
datatype 'b flexArray = Flex of int * 'b * 'b chunk list ref
```

where the first tuple component (the integer) is the chunk size and the second component (of type 'b) is the default value for chunk elements when creating a new chunk. The polymorphic chunk type has the following definition:

```
type 'b chunk = int * 'b array
```

where the first tuple component (the integer) is the starting index of the second component (the array) within the flexible array. Chunks in the list of a flexible array f must be ordered increasingly by their starting indices, which must be integer multiples of the chunk size of f.

Example 1 A flexible array f with two chunks of s = 10 elements and with currently only the elements at indices 3 and 997 being used is depicted in Figure 1 (careful, the default value is not shown). We have only allocated memory for 20 elements of the flexible array although it seems to have 1,000 elements. If we need to store a value at index 6, then we first go to the initial chunk (which holds the elements at indices 0 to 9) and then store that value at index 6 in that chunk. If we need to update the value at index 997, then we first find the chunk with starting index ℓ such that ℓ ≤ 997 < ℓ + s and then store the new value at index 7 in the array of that chunk; in this case, we have such a chunk, namely the last one, with ℓ = 990. If we need to store a value at index 556, then we first create a new chunk for indices 550 to 559, link it in between the two existing chunks, and then store that value at index 6 in that new chunk.

Work To Be Done

Implement the following functions, making sure that they pass the training test cases (at http://www.it.uu.se/edu/course/homepage/pkd/ht10/labs/inlupp5-training.sml):

- `array (s, d)` returns the empty flexible array of integer chunk size s and default value d, assuming s > 0;
Figure 1: A flexible array with chunks of size $s = 10$.

- sub $(f, i)$ returns $f[i]$, which is possibly the default value of flexible array $f$, assuming $i \geq 0$;
- update $(f, i, x)$ destructively replaces $f[i]$ by $x$ in flexible array $f$, assuming $i \geq 0$, and returns $()$.

In a separate report in PDF format, give an explicit reasoning (including recurrences and their closed forms) establishing the average-case and worst-case runtime complexities of your functions.

**Grading**

Your solution is graded with 0 to 100 points in the following way:

a. If your solution was submitted by the deadline, your program loads under working version 110.72 of SML/NJ and passes all the training test cases, and your solution is deemed by us to be a serious attempt at implementing, commenting (under at least the coding convention), and analysing all the requested functions, then you get 20 points; otherwise (including when no solution was submitted by the submission deadline), you get a U grade for this assignment (even if an insufficiently commented program is actually correct).
b. Your program is run on an unspecified number $t$ of orthogonal grading test cases that satisfy all pre-conditions but also check boundary conditions. For each fully correct test result, you get $50/t$ points. We reserve the right to run these tests automatically, so be careful with function names and argument orders.

c. Your program is graded for style and comments (including function specifications, datatype representation conventions and invariants, as well as recursion variants), provided it does not fail on all the grading test cases. This covers 10 points.

d. Your complexity analysis is graded for correctness of results and explicitness of reasoning. This covers the remaining 20 points.

Have fun!