CSE326 – Data Structures and Algorithms
Winter 2004
Dry assignment #1 - Solutions

Problem 1.
The time complexity of bar is $T_{bar}(n) = \Theta(\log n)$. The first recursive call from bar(n) is bar(n/2). The argument of the next recursive call will be n/4, the following n/8, and in general the i-th recursive call will have argument $n/2^i$. Thus when $i=\Theta(\log n)$, the statement "if (k <= 1)" will be true and the recursion will stop. As each recursive call takes $\Theta(1)$ time, the total time is $\Theta(\log n)$.

$T_{foo}(n) = \Theta(\log n)$ as well. Each operation in foo() is constant-time, except the call to bar(), which will take time $\Theta(\log(n^3))= \Theta(3\log(n))= \Theta(\log n)$. Any of these answers is technically correct, but usually when using asymptotic notation we drop as many of the constants as we can.

Problem 2.
a) Yes. (An example is the BubbleSort algorithm in the case when the array is initially sorted.) Since all we know is the worst-case time complexity, the average-case or best-case complexities can be anything smaller than that worst-case boundary. Since O(n) is smaller than $O(n^2)$, the answer is positive.

b) Yes. If T(n) = O(n), then $T(n) = O(n^2)$ too. In this case, the bound $O(n^2)$ is an upper bound, but nothing in the definition of the problem indicates that this is a tight upper bound, so it is possible that a better (smaller) upper bound, such as O(n), exists.

c) Yes. (Again, an example is the BubbleSort algorithm.) The bound $\Theta(n^2)$ only concerns the worst case, but says nothing about the time complexity for other inputs, including for the best case. Since O(n) is better (smaller) than $O(n^2)$, it is a possible time complexity for some inputs (other than the worst case).

d) No. $T(n) = \Theta(n^2)$ implies that $T(n) = \Omega(n^2)$, which by definition leads to $T(n) \geq c_1.n^2$ for some constant $c_1>0$. This is in the worst case.
If we assume that the algorithm takes O(n) on all inputs, it will take O(n) in the worst-case too. Hence, by definition, T(n) = O(n) implies that $T(n) \leq c_2.n$ for some constant $c_2>0$. This is in the worst case too.
Combining the two results above, we obtain $c_1.n^2 \leq T(n) \leq c_2.n$. The inequality $c_1.n^2 \leq c_2.n$ cannot possibly hold for arbitrarily large values of n, regardless of the choice of the two positive constants c_1 and c_2. We have reached a contradiction. It stems from the incorrect assumption we made that the algorithm can take O(n) on all inputs.

Problem 3.
The algorithm works as follows. Since we need to cover the entire array a[] and are not allowed to use loops, the only alternative is to use recursion.

The base case of the recursion will be when there are two elements in the array (n=2) – in this case, the maximal sum will be the sum of these two elements (since there are no other sums to compare against). In the case of n>2 elements, we can recursively compute the sum of the first (n-1) of them and compare that to the sum of the last two elements. This will cover all possible sums of two consecutive elements in a[], and, as should be the case with recursion, it will indeed reduce the initial problem of size n to the same problem of size n-1. (Notice that since n>2 and n is integer, we can be sure that n-1 ≥ 2 and so we won't miss the base case while decreasing the size of the recursively examined part of a[].)

```
MaxPair (a[]: integer array; n: integer): integer
{
   candidateMax1: integer; /* candidate sum #1 to compare */
   candidateMax2: integer; /* candidate sum #2 to compare */

   candidateMax1 = a[n-1] + a[n-2];      /* the sum of the last two
elements */
   if (n>2)
   {
      candidateMax2 = MaxPair(a, n-1);      /* recursing over the first
*/
                                            /* (n-1) elements of the
array */

      if (candidateMax1 < candidateMax2)   /* If recursion yielded a
better */
         candidateMax1 = candidateMax2;    /* candidate sum, replace the
*/
                                           /* current maximal sum with
it.  */
   }

   /* In all cases when we reach this point (either when n=2, or when
the  */
   /* if-statement is completed), the maximal sum of two consecutive
*/
   /* elements in a[] will be stored in candidateMax1.  So we return
that. */
   return (candidateMax1);
}
```

Space complexity. `MaxPair()` has two local variables, so the space complexity of each recursive call is O(1). Since `MaxPair(n)` requires (n-2) recursive calls: `MaxPair(n-1)`, `MaxPair(n-2)`, …, `MaxPair(2)`, the total space complexity becomes (n-2) * O(1) = O(n).

Time complexity. Let T(n) denote the time complexity of `MaxPair()` on an input of size n. Since the computation of `MaxPair()` on an array of n elements is done by doing the same for an array of (n-1) elements (the recursive call), plus a few primitive operations, we can write:

T(n) = T(n-1) + c, for all n>2.
T(n) = b, for n=2.

Here b and c are constants, each representing a small number of primitive operations.
We can solve this recursive equation by simplifying the right hand side until we get to the base case n=2 and substitute b for T(2). Thus:

$T(n) = T(n-1) + c = (T(n-2) + c) + c = ((T(n-3) + c) + c) + c = \ldots = (\ldots(T(2) + c) + \ldots) + c =$

$= b + (n-2)*c = O(n).$

Problem 4.

We will use 0-based arrays. One observation we make from the start is that row i is identical to row (N-i), and column j is identical to column (N-j). This means that it suffices to concentrate on the upper left quadrant of the matrix where the rows and columns run between 0 and $\lceil N/2 \rceil - 1$. (The symbol $\lceil x \rceil$ denotes the smallest integer larger than or equal to x.)

The data structure we use is an array `FrameValue[]` where the index is the frame ID (between 0 to $\lceil N/2 \rceil - 1$, since there are $\lceil N/2 \rceil$ frames in an $N \times N$ frame matrix) and the value of `FrameValue[k]` is the value stored in all matrix cells belonging to the frame with ID equal to `k`. The space complexity of this data structure is clearly O(N), so this constraint is satisfied.

The operations in pseudocode depend on a helper function `getFrameID(i,j)` that returns the frame ID given row `i` and column `j` in the frame matrix. Their definition is as follows:

```
get (i: integer; j: integer): integer
{
   return ( FrameValue[ getFrameID(i,j) ] );
}

put (i: integer; j: integer; x: integer): void
{
   FrameValue[ getFrameID(i,j) ] = x;
}
```

Clearly, the time complexities are T(`get()`) = T(`put()`) = T(`getFrameID()`) + O(1), so everything depends on `getFrameID()`. Its definition is as follows:

```
getFrameID (i: integer; j: integer): integer
{
   i = min(i, N-1-i); /* pick a row that is up to ⌈N/2⌉ (as discussed
above)    */
   j = min(j, N-1-j); /* pick a column that is up to ⌈N/2⌉ (as discussed
above) */
   return ( min(i,j) );
}
```

The key observation in this problem is that, once we start looking only at the upper left quadrant (since the other quadrants are symmetric), below the main diagonal (that runs from upper left to lower right) of the matrix the frame ID is the same as the column number, while above the main diagonal it is the same as the row number. On the diagonal itself, both row and column are the same, and so the frame ID is equal to both.

For example, cell `(1,2)` in the matrix lies above the main diagonal, so we take its row number 1 – indeed that is the frame number. This cell currently holds the value 14, which will be stored in `FrameValue[1]`.

Since the time complexity of finding the minimum of two values is O(1) – it only involves a comparison, the time complexity of `getFrameID()`, and consequently that of `get()` and `put()` is also O(1), as desired.

Problem 5.

As a result of the execution of `rec_func(t)`, the linked list t will be reversed in the sense that the data elements stored in the list will be in reverse order as compared to the original list.
(Notice how difficult it may be to understand even a short and cleanly written function when there are absolutely no comments written on it, including no meaningful variable names or function names!)

The list is being reversed one element at a time – every recursive call to `rec_func1()` moves exactly one element from the list t to the list r (which starts out empty and eventually becomes the reversed list t). Therefore there are length(t) recursive calls being made.

Time complexity: Since each recursive call by itself takes O(1) time (for the assignments and the conditional), the time complexity is length(t) * O(1) = O(length(t)).
Space complexity: Since each recursive call introduces O(1) (a constant number of) new variables, the space complexity is length(t) * O(1) = O(length(t)).