CS106B
Spring 99-00

Handout #20
April 24, 2000

Practice Solutions

Review Session: Wednesday, April 26th 7-9pm 420-041 (basement of Pysch)
Midterm Exam: Friday, April 28th 2:15 - 4:15pm Kresge Aud (in Law School)
Alternate Exam: Thursday, April 27th 6-8pm 550-550A

We would be grateful if all of you could manage to attend one of the two scheduled times, but if this is an impossiblity, you must e-mail Yves (yves@cs) by 5:00P.M. today to schedule an alternate time. **Late requests will not be honored.**

Problem 1: Recursion and big-O

This problem can be solved by analyzing each function formally or by figuring out exactly what each one does. Although not always possible, the latter approach works okay in this case. `Enigma` computes the product of `n1` and `n2` by recursively summing `n1` copies of `n2`. `Mystery` uses `Enigma` to multiply `n` copies of the integer 2. Thus, the value returned by `Mystery(n)` is 2^n, so `Mystery(3)` returns 8. In figuring the complexity, note that `Mystery(n)` makes `n` calls to `Enigma`. Within `Mystery`, each call to `Enigma` requires a constant amount of work because the first argument is always 2. `Mystery`'s complexity is therefore proportional to `n` times a constant, which is *O*(*N*).

You can also analyze the functions by means of recurrence relations. First, consider the `Enigma` function, which has two parameters, n1 and n2.

$$T(n1, n2) = \{ 1 \quad \text{if } n1 = 0 \\ \quad 1 + T(n1-1, n2) \quad \text{otherwise} \}$$

You might immediately recognize this as linear in terms of n1, but in case you didn't, you can repeatedly substitute and expand to produce:

$$T(n1, n2) = 1 + [1 + T(n1-2, n2)]$$
$$T(n1, n2) = 1 + [1 + (1 + T(n1-3, n2))]$$
$$\ldots$$
$$T(n1, n2) = i*1 + T(n1-i, n2)$$

Solving for i = n1 to get:

$$T(n1, n2) = n1 + 1$$

Which makes `Enigma` *O(N)*, where N is n1. The parameter n2 is irrelevant because has no effect on the work done at each step or the progress toward the base case.

Now consider the `Mystery` function. Using T for Mystery's time and TE for Enigma's:

$$T(n) = \{ 1 \quad \text{if } n = 0$$

TE(2, _) + T(n-1)   otherwise }

I used an underbar as a placeholder in the TE equation because we don't need to know the value of the second parameter since it is irrelevant in computing the time taken by `Enigma`. TE(2, anything) requires a constant amount of time always, thus we simplify to:

$$T(n) = \begin{cases} 1 & \text{if } n = 0 \\ 1 + T(n-1) & \text{otherwise} \end{cases}$$

Which again is a linear recurrence, this time in terms of n. `Mystery` is *O(N)* as well.

Problem 2: Recursive procedures

The description of the problem in the practice handout had a small error at the end. It is possible to create a chain of the first four dominos (in fact, the given sequence for five contains it as the first four dominos), but trying with only the first three or two dominos in the array would fail. Sorry about that.

This solution is designed as a variant on the permutation program. For each position, we consider all the possibilities by examining each remaining array entry. Unlike permutations, where every entry must be tried at each position, we optimize by only trying those dominos that make a legal chain with what we have so far. Once we find any legal chain, we can stop looking.

```
bool FormsDominoChain(dominoADT dominos[], int n)
{
    return (RecDominoChain(dominos, 0, n));
}

/*
 * Function: RecDominoChain
 * Usage: chainExists = RecDominoChain(dominos, k, n);
 * ---------------------------------------------------
 * This function is based on a similar idea to the recursive permutation
 * algorithm from Chapter 5.  At each step, we consider each remaining
 * domino that can a legal neighbor the chain so far, swap it into
 * position, and then try to solve from there.
 */
static bool RecDominoChain(dominoADT dominos[], int k, int n)
{
   int i;
   bool success = FALSE;

   if (k == n) { // successfully made it to end, have formed chain!
      return TRUE;
   } else {
      for (i = k; i < n && !success; i++) { // stop at success
        if (k == 0 || RightDots(dominos[k-1]) == LeftDots(dominos[i])) {
           ExchangeDominos(dominos, k, i);
           success = RecDominoChain(dominos, k + 1, n);
           ExchangeDominos(dominos, k, i);
        }
      }
      return success;
   }
}


static void ExchangeDominos(dominoADT dominos[], int p1, int p2)
{
    dominoADT tmp;

    tmp = dominos[p1];
    dominos[p1] = dominos[p2];
    dominos[p2] = tmp;
}
```

Problem 3: Editor buffers

The following code seems to be the most straightforward.

```
void Cut(bufferADT buffer, int n)
{
   int i;

         // first empty any previous clipboard contents
   while (!IsStackEmpty(clipboard))
      Pop(clipboard);

   for (i = 0; i < n && !IsStackEmpty(buffer->after); i++)
      Push(clipboard, Pop(buffer->after));
}

void Paste(bufferADT buffer)
{
   int i, n;

      // note we access them in opposite order than stored
      // we also don't destroy the clipboard, just peek at
      // contents so we can multiple pastes later
   n = StackDepth(clipboard);
   for (i = n - 1; i >= 0; i--)
      Push(buffer->before, GetStackElement(clipboard, i));
}
```

The temptation to break the wall and poke around inside the underlying representation shows up above. When cutting old text, it is necessary to clear the clipboard first, but no `ClearStack` function exists in the interface. Writing

```
clipboard->count = 0
```

has the right effect given an array-based stack and is clearly faster than

```
while (!IsStackEmpty(clipboard))
   Pop(clipboard);
```

but the latter coding does not violate the abstraction boundary. Probably the best solution would be to negotiate with the stack interface designer to add the **`ClearStack`** function when you realized it was necessary.

Problem 4: Data Structure Design and ADTs
This implementation keeps each row in linked list, sorted by column order, it does not use a dummy header cell, but doing so would have meant a little more work to set up a new matrix and one less special case to handle in SetElement.

```
typedef struct _cell {
   int col;
   double val;
   struct _cell *next;
} Cell;

struct matrixCDT {
   Cell **rows;
   int numRows, numCols;
};


matrixADT NewMatrix(int numRows, int numCols)
{
   matrixADT m = New(matrixADT);
   int i;

   m->rows = NewArray(numRows, Cell *);
   for (i = 0; i < numRows; i++)
      m->rows[i] = NULL; // all rows start empty
   m->numRows = numRows;
   m->numCols = numCols;
   return m;
}

void SetElementAt(matrixADT m, double value, int row, int col)
{
   Cell *prev, *newOne;

   if (row < 0 || row >= m->numRows || col < 0 || col >= m->numCols)
      Error("SetElement out of bounds!");

   prev = FindPrevCell(m->rows[row], col);
   if (prev == NULL) {        // insert new cell at head of list
      newOne =  CreateCell(col, value);
      newOne->next = m->rows[row];
      m->rows[row] = newOne;
   } else if (prev->next != NULL && prev->next->col == col) {
      prev->next->value = value; // update existing cell to new val
   } else {             // insert new cell in middle of list
      newOne = CreateCell(col, value);
      newOne->next = prev->next;
      prev->next = newOne;
   }
}
```

```
// An internal helper to find the cell that would precede a given
// column in the list of cells for this row, this will be useful when
// trying to insert a new cell. It will return NULL of the col belongs
// at the head of the list
static Cell *FindPrevCell(Cell *head, int col)
{
   Cell *cur, *prev = NULL;

   for (cur = head; cur != NULL && cur->col < col; cur = cur->next)
      prev = cur;
   return prev;
}


// An internal helper to create a new cell
static Cell *CreateCell(int col, double value)
{
   Cell *newOne = New(Cell *);
   newOne->col = col;
   newOne->value = value;
   newOne->next = NULL;
   return newOne;
}


// In order to make this O(NR *NC) we have to walk the matrix in
// in order (ie we cannot just iterate through calling GetElementAt
// on each row/col combination, that would be O(NR*NC2).
void PrintMatrix(matrixADT m)
{
   int row, col;
   Cell *cur;

   for (row = 0; row < m->numRows; row++) {
      cur = m->rows[row];
      for (col = 0; col < m->numCols; col++) {
         if (cur != NULL && cur->col == col) {
            printf("%g  ", cur->value);
            cur = cur->next;
         } else // no element in that col, assumed 0
            printf("0.0  ");
      }
      printf("\n");
   }
}
```

Problem 5: Symbol tables

```
 cellT *FindAndReposition(symtabADT table, int bucket, string key)
 {
     cellT *cp, *prev;

     cp = table->buckets[bucket];
     prev = NULL;
     while (cp != NULL && !StringEqual(cp->key, key)) {
         prev = cp;
         cp = cp->link;
     }
     if (prev != NULL && cp != NULL) {
         prev->link = cp->link;
         cp->link = table->buckets[bucket];
         table->buckets[bucket] = cp;
     }
     return (cp);
}
```