Alan Marcero

Algorithm Analysis

Professor Ed Harcourt

Friday Dec 1st, 2006

1. Use the LCS algorithm developed in class to compute the LCS of "exponential" and "polynomial". Show all of your work including the table.

LCS: PONIAL

			E	X	P	O	N	E	N	T	I	A	L
		0	1	2	3	4	5	6	7	8	9	10	11
	0	0	0	0	0	0	0	0	0	0	0	0	0
P	1	0	0	0	1	1	1	1	1	1	1	1	1
O	2	0	0	0	1	2	2	2	2	2	2	2	2
L	3	0	0	0	1	2	2	2	2	2	2	2	3
Y	4	0	0	0	1	2	2	2	2	2	2	2	3
N	5	0	0	0	1	2	3	3	3	3	3	3	3
O	6	0	0	0	1	2	3	3	3	3	3	3	3
M	7	0	0	0	1	2	3	3	3	3	3	3	3
I	8	0	0	0	1	2	3	3	3	3	4	4	4
A	9	0	0	0	1	2	3	3	3	3	4	5	4
L	10	0	0	0	1	2	3	3	3	3	4	5	6

2. Assume we are computing the LCS of S and T using the algorithm we developed in class (not the bad recursive algorithm).

a) **Find <u>and prove</u> a run-time Big-Oh for the LCS algorithm that computes the length of the LCS.**

```
LCS(s, t) {
   // Fill in row and column 0
   for j = 0 to s c[0][j] = 0; // O(s)
   for i = 0 to t C[i][0] = 0; // O(t)

   for i = 1 to s     // for each row i
      for j = 1 to t  // for each column j of I O(s*t)

            //do constant operation

}
```

The two initial for loops run (S + T) times, the for loop with the nested for loop runs (S * T) times. Thus this algorithm takes (S + T) + (S * T) or $2n+n^2$ which is $O(n^2)$

Proof:

$2n + n^2 \le C_1 n^2$

replace 2n with 2n[2]:

$2n^2 + n^2 \le C_1 n^2 = 3n^2 \le C_1 n^2$

set the constant = 3:

$3n^2 \le 3n^2$

b) **Find <u>and prove</u> a Big-Oh that describes the amount of memory that LCS uses.**

```
LCS(s, t) {
   // Fill in row and column 0
   for j = 0 to s c[0][j] = 0;
   for i = 0 to t C[i][0] = 0;

   for i = 1 to s    // for each row i
      for j = 1 to t  // for each column j of I O(s*t)

           //do constant operation

}
```

In the second pair of for loops, the algorithm fills a two dimensional array of size (S * T). Thus this algorithm takes up (S * T) space in memory or O(ST) space.

Proof:

$\mathrm{ST} \le C_1 \mathrm{ST}$

Set the constant = 1:

$\mathrm{ST} \le \mathrm{ST}$

3. Develop a version of LCS that uses O(n+m) space.

```
// The LCS algorithm only looks at the array position to the top
// of, and directly to the left of the current iteration in order
// to calculate the LCS.  This algorithm utilizes that fact in
// order to implemented an LCS algorithm that uses only the space
// of two arrays in memory - O(n+m).

LCS(m, n) {
   // Fill in the top array with 0s and position 0 of the bottom
   // array with a 0
   for (i = 0 to m.length + 1) { top[i] = 0; }
   bottom[0] = 0;

   for i = 1 to m.length    // for each row i
       for j = 1 to n.length { // for each column j of i

        // if the string chars are equal at the current iteration
        // set the current array position to 1 + the
        // left-diagonal array position
         if m[i] = n[j]
           bottom[i] = 1 + top[j-1];

        // if they are not equal, set the current array position
        // to the max of the left array position, and the
        // the adjacent top array position
         else
           bottom[j] = max(bottom[j-1], top[j])

       }

      // copy the bottom row to the top row and continue
      top = bottom;

   }

   // the algorithm is now complete, return the last position of
   // the bottom array
   return bottom[m.length];

} // end LCS
```

4. Develop a modified version of LCS that computes the *minimal* number of insert and delete editsneeded to convert a string S into a string T (rather than the length of the LCS of S and T).

```
//
// This modified version of LCS computes the minimal number of
// edits (inserts and deletes) required to change the input
// string s into the input string t
//
// First it fills the top row and left-most column with
// 0..s.length or 0..t.length
// Second it calculates the edit distance by going through each
// cell in the 2-dimensional array.
//
// It does this in two ways:
// If the chars of the two strings s&t are equal at the current
// iteration, set the current cell block to the key in the
// left-diagonal cell.
//
// If they are not equal, calculate the min of the position to
// the left of the current iteration + 1 and to the top of the
// current iteration + 1 and set the current cell block to the
// min.
//
// Then proceed to fill the rest of the table cells until it is full
//
editDistance(s, t) {

   // Fill in row and column 0 with 0..n

   // fill row 0 with 0..s.length
   for j = 0 to s.length C[0][j] = j;

   // fill column 0 with 0..t.length
   for i = 0 to t.length C[i][0] = i;

   for i = 1 to s.length  {  // for each row i
      for j = 1 to t.length { // for each column j of i

                // if the two strings are equal at this iteration
                if (s[i] == t[j])
                        C[i][j] = C[i-1][j-1];
                // if they are not equal, calculate the min of the
                // the top position + 1 or the left position + 1
                // then place the min in the current table cell
                else
                        C[i][j] = min(
                                        C[i][j-1]+1,
                                        C[i-1][j]+1
                                        );
        }
    }
   // the algorithm is complete, return the final position in
   // the table
   return C[s.length][t.length];
}
```

5. Modify the algorithm you developed in 4 to compute the *minimal* number of edits including the change operation needed to convert S into T. Trace your algorithm on “exponential” and “polynomial”.

```
//
// This modified version of the LCS based editDistance algorithm
// also accounts for the “change operation” required to change a
// string s into the string t.
//
// It is nearly exactly the same as the original algorithm with
// one key difference:
//
// If they are not equal, calculate the min of the position to
// the left of the current iteration + 1, to the top of the
// current iteration + 1, and to the diagonal-left of the current
// position + 1.  Then set the current cell block to the min.
//
editDistance(s, t) {

   // Fill in row and column 0 with 0..n

   // fill row 0 with 0..s.length
   for j = 0 to s.length C[0][j] = j;

   // fill column 0 with 0..t.length
   for i = 0 to t.length C[i][0] = i;

   for i = 1 to s.length  {  // for each row i
      for j = 1 to t.length { // for each column j of i

               // if the two strings are equal at this iteration
               if (s[i] == t[j])
                       C[i][j] = C[i-1][j-1];

               // Here is the key difference:
               // if they are not equal, calculate the min of the
               // the top position + 1, the left position + 1,
               // and the diagonal-left position + 1.  Then place the
               // min in the current table cell.
               else
                       C[i][j] = min(
                                       C[i][j-1]+1,
                                       C[i-1][j]+1,
                                       C[i-1][j-1]+1
                                       );
        }
    }
   // the algorithm is complete, return the final position in
   // the table
   return C[s.length][t.length];
}
```

After the first two for loops run:

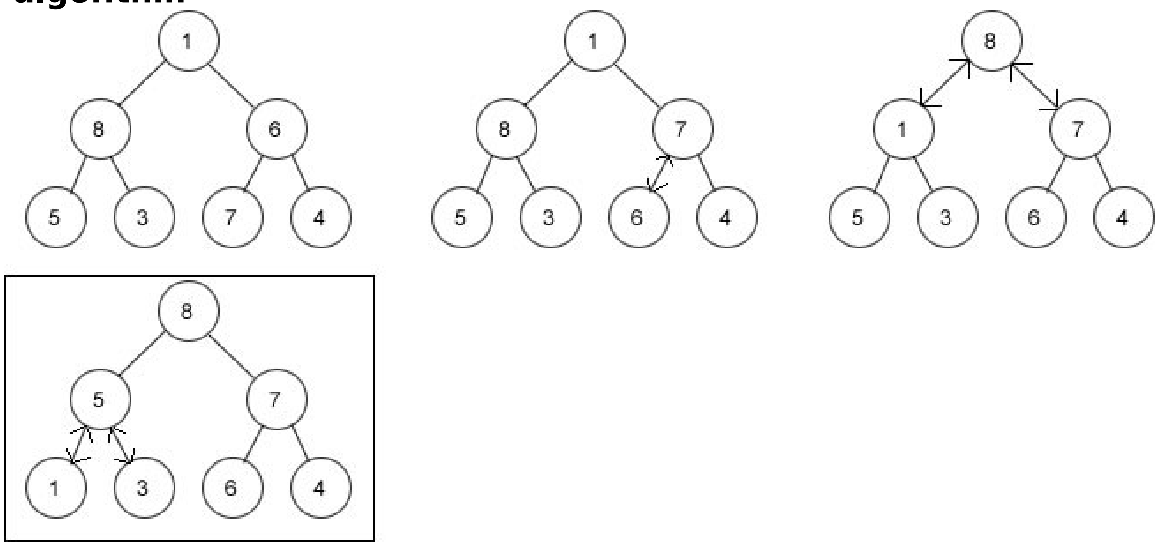
		e	**x**	**p**	**o**	**n**	**e**	**n**	**t**	**i**	**a**	**l**
	0	1	2	3	4	5	6	7	8	9	10	11
p	1											
o	2											
l	3											
y	4											
n	5											
o	6											
m	7											
i	8											
a	9											
l	10											

When the algorithm is complete:

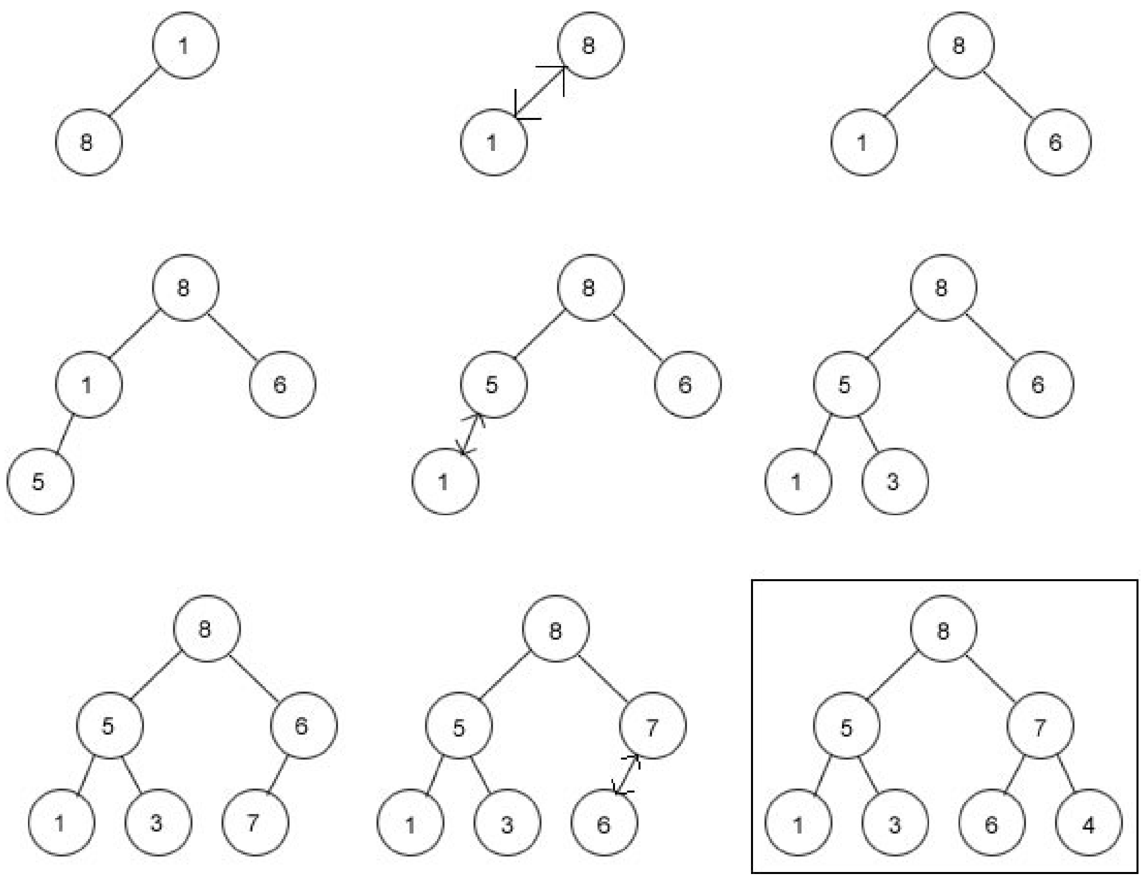
		e	**x**	**p**	**o**	**n**	**e**	**n**	**t**	**i**	**a**	**l**
	0	1	2	3	4	5	6	7	8	9	10	11
p	1	1	2	2	3	4	5	6	7	8	9	10
o	2	2	2	3	2	3	4	5	6	7	8	9
l	3	3	3	3	3	3	4	5	6	7	8	8
y	4	4	4	4	4	4	4	5	6	7	8	9
n	5	5	5	5	5	4	5	4	5	6	7	8
o	6	6	6	6	5	5	5	5	5	6	7	8
m	7	7	7	7	6	6	6	6	6	6	7	8
i	8	8	8	8	7	7	7	7	7	6	7	8
a	9	9	9	9	8	8	8	8	8	7	6	7
l	10	10	10	10	9	9	9	9	9	8	7	**6**

The edit distance between exponential and polynomial is 6.

6a. Construct a heap for the list1, 8, 6, 5, 3, 7, 4 by the bottom-up algorithm



6b. Construct a heap for the list 1, 8, 6, 5, 3, 7, 4 by successive key insertions (top-down algorithm)



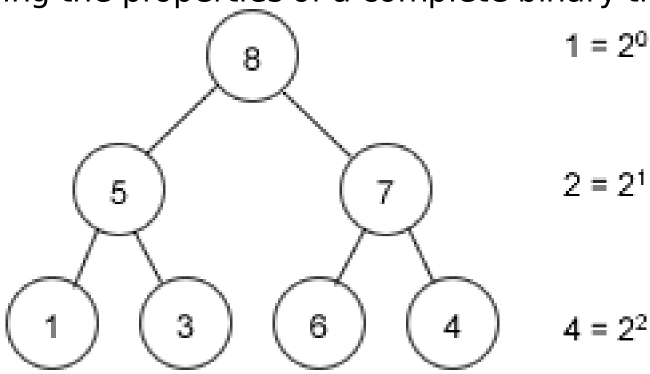
7. Outline an algorithm for checking whether an array H[1…n] is a heap and determine its time efficiency.

```
   // initially call isHeap with the array h and 1
   // precondition: the array has no value set, or an
   // abstract value set, in position 0

   isHeap(array h, index i) {

         // base case, if we've reached the end of the parents
      // without returning false, it is a heap, return true

         if(i > h.length/2) return true;

         // verify that i is >= its children

         // we must first verify that the right child of i exists
         if(h[i*2+1] != NULL) {

            // right child of i exists
            if(h[i] >= h[i*2] && h[i] >= h[i*2+1])
                 isHeap(h, i+1);
            else
                 return false; // not a heap
      }

      else {

            // right child of i does not exist
            if(h[i] >= h[i*2])
                 isHeap(h, i+1);
            else
                 return false; // not a heap
      }
} // end isHeap
```

The isHeap algorithm will recur, at most, n/2 times where n is the length of the array. Each recurrence is C time, n/2 is O(n). Thus this is a O(n) algorithm.

8b) Prove that the height of a heap with n nodes is equal to $\lfloor \log_2(n) \rfloor$

For i = the current depth in the heap
The max nodes per level $=2^i$
(using the properties of a complete binary tree)



By summation formula #5, Page 470 (Levitin), the max amount of nodes n in a tree of height h is equal to:

$$\sum_{i=0}^{h} 2^i = 2^{h+1} - 1 = n$$

Add one to the right side to make an inequality:

$$n < 2^{h+1}$$

2^h is less than or equal to n since 2^{h+1} is strictly greater than n

$$2^h \le n < 2^{h+1}$$

Take the log of all three:

$$\log(2^h) \le \log(n) < \log(2^{h+1})$$

By log law #3, Page 469 (Levitin):

$$\log(2^h) = h$$
$$\log(2^{h+1}) = h + 1$$

On a number line, h is less than or equal to log(n) and h+1 is strictly greater than log(n):

$$h \le \log(n) < h + 1$$

Taking the floor of log(n) makes log(n) = h

Thus: $h = \lfloor \log_2(n) \rfloor$

9. Prove the following equality:

$$\sum_{i=0}^{h-1} 2(h-i)2^i = 2(n - \log_2(n+1)) \quad \text{where} \quad n = 2^{h+1} - 1$$

Express the right-hand-side as a function of h:

$$\sum_{i=0}^{h-1} 2(h-i)2^i = 2(2^{h+1} - 1 - \log_2(2^{h+1}))$$

Simplify the summation through distribution:

$$\sum_{i=0}^{h-1} 2(h-i)2^i = \sum_{i=0}^{h-1} (2h-2i)2^i = \sum_{i=0}^{h-1} 2h2^i - 2i2^i$$

By Sum Manipulation Rule #2, Page 470 (Levitin)

$$\sum_{i=0}^{h-1} 2h2^i - 2i2^i = \sum_{i=0}^{h-1} 2h2^i - \sum_{i=0}^{h-1} 2i2^i$$

By Sum Manipulation Rule #1, Page 470 (Levitin)

$$\sum_{i=0}^{h-1} 2h2^i = 2h\sum_{i=0}^{h-1} 2^i$$

By Summation Formula #6, Page 470 (Levitin)

$$\sum_{i=0}^{h-1} 2i2^i = 2\left[((h-2)2^h + 2)\right]$$

By Summation Formula #5, Page 470 (Levitin)

$$2h\sum_{i=0}^{h-1} 2^i = 2h(2^h - 1)$$

Thus:

$$\sum_{i=0}^{h-1} 2h2^i - \sum_{i=0}^{h-1} 2i2^i =$$

$$2h(2^h - 1) - 2\left[((h-2)2^h + 2)\right]$$

Simplify:

$$2h(2^h - 1) - 2\left[((h-2)2^h + 2)\right]$$

Factor out 2:

$$2\left[h(2^h - 1) - ((h-2)2^h + 2)\right]$$

Distribute:
$2(h2^h - h - h2^h + 2^{h+1} - 2)$

Cancel like terms:
$2(-h + 2^{h+1} - 2)$

We are done simplifying, put the simplified left term = the un-simplified right term
$2(-h + 2^{h+1} - 2) = 2(2^{h+1} - 1 - \log_2(2^{h+1}))$

By log law #3, Page 469 (Levitin):
$\log_2(2^{h+1}) = (h+1)\log(2)$

By log law #2, Page 469 (Levitin):
$\log_2(2) = 1$

Thus:
$\log_2(2^{h+1}) = h + 1$

Put h+1 back into the equation for $\log_2(2^{h+1})$:
$2(-h + 2^{h+1} - 2) = 2(2^{h+1} - 1 - (h+1))$

Distribute the negative on the right side:
$2(-h + 2^{h+1} - 2) = 2(2^{h+1} - 1 - h - 1)$

Combine like terms and reorder:
$2(2^{h+1} - h - 2) = 2(2^{h+1} - h - 2)$

The two sides are equal, thus $\sum_{i=0}^{h-1} 2(h-i)2^i = 2(n - \log_2(n+1))$ is true.