Data Structures and Algorithms — CSCI 230
Algorithm Analysis

Here are the solutions to the last problems in the Chapter 2 notes.

Exercises:

1. Derive a recursive equation to analyze MergeSortand then solve this equation. Assume the array is of size $n = 2^k$ for integer $k \geq 0$. For completeness, here is the algorithm (combining material from Ch 1 and the Ch 1 review):

```
template <class T>
void MergeSort( T * pts, int n )
{
  MergeSort( pts, 0, n-1 );
}

templage <class T>
void MergeSort( T * pts, int low, int high )
{
  if ( low == high ) return;

  int mid = (low + high) / 2;
  MergeSort( T, low, mid );
  MergeSort( T, mid+1, high );

  //  At this point the lower and upper halves
  //  of "pts" are sorted. All that remains is
  //  to merge them into a single sorted list.
  T* temp = new T[high-low+1];  // scratch array for merging
  int i=low, j=mid+1, loc=0;

  //  while neither the left nor the right half is exhausted,
  //  take the next smallest value into the temp array
  while ( i<=mid && j<=high ) {
    if ( pts[i] < pts[j] ) temp[loc++] = pts[i++];
    else                   temp[loc++] = pts[j++];
  }

  //  copy the remaining values --- only one of these will iterate
  for ( ; i<=mid; i++, loc++ ) temp[loc] = pts[i];
  for ( ; j<=high; j++, loc++ ) temp[loc] = pts[j];

  //  copy back from the temp array
  for ( loc=0, i=low; i<=high; loc++, i++ ) pts[i]=temp[loc];
```

```
        delete [] temp;
    }
```

Solution:

We can count the number of comparison and assignments on the arrays `pts` and `temp` as a function of $n = high - low + 1$ (initially, $low = 0$ and $high = n-1$). Define $A(n)$ to be this number and assume $n = 2^k$ for some integer $k >= 0$. The cost of the recursive calls is $2 \cdot A(n/2)$ since the array is split exactly in half for each call. Counting the rest, the `while` loop executes at most $n-1$ iterations with 1 comparison and one assigment per iteration. The two `for` loops that follow, execute at least 1 iterations and at most $n/2$ iterations, with 1 assignment per iteration. The final `for` loop involves at most n assignments. Overall, this work is bounded by $c \cdot n$ for some small constant c, which is about 4. It turns out not to matter.

Putting this together we get

$$
\begin{aligned}
A(n) &= 2A(n/2) + c\,n \\
A(1) &= 0
\end{aligned}
$$

or, using the assumption that $n = 2^k$,

$$
\begin{aligned}
A(2^k) &= 2A(2^{k-1}) + c\,2^k \\
A(2^0) &= 0
\end{aligned}
$$

Using back substitution,

$$
\begin{aligned}
A(2^k) &= 2[2A(2^{k-2}) + c\,2^{k-1}] + c\,2^k = 2^2 A(2^{k-2}) + 2c2^k \\
&= 2^2[A(A(2^{k-3}) + c\,2^{k-2}] + 2\,c\,2^k = 2^3 A(2^{k-3}) + 3c2^k \\
&\dots \\
&= 2^k A(2^0) + kc2^k = k\,c\,2^k = c\,(\log_2 n)n \qquad = O(n \log n)
\end{aligned}
$$

2. Find an efficient algorithm (along with a running time analysis) to find the *minimum* subsequence sum.

   Solution:

   All we need to do is to reverse the signs on the values in the array, run the linear time MaxSubsequence sum algorithm, and negate the result. This gives an $O(n)$ algorithm.