Assignment 2

Sample Solutions

1

- by case 3 of Master's theorem $T(n) = 8T(n/3) + O(n^2) \Rightarrow T(n) = O(n^2)$
- by case 3 of Master's theorem $T(n) = O(n^2)$
- by case 2 of Master's theorem $T(n) = O(n^2 \lg n)$
- by case 2 of Master's theorem $T(n) = O(n^{\log_3 3} \lg^{k+1} n) = O(n \lg^2 n)$. Note that k is the power of $\lg n$ where $f(n) = \Theta(n^{\log_b a} \lg^k n)$

2

Using k bits, 2^k different indices could be made. Since $\sum_{1 \leq i \leq \lg n - 2} 2^i < 2^{\lg n - 1} \leq n/2$, at most half of indices can be of length less or equal to $\lg n - 2$.

Note that in this problem "00" and "0" are different indices.

3

```
Rev(n)
  If (n = 1) Return 0;
  A = Rev(n/2);
  A = A*2;\\inserting a 0 to the left of each number in Rev(n/2)
  B = A+1;\\inserting a 1 to the left of each number in Rev(n/2)
  Return (A,B)\\appending the two arrays and returning the result
```

Note that the ith element of $[0 \ldots 2^{k-1})$ and the ith element of $[2^{k-1} \ldots 2^k)$ only differ in their kth bit. Also, $rev(\overline{0x})$ would be $rev(\overline{x})0$

The running time is $T(n) = T(n/2) + O(n)$ which by Master's theorem is $O(n)$.

As an alternative solution, this also would work in $T(n) = 2T(n/2) + O(1) = O(n)$ if initially called by $Rev(n, 0, 1)$:

```
Rev(n,r,d)
  If (n = 1) Print r;
  Rev(n/2,r,2*d);
  Rev(n/2,r+d,2*d);
```

4

Let A and B be two sorted arrays we wish to search. Without loss of generality, let $m_A \leq m_B$. Let m be the mean of m_A and m_B. If $k < m$, then the kth element can not be within the second half of the larger array, B (every element in the second half of B is larger than at least m elements). Similarly, if $k > m$, then the kth element can not be within the first half of the smaller array, A. Note that in the latter case we should search for the $(k - m_A)$th in the remaining of the arrays. This way, we could throw away half of A or B in each iteration and recurse in the remaining parts until one array remains and the problem becomes obvious.

Let $x = m_A m_B$. Then, each time one of the m_A or m_B is halved, x is halved too. This means that the running time of our algorithm is $T(x) = T(x/2) + O(1) = O(\lg x)$. Since $\lg x = \lg m_A m_B = \lg m_A + \lg m_B$, we are good.

5

The basic idea is two divide the set of buildings into two roughly equal sets, solving the problem for them recursively, then merging the two outlines. The merge process could by traversing the two outlines from left to right (similar to the merge process of mergesort) and at each point, setting the outline as the highest of the two outlines at that point.

```
  procedure baseCaseSolution(Bldngs[1..1])
    Skyline = { (Bldngs[1].lx , 0), (Bldngs[1].lx , Bldngs[1].h),
```

```
                (Bldngs[1].rx , Bldngs[1].h), (Bldngs[1].rx , 0) }
   return Skyline


 procedure FindOutline(Bldngs[1..n]) {
   if (n == 1) then
      return baseCaseSolution(Bldngs)

   q = n div 2
   Out1 = FindOutline(Bldngs[1..q])
   Out2 = findOutline(Bldngs[q+1..n])
   Outline = merge(Out1, Out2)
   return Outline
 }

procedure merge(Out1, Out2) {
   Outline = Empty-List
   CurH1 = 0
   CurH2 = 0
   While (Out1 is not Empty and Out2 is not Empty) do {
      if (head(Out1).lx < head(Sk2).lx) {
          CurX = head(Out1).lx
          CurH1 = head(Out1).h
          Append {(CurX, Max(CurH1, CurH2))} to Outline
          Out1.Remove(head)
      }
      else {
```

```
            CurX = head(Out2).lx
            CurH2 = head(Out2).h
            Append {(CurX, Max(CurH1, CurH2))} to Outline
            Out2.Remove(head)
        }
    }
    if (Out1 is Empty)
        Append  Out2 to Outline
    else //Out2 is Empty...
        Append Out1 to Outline
    return Outline
}
```