Recursive MaxSort:

```
public void maxSort(int size)
{      int i, t, maxPos;

       if (size <= 1) return;
       maxPos=0;
       for (i=1; i < size; i++)
          if (A[i] >= A[maxPos]) maxPos=i;
       t= A[maxPos];
       A[maxPos]= A[size-1];
       A[size-1] = t;
       maxSort(size-1);
 }
```

Derive and solve recurrence relations for:

K(n)= number of key comparisons on a problem of size n.

S(n)= number of swaps for a problem of size n.

```
for (end= n-1; end > 0; end--)
{  max_pos=0;
   for (i= 1; i <= end; i++)
   {
     if (A[i] >= A[max_pos] )
         max_pos= i;
   }

//    Swap the max.
//    element with the  end.
   t= A[max_pos];
   A[max_pos]= A[end];
   A[end]= t;
}
```

For the iterative approach: if you think about it in terms of the first iteration of the loop and then solving a problem of size n-1 you get the same recurrence relations.

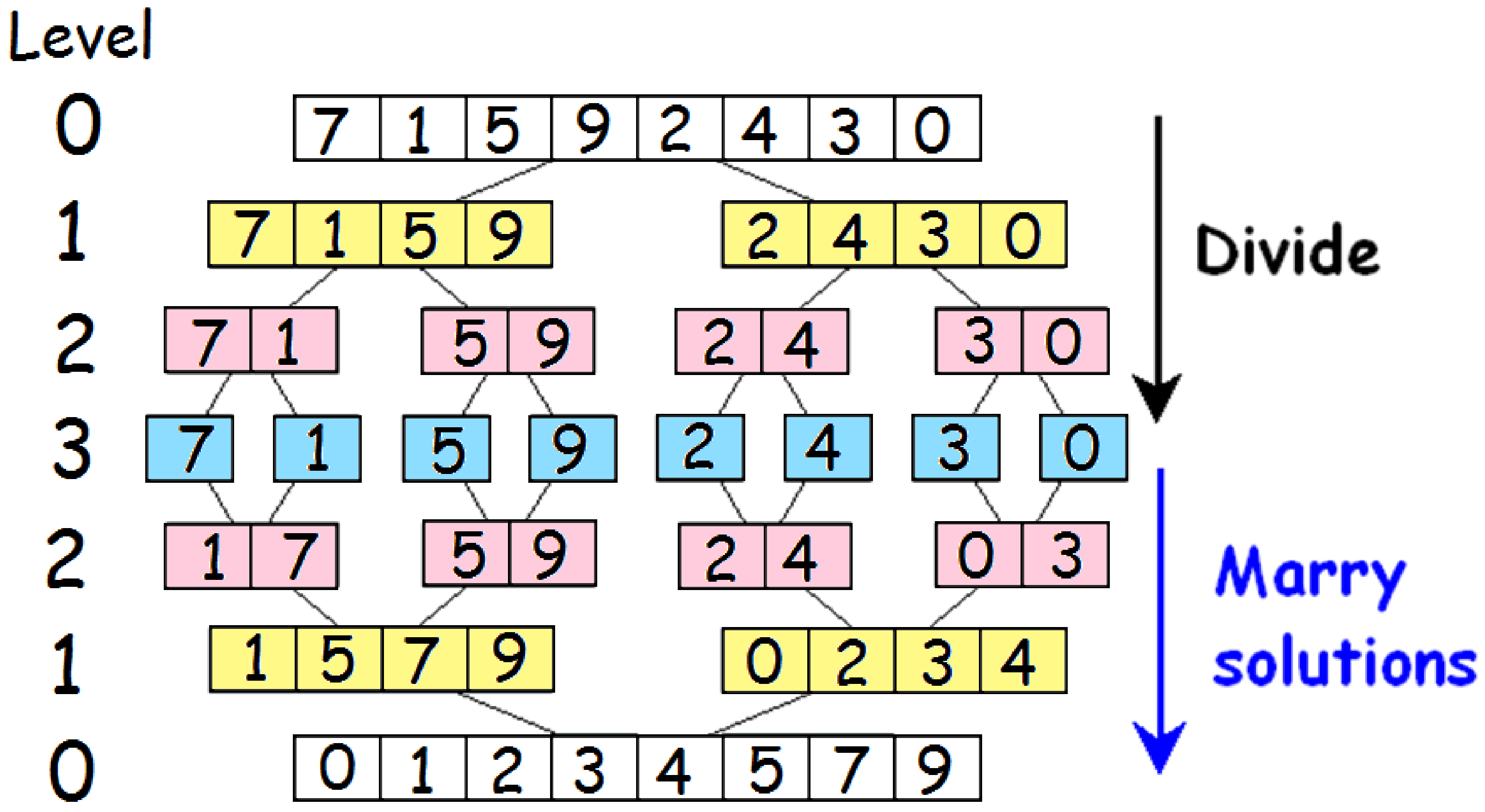
Assignment 1B is due at the beginning of class on Thursday.

Stay after class today if you have questions.

Please turn your submission upside down before you hand it in and write your name but not your ID number on the back of the last page in the upper RH corner. Your name and ID number should be on the first page.

Assignment 2A Programming: due Thursday Oct. 12 at 11:55pm.
Assignment 2B: Written: due Monday Oct. 16 at the beginning of class.

Mergesort



Sorted answer

Divide and Conquer

1. Divide the problem into two or more subproblems.
2. Solve the subproblems.
3. Marry the solutions.

This is one of the most common problem solving tactics and leads naturally to recursive algorithms.

Merge Sort- with linked lists

[Basis] If the list has size 0 or 1 it is already sorted so return.

[Divide] Otherwise, split the list into two lists, list1 and list2, of roughly equal sizes.

[Conquer] Sort list1 and list2 (recursively).

[Marry solutions] Merge list1 and list2 together to get the answer.

Using the data structures for the lab:

```
class ListNode{

   public int data;
   public ListNode next;

   public ListNode(int x, ListNode ptr)
   {
      data= x;
      next= ptr;
   }
}
```

```
public class LinkedList
{
   int n;
   ListNode start;
   ListNode rear;

   public LinkedList()
   {
      n= 0;
      start= null;
      rear= null;
   }
```

```
public class LinkedList // Same as Lab 1
{  int n;
   ListNode start;
   ListNode rear;

   public void mergeSort(int level)
   {
       LinkedList list1;
       LinkedList list2;
       int i;

/*     A list of size 0 or 1 is already sorted. */

       if (n <=1) return;
```

Code omitted here.

I am assuming for this program, the list is split into two sublists list1 and list2 the same way you do it for your reverse method. The list1 has the first floor(n/2) items and the list2 has the next ceiling(n/2).

Make sure that in your code:

list1.start, list1.rear, list1.n, and

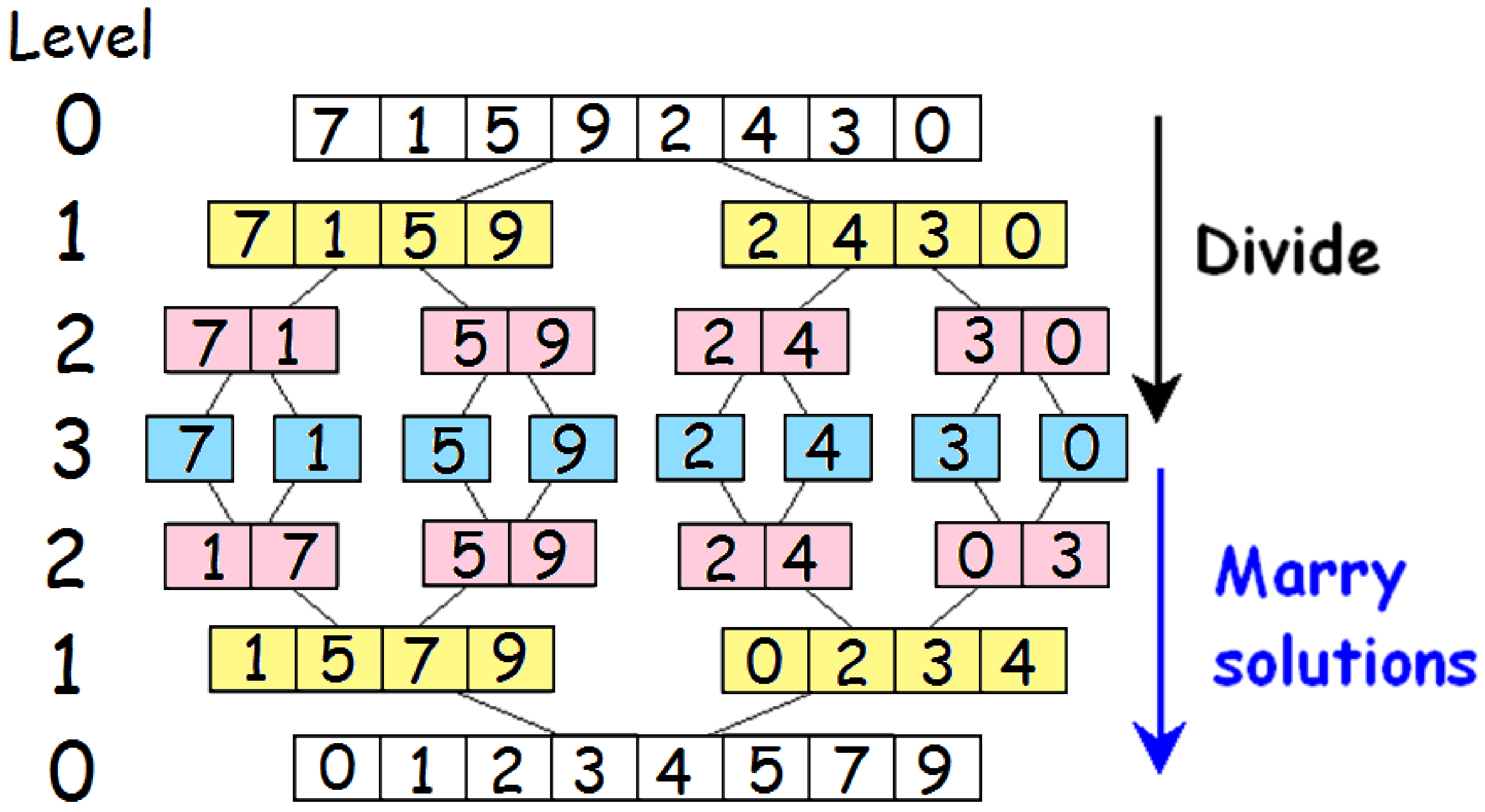
list2.start, list2.rear, list2.n all have correct values and that both list1 and list2 are null terminated.

```
/*     Sort the 2 sublists recursively: */

    list1.mergeSort(level+1);

    list2.mergeSort(level+1);
```

```
/*     Merge the two sorted sublists. */
start= null; rear= null;
LinkedList tmp; // Keeps track of list with smallest key
while (list1.start != null && list2.start != null)
{
       if (list1.start.data < list2.start.data)
              tmp= list1;
       else tmp= list2;

       if (start == null) start= tmp.start;
       else rear.next= tmp.start;

        rear= tmp.start;

      tmp.start= tmp.start.next;
      rear.next= null;
}
```

```
//    Now append the list that still has
//    items in it to the end.

      if (list1.start != null)
           tmp= list1;
       else
           tmp = list2;
       rear.next= tmp.start;
// Make sure our
// object has a correct rear pointer
       rear= tmp.rear;

} // end mergeSort
```

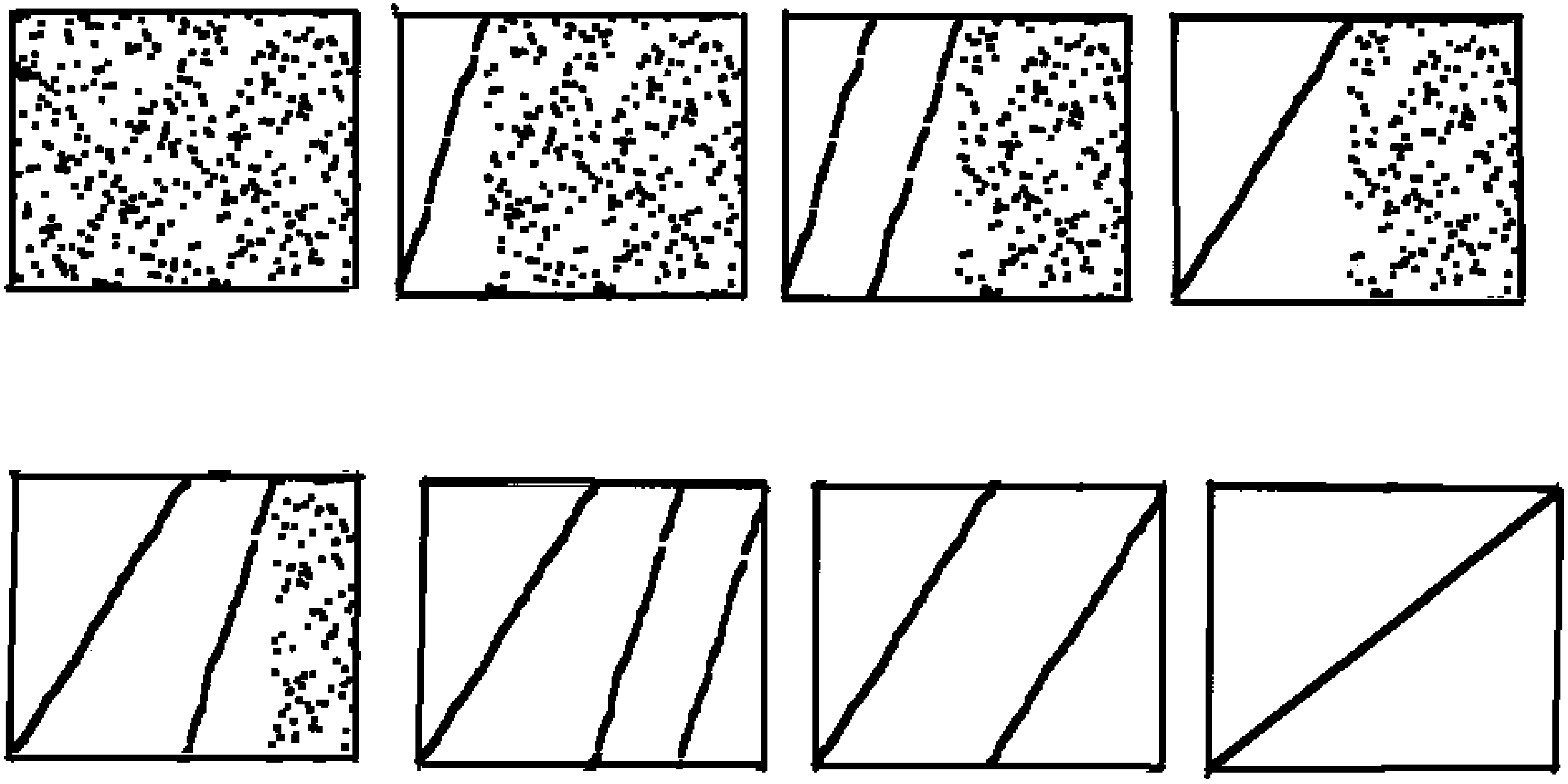
Mergesort



Sorted answer

```
Level 0 : The list to be sorted is:
  7   1   5   9   2   4   3   0
Level 0 : Sublist 1.
  7   1   5   9
Level 0 : Sublist 2.
  2   4   3   0
OUTPUT from levels >= 1 has been deleted.
Level 0 : The 2 sorted lists to merge are:
Level 0 : Sublist 1.
  1   5   7   9
Level 0 : Sublist 2.
  0   2   3   4
Level 0 : After step 1 of the merging.
Level 0 : Sublist 1.
  1   5   7   9
Level 0 : Sublist 2.
  2   3   4
Level 0 : The sorted list being created.
  0
Level 0 : After step 2 of the merging.
Level 0 : Sublist 1.
  5   7   9
Level 0 : Sublist 2.
  2   3   4
Level 0 : The sorted list being created.
  0   1
Level 0 : After step 3 of the merging.
Level 0 : Sublist 1.
  5   7   9
Level 0 : Sublist 2.
  3   4
Level 0 : The sorted list being created.
  0   1   2
Level 0 : After step 4 of the merging.
Level 0 : Sublist 1.
  5   7   9
Level 0 : Sublist 2.
  4
Level 0 : The sorted list being created.
  0   1   2   3
Level 0 : After step 5 of the merging.
Level 0 : Sublist 1.
  5   7   9
Level 0 : Sublist 2.
Level 0 : The sorted list being created.
  0   1   2   3   4
Level 0 : The sorted list.
  0   1   2   3   4   5   7   9
```

Scatter Plots for Merge Sort:



Taken from: Algorithms in C++ by Sedgewick.

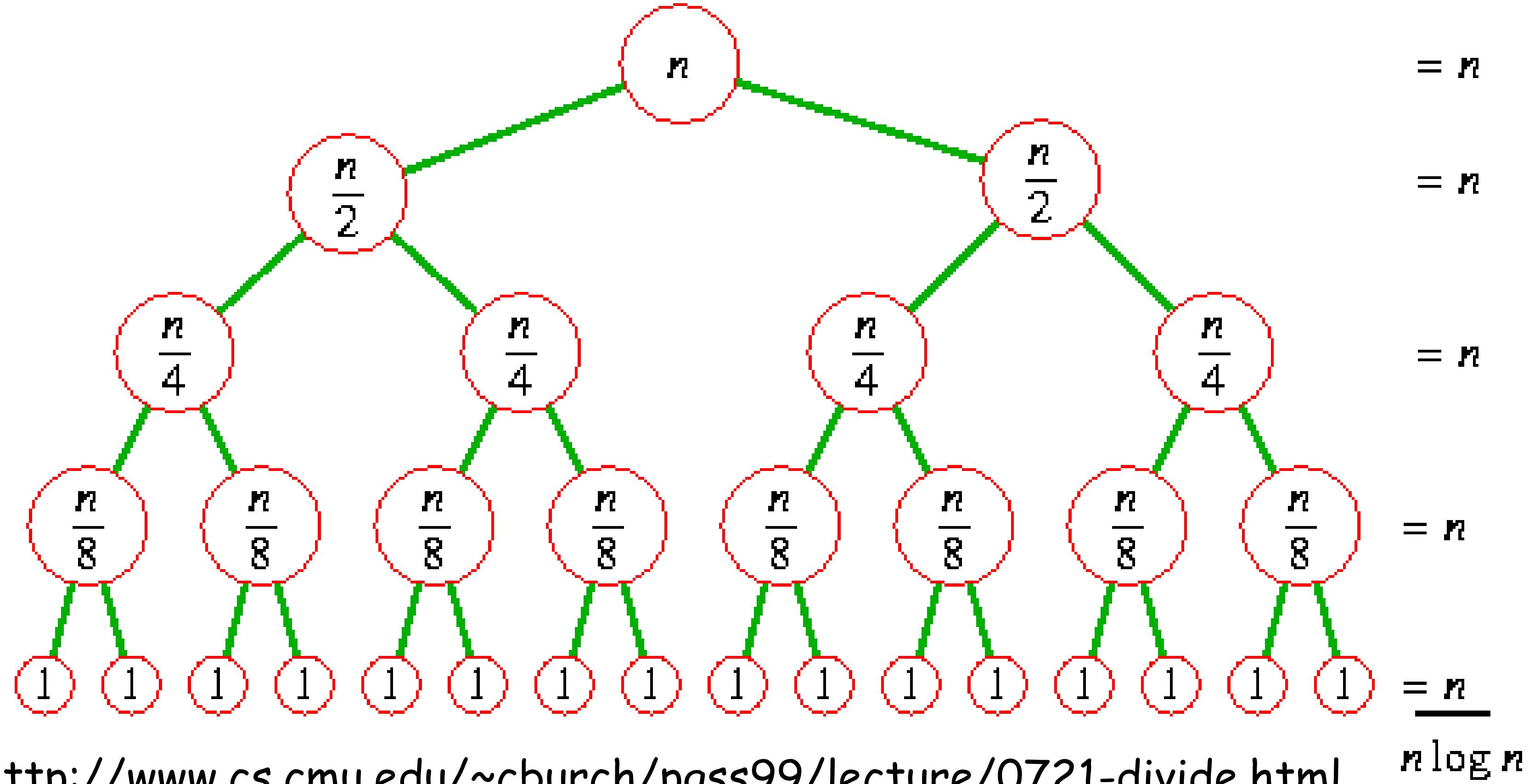
How much work does the Mergesort algorithm do?

1. Divide the problem into two or more subproblems.
2. Solve the subproblems.
3. Marry the solutions.

So a recurrence for the time is:

If n= 2^k, the computation tree has $2^{k+1} - 1$ nodes, height k, and k+1 levels. So the work done by mergesort is proportional to

$$n * (k+1) = n * (\log_2(n) + 1) \in O(n * \log_2(n))$$



http://www.cs.cmu.edu/~cburch/pgss99/lecture/0721-divide.html

$T_0(n) = n + 2\,T(n/2),\ T(1) = 1.$

$T_1(n) = 10n + 20 + 2\,T(n/2),\ T(1) = 30$

$T_2(n) = 10n + 20n + 2\,T(n/2),\ T(1) = 30$

T_0: Used for the time complexity of mergeSort.
T_1: If for any n, mergeSort does at most 10n + 20 machine instructions at the top level of recursion (ignoring those done by a recursive call), $T_1(n)$ is an upper bound on the actual number of machine instructions.
T_2 (n): Upper bound on $T_1(n)$.

$T_0(n) \leq T_1(n) \leq T_2(n) = 30 * T_0(n)$ for all $n \geq 1$.
So the actual number of machine instructions $T_1(n)$ is in $O(T_0(n))$.