Sorting (cont.)

Quicksort

Quicksort

- A "partition-exchange" sorting method:

  Partition an original array into:

  (1) a subarray of small elements

  (2) a single value in-between (1) and (3)

  (3) a subarray of large elements

  Then partition (1) and (3) independently using the same method.

- For partitioning we need to choose a value ***a***. (simply select ***a*** = ***x***[0])
- During a partition process: pairwise exchanges of elements.

Eg. 25 10 57 48 37 12 92 86 33

=> 12 10 25 48 37 57 92 86 33

A possible arrangement: simply use first element (ie. 25) for partitioning

x[0..***N***-1]

Eg. 25 10 57 48 37 12 92 86 33

=> 12 10 25 48 37 57 92 86 33

a

Quicksort

Original: 25 10 57 48 37 12 92 86 33

Partitioning: Select **a = 25**

Use 2 indices:

down **up**

25 10 57 48 37 12 92 86 33

Move **down** towards **up** until x[**down**]>25
Move **up** towards **down** until x[**up**]<=25 (*)

down **up**

25 10 **57** 48 37 **12** 92 86 33

down **up**

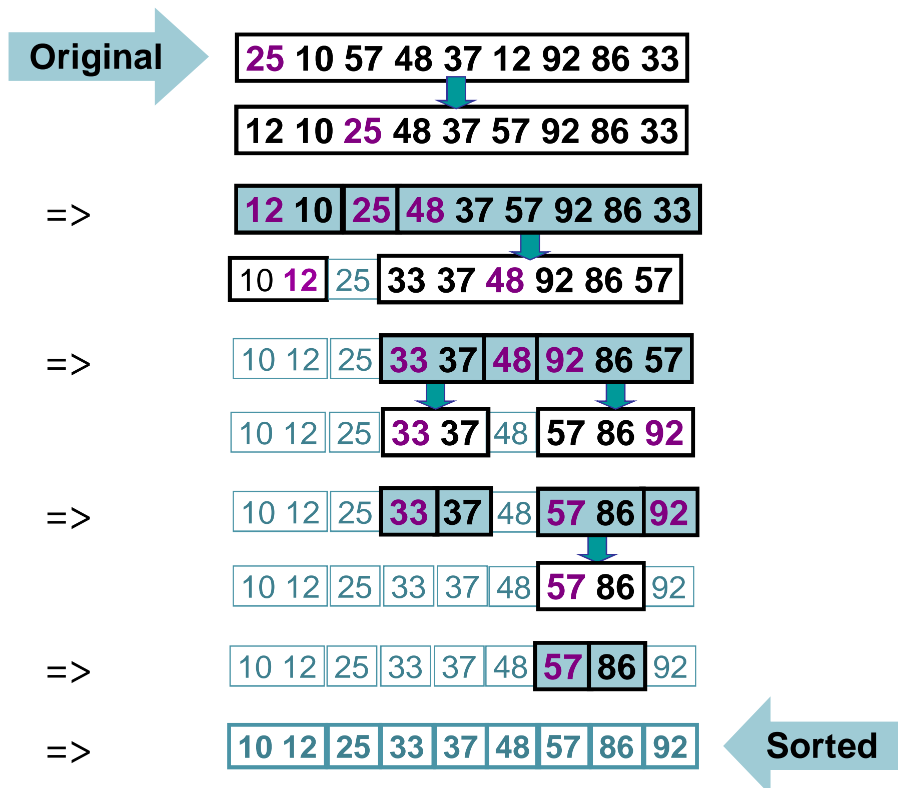
25 10 **12** 48 37 **57** 92 86 33 Swap

up down

25 10 **12** **48** 37 57 92 86 33 Continue repeat **(*)** until **up** crosses **down** (ie. **down** >= **up**)

12 10 **25** 48 37 57 92 86 33 **up** is at right-most of smaller partition, so swap **a** with x[**up**]

Quicksort



Quicksort

```
void quick_sort(int x[ ], int idLeftmost, int idRightmost)
/* Sort x[idLeftmost].. x[idRightmost] into ascending numerical order. */
{
    int j;

    if (idLeftmost >= idRightmost)
        return; /* array is sorted or empty*/

    partition(x, idLeftmost, idRightmost, &j);
        /* partition the elements of the subarray such that one of the elements
            (possibly x[idLeftmost]) is now at x[j] (j is an output parameter) and
            1) x[i] <= x[j] for idLeftmost <= i < j
            2) x[i] >= x[j] for j<i<= idRightmost
            x[j] is now at its final position */

    quick_sort(x, idLeftmost, j-1);
    /* recursively sort the subarray between positions idLeftmost and j-1 */

    quick_sort(x, j+1, idRightmost);
    /* recursively sort the subarray between positions j+1 and idRightmost */
}
```

Quicksort

```
void partition(int x[ ], int idLeftMost, int idRightMost, int *pj)
{
    int down, up, a, temp;
    a = x[idLeftMost];
    up = idRightMost;
    down = idLeftMost;

    while (down < up)
    {   while ((x[down] <= a) && (down < idRightMost))
            down++;            /* move up the array */
        while (x[up] > a)
            up--;              /* move down the array */

        if (down < up)         /* interchange x[down] and x[up] */
        {   temp = x[down];   x[down] = x[up];   x[up] = temp;
        }
    }
   x[idLeftMost] = x[up];
   x[up] = a;
   *pj = up;
}
```

Quicksort

Analysis of Quicksort

- The best case complexity is O(N log N)

  Each time when a is chosen (as the first element) in a partition, it is the median value in the partition. => the depth of the "tree" is O(log N).

- In worst case, it is $O(N^2)$.
  For most straightforward implementation of Quicksort, the worst case is achieved for an input array that is already in order.

  Each time when a is chosen (as the first element) in a partition, it is the smallest (or largest) value in the partition. => the depth of the "tree" is O(N).

- When a subarray has gotten down to some size M, it becomes faster to sort it by straight insertion.

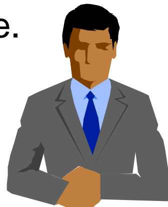
- Fastest sorting algorithm for large N.

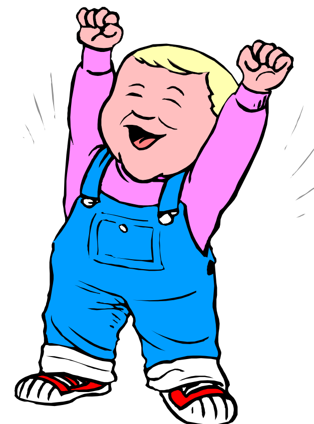
Merge Sort

Merge Sort

- a divide-and-conquer approach
- split the array into two roughly equal subarrays
- sort the subarrays by recursive applications of Mergesort and merge the sorted subarrays

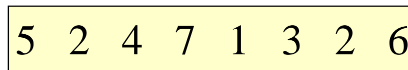
- Suppose there are some people called Mr. MergeSort. They are identical.
- They don't know how to do sorting.
- But each of them has a secretary called Mr. Merge, who can merge 2 sorted sequences into one sorted sequence.





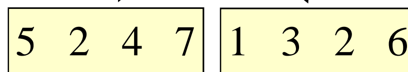
Merge Sort

At the beginning, a Mr. MergeSort is called to sort:



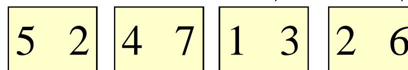
"So complicated!!, I'll split them and call other Mr. MergeSorts to handle."

Then 2 other Mr. MergeSorts are called to sort:



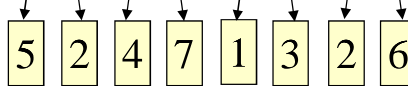
Both of them say "Still complicated! I'll split them and call other Mr. MergeSorts to handle."

Then 4 other Mr. MergeSorts are called to sort:

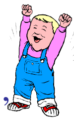


All of them say "Still complicated! I'll split them and call other Mr. MergeSorts to handle."

Then 8 other Mr. MergeSorts are called to sort:

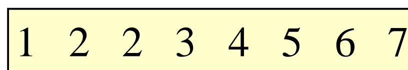


All of them say 'This is easy. No need to do anything.'



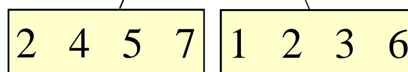
Merge Sort

Then the first Mr. MergeSort succeeds and returns.



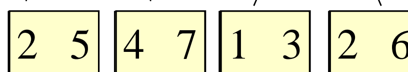
The first Mr. MergeSort calls his secretary Mr. Merge to merge the returned numbers

Then each of the 2 Mr. MergeSorts returns the merged numbers.



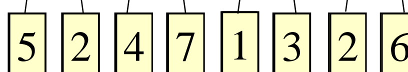
Both Mr. MergeSorts call their secretaries Mr. Merge to merge the returned numbers

Then the 4 Mr. MergeSorts returns the merged numbers.

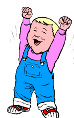


The 4 Mr. MergeSorts call their secretaries Mr. Merge to merge the returned numbers

Then the 8 Mr. MergeSorts return.



All of them say 'This is easy. No need do anything.'



Merge Sort

void MERGE-SORT(x, Lower_bound, Upper_bound)

Sorts the elements:

x = | .. 5 2 4 7 1 3 2 6 .. |

↑ Lower_bound ↑ Upper_bound

```
void merge-sort(int x[ ], int lower_bound, int upper_bound)
{
    int mid;
    if (lower_bound != upper_bound)
    {
          mid = (lower_bound + upper_bound) / 2;
          merge-sort(x, lower_bound, mid);
          merge-sort(x, mid+1, upper_bound);
          merge(x, lower_bound, mid, upper_bound);
    }
}
```

Merge Sort

void merge(int x[ ], int lower_bound, int mid, int upper_bound)

-- merges 2 sorted sequences:

L: x_{lower_bound}, $x_{lower_bound+1}$, … x_{mid}

R: x_{mid+1}, , x_{mid+2}, … x_{upper_bound}

Step 1: Continuously copy the smallest one from L and R to a result list until either L or R is finished.

Step 2: L may still have some numbers not yet copied. So copy them in order.

Step 3: R may still have some numbers not yet copied. So copy them in order.

Step 4: Copy the result list back to x.

L R

x = | .. | 2 4 5 7 | 1 2 3 6 | .. |

↑ x_{lower_bound} ↑ x_{mid} ↑ x_{upper_bound}

x = | .. | 2 4 5 7 | 1 2 3 6 | .. |

Result = | .. | 1 2 2 3 4 5 6 7 | .. |

```
/* Assuming that x[lower_bound..mid] and x[mid+1..upper_bound] are sorted, */
/* this procedure merges the two into x[lower_bound..upper_bound] */
void merge(int x[ ], int lower_bound, int mid, int upper_bound)
{
    int idLeft, idRight, idResult, result[10]; int i;
    idLeft = lower_bound;
    idRight = mid+1;

    // Continuously remove the smallest one from either partitions until any
    // one partition is finished.
    for (idResult = lower_bound; idLeft <= mid && idRight <= upper_bound; idResult++)
    {   if (x[idLeft] <= x[idRight])
            result[idResult] = x[idLeft++];
        else
            result[idResult] = x[idRight++];
    }
    //Copy remaining elements in any unfinished partition to the result list.
    while (idLeft <= mid)
        result[idResult++] = x[idLeft++];
    while (idRight <= upper_bound)
        result[idResult++] = x[idRight++];
    //Copy the result list back to x
    for (i=lower_bound; i<=upper_bound; i++)
        x[i] = result[i];
}
```

Analysis of Merge Sort

void merge(..)

To merge n numbers from 2 sorted arrays, the running time is roughly proportional to n.

Then, what is the complexity of Merge Sort?

```
void merge-sort(int x[ ], int low_bound, int up_bound)
{
    int mid;
    if (low_bound != up_bound)
    {
            mid = (low_bound + up_bound) / 2;
            merge-sort(x, low_bound, mid);
            merge-sort(x, mid+1, up_bound);
            merge(x, low_bound, mid, up_bound);
    }
}
```

To sort x[0..n-1] using Merge Sort, we call **MERGE-SORT(x,0,n-1)**

Let T(n) be the MERGE-SORT running time to sort n numbers.

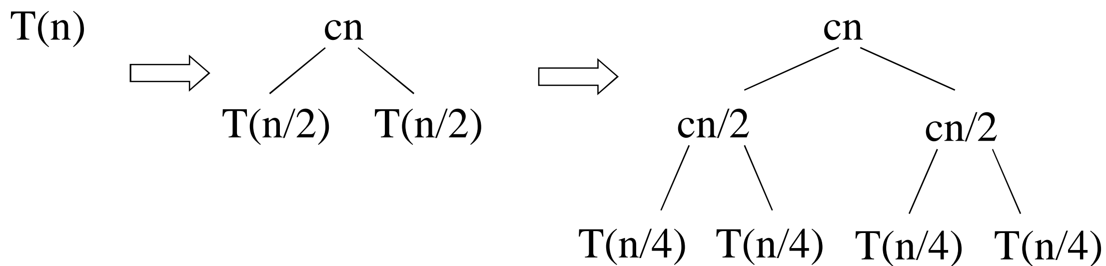
MERGE-SORT involves:

2 recursive calls to itself (ie. 2 * T(n/2)), plus a call to MERGE (ie. c*n, where c is a constant).

The Running time:

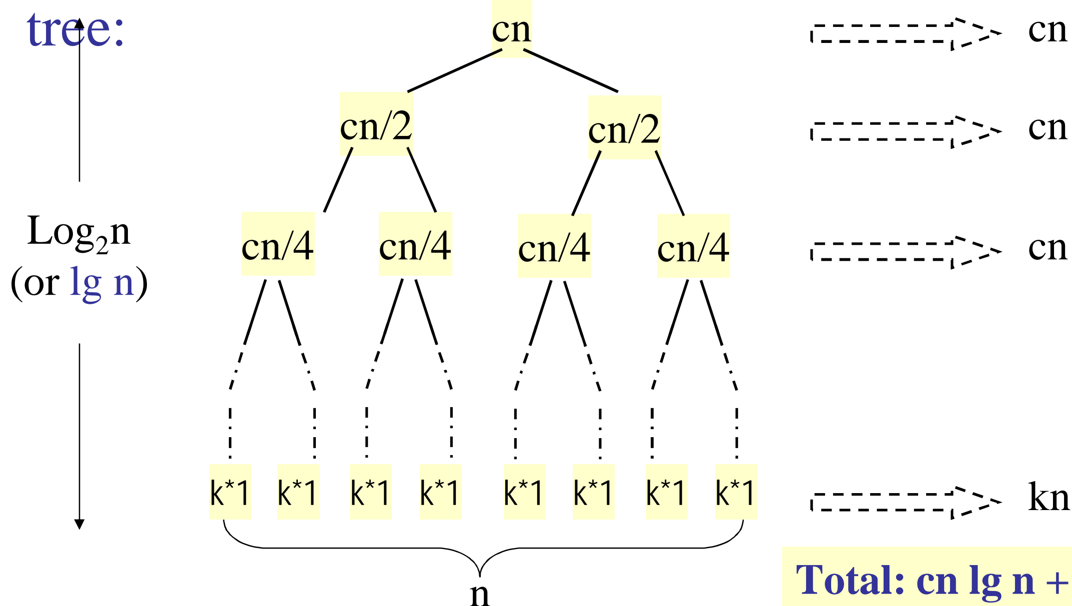
$$T(n) = \begin{cases} k \text{ (a constant)} & \text{if } n=1 \\ 2T(n/2)+ c*n & \text{if } n>1 \end{cases}$$

Analysis of Merge Sort



Analysis of Merge Sort

Fully Expanded recursion tree:



Total: cn lg n + kn

ie. T(n) = O(______)