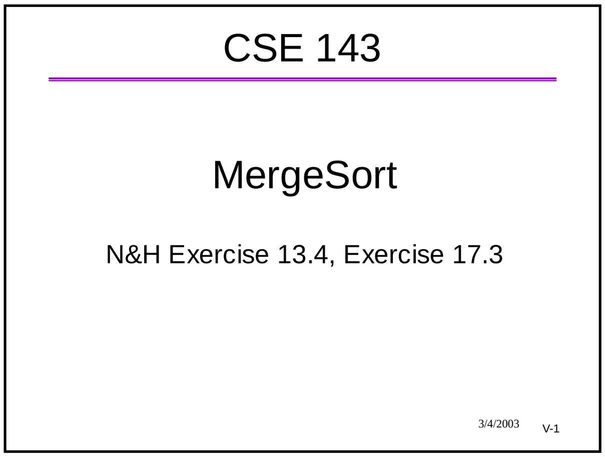
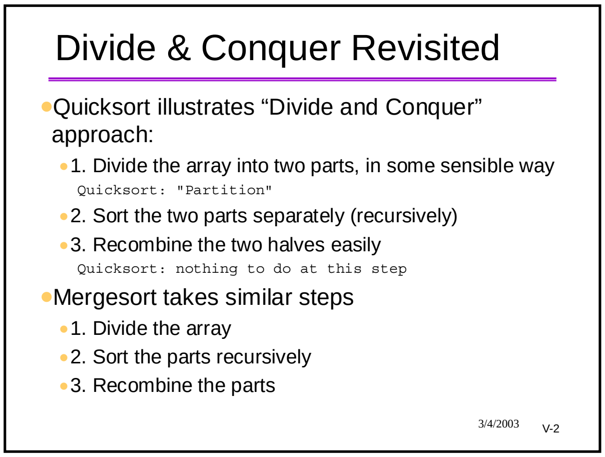


CSE 143

MergeSort

N&H Exercise 13.4, Exercise 17.3

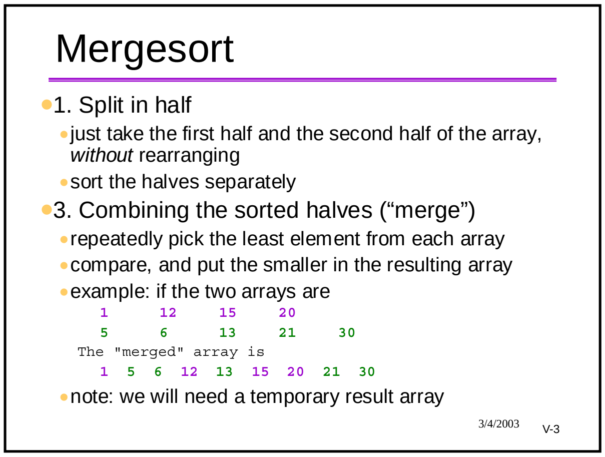


Divide & Conquer Revisited

- Quicksort illustrates “Divide and Conquer” approach:
  - 1. Divide the array into two parts, in some sensible way

    `Quicksort: "Partition"`
  - 2. Sort the two parts separately (recursively)
  - 3. Recombine the two halves easily

    `Quicksort: nothing to do at this step`
- Mergesort takes similar steps
  - 1. Divide the array
  - 2. Sort the parts recursively
  - 3. Recombine the parts

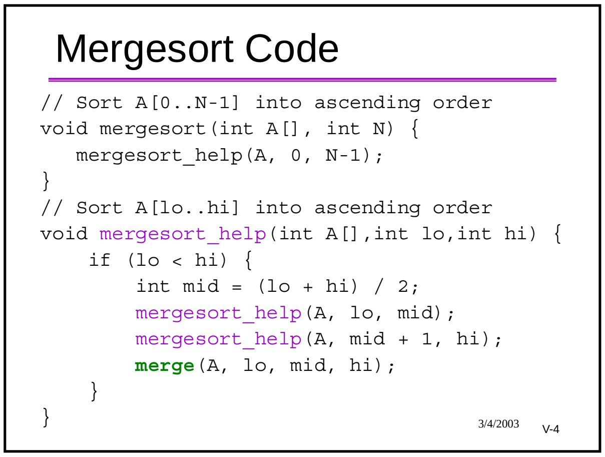


Mergesort

- 1. Split in half
  - just take the first half and the second half of the array, *without* rearranging
  - sort the halves separately
- 3. Combining the sorted halves (“merge”)
  - repeatedly pick the least element from each array
  - compare, and put the smaller in the resulting array
  - example: if the two arrays are

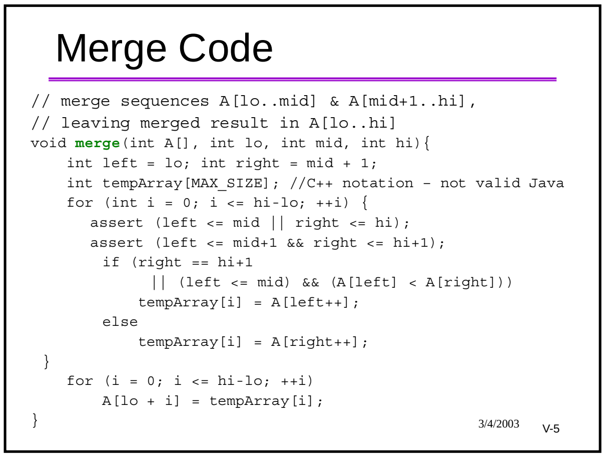
```
   1      12      15      20
   5      6       13      21      30
 The "merged" array is
   1  5  6  12  13  15  20  21  30
```

  - note: we will need a temporary result array



Mergesort Code

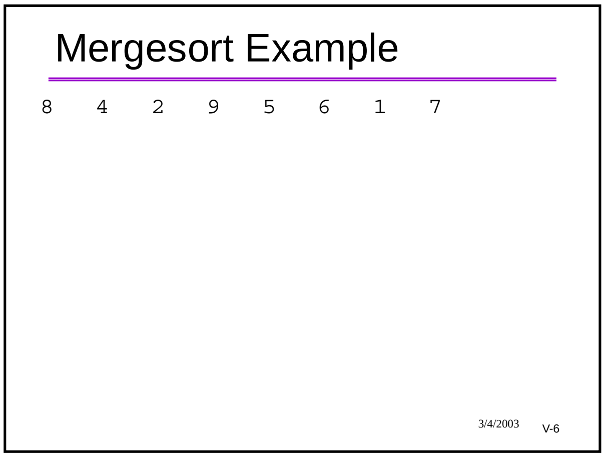
```
// Sort A[0..N-1] into ascending order
void mergesort(int A[], int N) {
   mergesort_help(A, 0, N-1);
}
// Sort A[lo..hi] into ascending order
void mergesort_help(int A[],int lo,int hi) {
    if (lo < hi) {
        int mid = (lo + hi) / 2;
        mergesort_help(A, lo, mid);
        mergesort_help(A, mid + 1, hi);
        merge(A, lo, mid, hi);
    }
}
```



Merge Code

```
// merge sequences A[lo..mid] & A[mid+1..hi],
// leaving merged result in A[lo..hi]
void merge(int A[], int lo, int mid, int hi){
    int left = lo; int right = mid + 1;
    int tempArray[MAX_SIZE]; //C++ notation – not valid Java
    for (int i = 0; i <= hi-lo; ++i) {
      assert (left <= mid || right <= hi);
      assert (left <= mid+1 && right <= hi+1);
       if (right == hi+1
            || (left <= mid) && (A[left] < A[right]))
          tempArray[i] = A[left++];
       else
          tempArray[i] = A[right++];
 }
    for (i = 0; i <= hi-lo; ++i)
       A[lo + i] = tempArray[i];
}
```

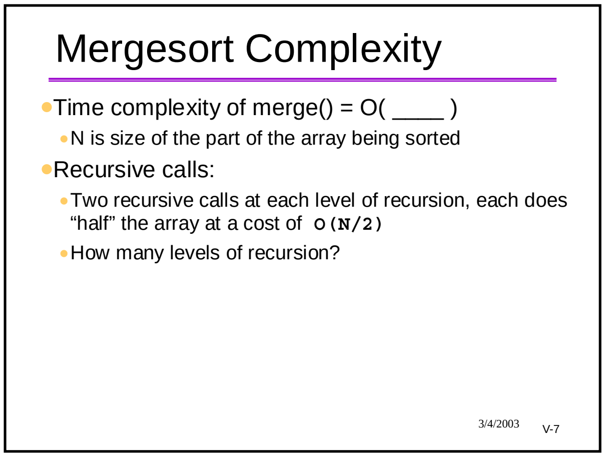
3/4/2003 V-5



Mergesort Example

8 4 2 9 5 6 1 7

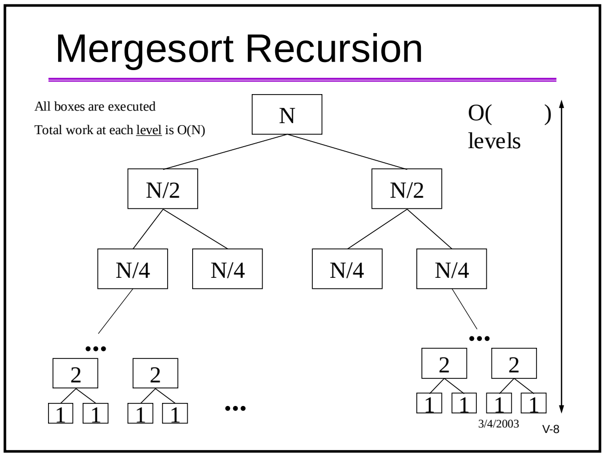
3/4/2003 V-6



Mergesort Complexity

- Time complexity of merge() = O(____)
  - N is size of the part of the array being sorted
- Recursive calls:
  - Two recursive calls at each level of recursion, each does “half” the array at a cost of `O(N/2)`
  - How many levels of recursion?

3/4/2003 V-7



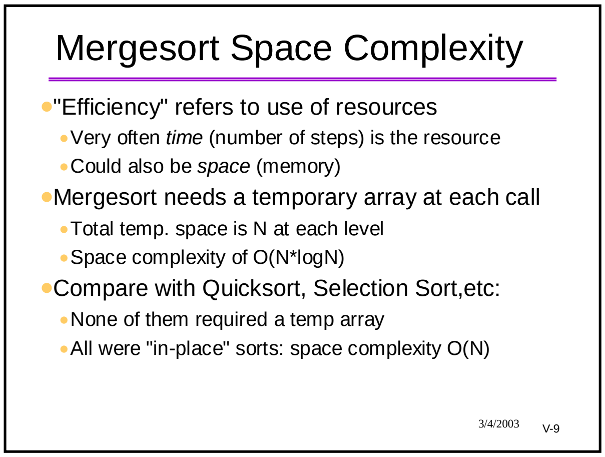
Mergesort Recursion

All boxes are executed

Total work at each level is O(N)

O( ) levels

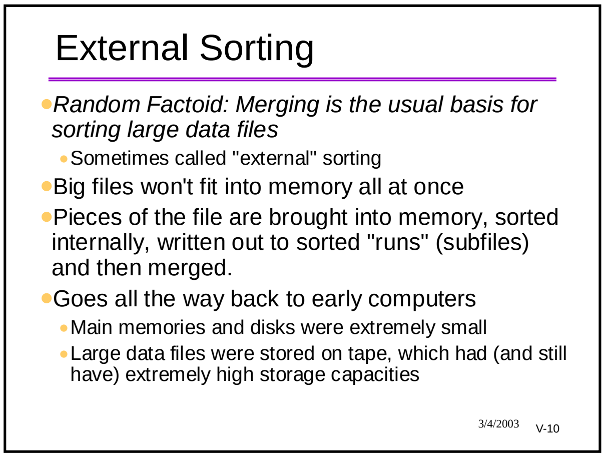
3/4/2003 V-8



Mergesort Space Complexity

- "Efficiency" refers to use of resources
  - Very often *time* (number of steps) is the resource
  - Could also be *space* (memory)
- Mergesort needs a temporary array at each call
  - Total temp. space is N at each level
  - Space complexity of O(N*logN)
- Compare with Quicksort, Selection Sort,etc:
  - None of them required a temp array
  - All were "in-place" sorts: space complexity O(N)

3/4/2003 V-9



External Sorting

- *Random Factoid: Merging is the usual basis for sorting large data files*
  - Sometimes called "external" sorting
- Big files won't fit into memory all at once
- Pieces of the file are brought into memory, sorted internally, written out to sorted "runs" (subfiles) and then merged.
- Goes all the way back to early computers
  - Main memories and disks were extremely small
  - Large data files were stored on tape, which had (and still have) extremely high storage capacities

3/4/2003 V-10