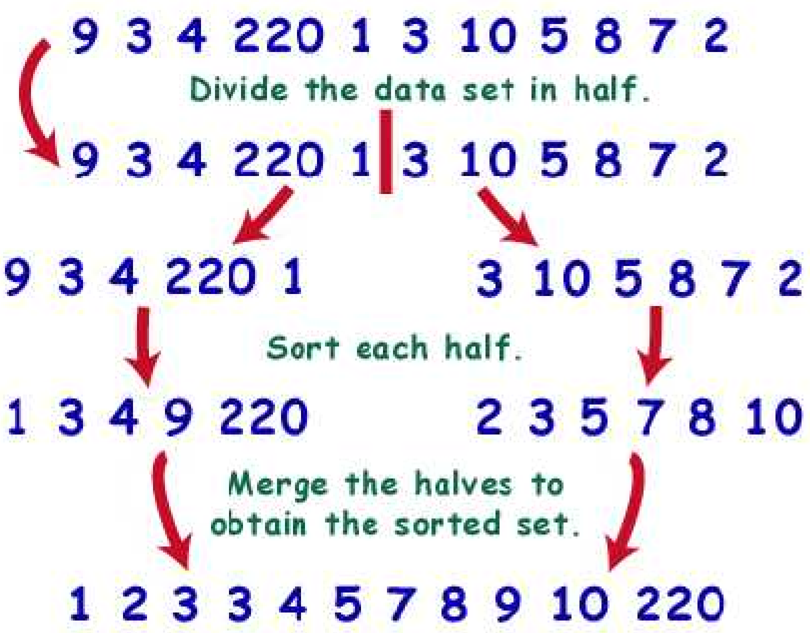
Lecture 11
Merge Sort

EECS 281:
Data Structures
and Algorithms



A Different Idea For Sorting

Quicksort is akin to dividing file into two parts

- *k smallest elements*
- *n - k largest elements*

Mergesort is akin to combining two ordered files to make one larger ordered file

Comparing Quicksort to Mergesort

```
Algorithm quicksort(array)
    partition(array)
    quicksort(lefthalf)
    quicksort(righthalf)
```

```
Algorithm mergesort(array)
    mergesort(lefthalf)
    mergesort(righthalf)
    merge(lefthalf,    righthalf)
```

- Much in common
- Top-down "management" approach
  - divide work
  - combine work
- Nothing gets done unless employees with no subordinates get work done

Important Concerns For Sorting

External Sort

- File to be sorted is on tape or disk
- Items are accessed sequentially or in large blocks

Memory Efficiency

- Sort in place with no extra memory
- Sort in place, but have pointers to or indices
  (*N* items need an additional *N* pointers or indices)
- *Need enough extra memory for an additional copy of items to be sorted*

C++ Shorthand (used in next slide)

```
c[k] = (a[i] <= b[j]) ? a[i++] : b[j++];
```

is the same thing as

```
if (a[i] <= b[j]) {
    c[k] = a[i];
    i++;
} // if
else {
    c[k] = b[j];
    j++;
} // else
```

Merging Sorted Ranges

```
void mergeAB(Item c[], Item a[], int size_a, Item b[], int size_b) {
  for (int i = 0, j = 0, k = 0; k < size_a + size_b; k++) {
    if (i == size_a)
      c[k] = b[j++];
    elseif (j == size_b)
      c[k] = a[i++];
    else
      c[k] = (a[i] <= b[j]) ? a[i++] : b[j++];
  } // for
} // mergeAB()
```

- Builds **c[]** by:
  - appending smallest remaining item from **a[]** or **b[]** onto **c[]**
  - until all items from both **a[]** and **b[]** are in **c[]**
- Θ(**size_a** + **size_b**) time for both arrays and linked lists (assuming **a[]** and **b[]** are sorted)

Example of `mergeAB()`

a

0	1	2	3
A	D	G	H

b

0	1	2
B	D	F

size_a = 4

size_b = 3

```
k = 0: (a[i = 0] = A) <= (b[j = 0] = B) ? Yes, i = 1
k = 1: (a[i = 1] = D) <= (b[j = 0] = B) ? No,  j = 1
k = 2: (a[i = 1] = D) <= (b[j = 1] = D) ? Yes, i = 2
k = 3: (a[i = 2] = G) <= (b[j = 1] = D) ? No,  j = 2
k = 4: (a[i = 2] = G) <= (b[j = 2] = F) ? No,  j = 3
k = 5: (j = 3) == size_b                ? Yes, i = 3
k = 6: (j = 3) == size_b                ? Yes, i = 4
```

c

0	1	2	3	4	5	6
A	B	D	D	F	G	H

Topdown Mergesort (Recursive)

```
void mergesort(Item a[], int left, int right) {
   if (right<= left)
      return;
   int mid = (right + left) / 2;
   mergesort(a, left, mid);          // [left,mid]
   mergesort(a, mid + 1, right);  // [mid+1,right]
   merge(a, left, mid, right);
} // mergesort()
```

- Prototypical combine and conquer algorithm
- Recursively call until sorting array of size 0 or 1
- Then merge sorted lists larger and larger
- Is it OK to use recursion here?

Modified `merge()`

```
void merge(Item a[], int left, int mid, int right) {
    int size = right - left + 1;
    vector<Item> c(size);

    for (int i = left, j = mid + 1, k = 0; k < size; ++k) {
        if (i > mid)
            c[k] = a[j++];
        else if (j > right)
            c[k] = a[i++];
        else
            c[k] = (a[i] <= a[j]) ? a[i++] : a[j++];
    } // while

    copy(c.begin(), c.end(), &a[left]);
} // merge()
```

Topdown Mergesort

Advantages (compare to Quicksort)

- Fast: $O(n \log n)$
- Stable (if merge is stable)
- Normally implemented to access data sequentially
  - does not require random access
  - great for linked lists, external-memory and parallel sorting

Topdown Mergesort

Disadvantages

- Best case performance $\Omega(n \log n)$ is slower than some elementary sorts
  - Insensitive to input
- $\Theta(n)$ additional memory, while Quicksort is in-place
  - Also extra data movement to/from copy
- Slower than Quicksort on typical inputs

Bottom-up Mergesort

```
void mergesortBU(Item a[], int left, int right) {
        for (int size = 1; size <= right - left; size = size + size)
            for (int i = left; i <= right- size; i += size + size)
        merge(a, i, i + size - 1, min(i + size + size - 1, right));
} // mergesortBU()
```

The `min()` template is defined by STL

Prototypical ‘combine and conquer’ algorithm

- view original file as N ordered sublists of size 1
- scan through list performing 1-by-1 merges to produce N/2 ordered sublists of 2
- scan through list performing 2-by-2 merges to produce N/4 ordered sublists of 4…

Job Interview Question

- In a file with 100M elements, how would you find the most frequent element ?

Questions for Self-study

- Can merge-sort (both versions) be implemented on linked lists?
  - How will this affect runtime complexity?
  - Can the merge step be done in-place?
- Show that both merge-sorts are stable iff the merge step is stable
- Why is the best-case complexity of merge-sort worse than linear?
  - How can it be improved?