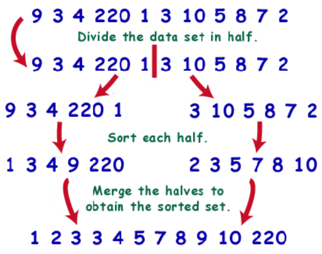
Lecture 12
Merge Sort

EECS 281:
Data Structures
and Algorithms



A Different Idea For Sorting

- *Quicksort* works by dividing a file into two parts
  – *k* smallest elements
  – *n - k* largest elements
- *Merge Sort* combines two ordered files to make one larger ordered file

Comparing Quicksort to Merge sort

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

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

- Much in common
- Top-down approach
  – Divide work
  – Combine work
- Nothing gets done until recursive calls 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++ Syntax: Ternary Operator

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

is equivalent to:

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

1. conditional
2. do if true
3. do if false

Merging Sorted Ranges

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

- Append smallest remaining item from **a** or **b** onto **c** until all items are in **c**
- Θ(**size_a + size_b**) time for both arrays and linked lists

Example of `mergeAB()`

a

A	D	G	H
0	1	2	3

b

B	D	F
0	1	2

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

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

Top-down Merge Sort (Recursive)

```
void merge_sort(Item a[], size_t left, size_t right) {
  if (right < left + 2)  // base case: 0 or 1 item(s)
    return;
  size_t mid = left + (right - left) / 2;
  merge_sort(a, left, mid);  // [left, mid)
  merge_sort(a, mid, right);  // [mid, right)
  merge(a, left, mid, right);
} // merge_sort()
```

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

Modified `merge()`

```
void merge(Item a[], size_t left, size_t mid, size_t right) {
  size_t n = right - left;
  vector<Item> c(n);

  for (size_t i = left, j = mid, k = 0; k < n; ++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++];
  } // for

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


Top-down Merge Sort

Advantages (compare to Quicksort)

- Fast: $O(n \log n)$
- Stable when a stable merge is used (it's always used! See `std::stable_sort<>`)
- Normally implemented to access data sequentially
  - Does not require random access
  - Great for linked lists, external-memory and parallel sorting

Top-down Merge Sort

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 Merge Sort

```
void merge_sortBU(Item a[], size_t left, size_t right) {
  for (size_t size = 1; size <= right - left; size *= 2)
    for (size_t i = left; i <= right - size; i += 2 * size)
      merge(a, i, i + size, std::min(i + 2 * size, right));
} // merge_sortBU()
```

Prototypical 'combine and conquer' algorithm

- View original file as n ordered sublists of size 1
- Perform 1-by-1 merges to produce $n/2$ ordered sublists of size 2
- Perform 2-by-2 merges to produce $n/4$ ordered sublists of size 4
- …

Job Interview Questions

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

Q: What is the time complexity?

Q: What is the space complexity?

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?