Slide 1

Administrivia

- Slightly revised version of `InsertionSort.cpp` sample program posted, including code to optionally print intermediate results. (Could be helpful in understanding how the algorithm works.)
- Reading Quiz 1 assigned. Linked from schedule page. Due in a week. Turn in by putting a PDF or plain-text file in the folder for this assignment in the Google Drive TurnIn folder I set up for you.

Slide 2

Another Sorting Algorithm: Merge Sort

- Several broad algorithm design techniques exist. Insertion Sort uses "incremental" approach. (I'm not 100% sure what the textbook means by that!)
- Another is "divide and conquer". This approach solves problems using three basic steps:

  Split the problem into subproblems that are smaller instances of the whole problem, unless they're so small and simple that they can be solved directly. (Call these *base cases*.)

  Solve the subproblems by solving them recursively, to produce subsolutions.

  Merge the subsolutions into a solution to the whole problem.
- Mergesort is a classic example.

Slide 3

Merge Sort

- Apply divide and conquer strategy to the basic problem of sorting an array:
- Base case is an array of size 1, or 0: Nothing to do; solution is just the input array.
- Split by splitting the array at its midpoint into two subarrays. (Or as close as we can get, if array size is not even.)
- Sort the subarrays by applying the algorithm recursively.
- Merge the two sorted subarrays. Since each is in order, this is a less difficult problem than a full sort; it works much as you'd solve the problem of merging two sorted decks by hand — repeatedly look at the top cards of the two decks and move the smaller into the output deck, until one deck is used up, and then move the remaining cards from the nonempty deck.

Slide 4

Algorithm for Merge

- We could more formally specify the merge step as follows:

  Input: An array $A[p, r]$ consisting of two subarrays $A[p, q]$ and $A[q + 1, r]$, each in order.

  Output: An array $A[p, r]$ that is a permutation of the input array and that is in order.

Slide 5

Algorithm for Merge, Continued

- Algorithm to merge subarrays $A[p:q]$ and $A[q+1:r]$ to produce $A[p,r]$:

MERGE(A, p, q, r)
$n_L = q - p + 1$
$n_R = r - q$
new arrays $L[0:n_L - 1], R[0:n_R - 1]$ // Note 0-based indexing
for $i = 0$ **to** $n_L - 1$ // Copy $A[p:q]$ into L
  $L[i] = A[p+i]$
for $j = 0$ **to** $n_R - 1$ // Copy $A[q+1:r]$ into R
  $R[j] = A[q+j+1]$
$i = 0$ // smallest remaining element in L
$j = 0$ // smallest remaining element in R
$k = p$ // next in A to fill

(continued on next slide)

Slide 6

Algorithm for Merge, continued

- *(continued from previous slide)*

// As long as each of arrays L and R contains an unmerged element,
// copy smallest unmerged element back into $A[p,r]$
while $(i < n_L)$ and $(k < n_R)$
  if $(L[i] <= R[j])$
    $A[k] = L[i]$
    $i += 1$
  else
    $A[k] = R[j]$
    $j += 1$
  $k += 1$

(continued on next slide)

Slide 7

Algorithm for Merge, continued

- *(continued from previous slide)*
  // Having gone through one of L and R entirely, copy
  // remainder to end of $A[p, r]$

  while $(i < n_L)$
  $A[k] = L[i]$
  $i += 1$
  $k += 1$
  while $(j < n_R)$
  $A[k] = R[j]$
  $j += 1$
  $k += 1$

Slide 8

Correctness and Analysis of MERGE

- It's not too hard to see that if both $A[p : q]$ and $A[q + 1 : r]$ are sorted, this algorithm does what it's supposed to do, and all the loops terminate.
- With regard to execution time, if we let n be the length of $A[p, r]$:
- The loops that copy the two pieces of $A[p, r]$ into L and R together require time proportional to n.
- The remaining loops combined copy each element of the original $A[p, r]$ from either L or R back into $A[p, r]$, so these together also require time proportional to n.
- Total execution time is therefore $\Theta(n)$.

Slide 9

Algorithm for Merge Sort

- Now we can write an algorithm for the full merge sort (sorting array $A(p:r)$). Pseudocode, using MERGE as a subprogram:

  MERGE-SORT(A, p, r)
  if $p \geq r$ //zero or one element?
  return
  $q = \frac{\lfloor (p+r) \rfloor}{2}$ //midpoint of $A[p:r]$
  MERGE-SORT(A, p, q) // recursively sort $A[p, q]$
  MERGE-SORT$(A, q+1, r)$ // recursively sort $A[q+1, r]$
  // Merge $A[p, q]$ and $A[q+1, r]$ into $A[p, r]$
  MERGE(A, p, q, r)

Slide 10

Correctness of MERGE-SORT

- The textbook doesn't seem to explicitly give an argument for correctness of this algorithm. I like to reason about divide-and-conquer algorithms using the following approach, which like loop invariants is based on the same kind of reasoning used in mathematical induction:
- Does it work on the base case(s)? (Here, yes, trivially true that an array of size 0 or 1 is sorted.)
- Does the "split" part split the whole problem into problems whose solution can be combined to solve the whole problem, and if the recursive calls work, does the "merge" merge their solutions into a solution for the full problem? (Here, we already looked at why MERGE works.)
- Does the recursion eventually stop? (Here, yes — the "split" produces smaller arrays, and when array size gets below 2, the recursion stops.)

Analysis of MERGE-SORT

Slide 11

- Methods of counting operations based on loops don't really work here, since the repetition is based on recursion rather than on loops. When recursion is involved, often possible to describe its running time with a *recurrence equation* or *recurrence* and then proceed using mathematical tools for solving recurrences.
- More about this in Chapter 4, but for now . . .

Analysis of MERGE-SORT

Slide 12

- Suppose:

  The algorithm divides a problem of size n into a subproblems, each of size n/b.

  The "split" part requires time $D(n)$.

  The "combine" part requires time $C(n)$.
- Then worst-case execution time $T(n)$ can be expressed with the following recurrence relation:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n < n_0 \\ D(n) + aT(n/b) + C(n) & \text{otherwise} \end{cases}$$

Slide 13

Analysis of MERGE-SORT, Continued

- For MERGE-SORT:
- a is 2, and b is also 2. (True that if the array size is odd, strictly speaking the two subproblems are not the same size, but their size differs by at most one, so we can reasonably say "close enough".)
- Solving the base case is $\Theta(1)$ (i.e., constant time).
- $D(n)$ (the "split") is also $\Theta(1)$.
- $C(n)$ (the "merge"') is $\Theta(n)$, as discussed previously.
- Combining these, and simplifying a little more, we get:

$$T(n) = \begin{cases} c_1 & \text{if } n = 1 \\ 2T(n/2) + c_2 n & \text{if } n > 1 \end{cases}$$

Slide 14

Analysis of MERGE-SORT, Continued

- We can apply the "master theorem" presented in Chapter 4 to solve this recurrence, giving
  $$T(n) = \Theta(n \log n)$$
  (Strictly speaking, the log function here is base-2 log, but all log functions have the same order of growth, so we can be a little sloppy.)
  (If you've forgotten: $\log_2 n$ is the number m such that $2^m = n$ — e.g., $\log_2 16 = 4$.)
- Since $\log n$ grows more slowly than n, this is very much a win over insertion sort. (Compare plots for some $n \log n$ functions and some n^2 functions.)

Slide 15

Analysis of MERGE-SORT, Continued

- If it bugs you to rely on a mysterious "master theorem", there's also a more-intuitive rationale, sketched out in the textbook:
- We can draw a tree representing the recursive process; we start with the full array, split into two halves, and then continue splitting the pieces at the current level into two subarrays each.
- This produces a tree, each level of which has double the number of subproblems as the level above, of half their size. So each level of the tree requires time $\Theta(n)$ to solve.
- And how many levels are there? well, you can only do that split $\log_2 n$ times before you get size 1 or 0.

Slide 16

Implementations

- I wrote code to implement this algorithm; it's `MergeSort.cpp` under "sample programs" on the course Web site.
- Worth noting that this is likely not the most efficient implementation, since it does involve allocating two new arrays and copying the whole array at every step. Better to set up two arrays to begin with, and repeatedly sort with one as input and the other as output.
- Note also that applying the algorithm to small arrays involves a lot of function overhead for the recursive calls, so an efficient sort might switch to insertion sort or another $\Theta(n^2)$ algorithm for arrays below some threshold size.

Slide 17

Minute Essay

- FIXME
- Any questions?