The UNIVERSITY of VERMONT

***Benjamin Fenster***
CS-104 (Lee)
27 May 2004

# Sorting

I. Introduction
   a. Internal sorting = inside main memory
   b. External sorting = on disk
   c. Array sorting vs. Linked list
   d. Direct sorting = move elements themselves
   e. Indirect sorting = move pointers to the elements
   f. In-place sorting means there's no need to declare temporary space
   g. Most of CS-104 is temporary, array, direct sorting

II. Selection Sort
   a. Run-time
      i. Does not depend on the actual values.
      ii. Comparisons = $\theta(N^2)$
      iii. $(N-1)N / 2 = O(N^2)$
      iv. Swaps = $\theta(N)$
      v. No worst-average, best case. Always $N^2$, N.
   b. No need to discuss algorithm. See CS-021

III. Bubble Sort
   a. Runtime
      i. Best: O(N).
         1. Already in sorted order.
         2. N – 1 comparisons, 0 swaps.
         3. Stop after one pass through the array.
      ii. Worst: $O(N^2)$
         1. Every single pair will need to be swapped.
         2. (N – 1) N / 2 swaps
      iii. Average: $O(N^2)$
         1. Elements in some random order
         2. For simplicity, consider (N – 1) / 2 passes on average (N – 1) is max).
         3. Then the number of comparisons is about $(3/8)N^2 - N/2$
            a. Let p be the number of passes until sorted
            b. Comparisons = (N – 1) + (N – 2) + … + (N – p)
            c. When p = (N–1)/2, Comparisons = (N–1)(N/2) – (N–1–1)(N–p)/2
            d. Comparisons = $(3/8)N^2 - N/2 + 1/8$
            e. This analysis is not precise
         4. Number of swaps is about (N-1) N/4 (about half of the worst case)
         5. Runtime: $O(N^2)$
   b. Again, no need to discuss algorithm. See CS-021

IV. Insertion Sort
   a. Concept
      i. For each element, insert to the subfile on the left.
      ii. The growing subfile thus remains sorted.
   b. Runtime
      i. Number of passes is always N – 1
      ii. Comprisons, swaps depend on the order of keys
      iii. Worst: $O(N^2)$
         1. (N–1)N/2 comparisons, (N–1) N/2 swaps
         2. Elements in reverse order
      iv. Best: O(N)
         1. Already in sorted order
         2. One comparison, no swaps for each pass.
      v. Average: $O(N^2)$
         1. Random order
         2. Between best and worst case

        3. For each element, need to go halfway through the subfile

V. Shellsort
   a. Concept
      i. Extended version of Insertion Sort
      ii. Problem with the Insertion Sort is that the number of comparisons, swaps increases with each pass.
      iii. First sort every $n_1$th element, then every $n_2$th element, then ultimately every element.
      iv. Sequence of intervals $\equiv$ increment sequence
      v. The 'presorting' shortens the number of swaps in the next phase.
   b. Runtime
      i. No precise analysis (infeasible)
      ii. Shell
         1. $h_k = h_{k+1}/2 = \ldots, 64, 32, 16, 8, 2, 1$ ($n_1, \ldots$)
         2. $O(N^2)$
      iii. Knuth
         1. $h_k = 1 + 3k = 364, 121, 40, 13, 4, 1, \ldots$
         2. $O(N^{3/2})$
      iv. Sedgewick
         1. $h_k = 9(4^k) – 9(2^k + 1)$ for $k = 0, 1, 2$
         2. $h_k = 4^k – 3(2^k) + 1$ for $k = 2, 3, 4$
         3. $h_k = \ldots, 209, 109, 41, 19, 5, 1$

VI. Heapsort
   a. Concept
      i. Uses the same kind of heap as a priority queue.
      ii. BuildHeap on the array, then perform deleteMax N times, store each max at the end of the array.
   b. Runtime
      i. buildHeap O(N)
      ii. {deleteMax} O(N log N)
         1. Proof: Percolating down the root in a binary heap of k nodes takes maximum log k swaps.
         2. Total percolations $\leq \log(N – 1) + \log(N – 2) + \ldots + \log 2$
         3. $< \log N + \ldots + \log N$
         4. Total percolations = O(N log N)

VII. Mergesort
   a. Concept
      i. "Divide and conquer"
      ii. Divide file into two equal-sized subfiles
      iii. Needs a temporary space to store the merged subfiles, so this is *not* an in-place algorithm.
      iv. There is an algorithm that doesn't use temporary space, but with a cost in complexity
   b. Runtime
      i. Let T(N) be the number of comparisons for N elements.
         1. $T(N) = T(\text{floor}(N/2)) + T(\text{ceil}(N / 2) + N$
            [left] [right] [merge]
         2. $T(1) = 0$
      ii. Solution
         1. $T(N) = 2 T(n/2) + N$
         2. Let $2^n = N$
         3. $T(2^n) = 2T(2^{n-1}) + 2^n$
            …
            $T(2^n) = 2^n * n$
         4. So T(N) = O(N log N)

   iii. Runtime is always O(N log N)
   iv. No best/average/worst cases since the algorithm works without regard to order.
- c. Side Note
   i. This algorithm requires extra storage, increases linearly with the number of elements.
   ii. Thus, Mergesort is hardly ever used for internal sorting. Much better to use for external (disk) sorting.

VIII. Quicksort
- a. Concept
   i. Fastest known sorting algorithm; allows efficient implementation
   ii. Split the file into two subfiles such that all keys in one subfile are all ≤ some pivot, and all keys in the over subfile are > the pivot.
   iii. The pivot is some element chosen for that purpose.
   iv. Keep repeating until the partitions have only one element – then it's sorted!
   v. Need to choose a partitioning element (pivot), then shuffle elements to meet the property stated above
      1. Pick the rightmost element to be the pivot (for now)
      2. Find the first element from the left that belongs AFTER the pivot.
      3. Find the first element from the right that belongs BEFORE the pivot.
      4. Swap those two.
      5. Repeat until the two indices (for those elements) cross.
      6. Then swap the pivot (rightmost element) with the pointer seeking the next element greater than the pivot.
   vi. Selecting a Pivot
      1. Always choose first or last element
         a. Very fast
         b. If the file is already sorted, the algorithm will suffer – you'd never split into two subfiles. Instead you'd just keeping one element at a time off the end.
      2. Randomly select an element
         a. Good on average for being able to split into two subfiles
         b. Incurs the overhead of random number generation
      3. Choose the median of several elements
         a. Median of {left, right, center}
         b. Median of {5, 7, …}
         c. Compromise between those two strategies.
- b. Runtime
   i. Worst: $O(N^2)$
      1. This occurs when the partitioned subfiles are always 0 and N-1 elements
      2. That is, the pivot element is always the largest or smallest element
   ii. Best: O(N log N)     Subfile sizes are always N/2
   iii. Average
      1. Partitioning is linear (scans each element once)
      2. Runtime for quick quicksort is the runtime for partitioning plus that for both recursive calls
      3. T(N) = T(i) + T(N-i-1) + N for N > 1     i = size of left subfile
         T(0) = T(1) = 1
      4. (Not really T(0) but we'll be dcomplete)
      5. Worst Case: T(N) = T(0) + T(N – 1) + N = T(N – 1) + N
      6. Best Case: T(N) = 2T(N /2) + N
      7. Average:
         a. T(N) = 1/N) SUM(i=0,N-1,T(i) + 1/N) SUM(i=0,N-1,T(N-i-1) + N
            [Average of T(0), T(1), …, T(N-1)]
         b. = (2/N) SUM(i=0, N-1,T(i) + N)
      8. Worst: $O(N^2)$

9. Best: O(N log N)
10. Average
    a. Multiply by N both sides
    b. Substitute N-1 for N
    c. Get (N-1) T(N-1) =
    $$2\left(\sum_{i=0}^{N-2} T(i)\right) + (N-1)^2$$
    d. Subtract from N T(N) =
    $$2\left(\sum_{i=0}^{N-1} T(i)\right) + (N)^2$$
    e. Result is N T(N) = (N + 1) T(N – 1) + 2N
    f. T(N) / (N + 1) = (T(N – 1) / N) + (2c / N + 1)
    g. We now have an equation that can be solved the usual way
    h. T(N – 1) / N = T(N – 2) / (N – 1) + 2c / N
    T(N – 2) / N = T(N – 2) / (N – 1) + 2c / N
    …
    T(2) / 3 = T(1) / 2 + 2c / 3
    i. T(N) / N + 1) =
    $$\frac{T(N)}{N+1} = \frac{T(1)}{2} + 2c\sum_{i=3}^{n+1} 1/i$$
    j. T(N) / (N + 1) = O(log N)
    k. T(N) = O(N log N)

c. Quickselect
   i. Select the kth smallest (or largest) element
   ii. Did this already with binary heap
   iii. Now do it using an array with quickselect
   iv. Striking similarity to quicksort
   v. Concept
      1. Pick a pivot and partition the file
      2. If k is smaller than size(Lsubfile), return quickselect(Lsubfile, k)
      3. else if k == size(Lsubfile) + 1 then return the pivot
      4. else return quickselect(Rsubfile, k)
   vi. Runtime
      1. Worst: Pivot is always the rightmost element so runtime is $O(N^2)$
      2. Best: T(N) = T(N / 2) + N = O(N) subfiles are always half.
      3. Average:
      $$T(N) = \left(\frac{1}{N}\right)\sum_{i=0}^{N-1} T(i) + N = O(N)$$
      4. Only difference is 1 / N instead of 2 / N

d. Variations
   i. When file size is small (5 to 20 elements), insertion sort is more efficient so use that instead.
   ii. Once the quicksort has created a small enough subfile, switch to the insertion sort for that too

e. Compare to Mergesort
   i. Mergesort does the sorting in the ‘conquer” phase of “divide and conquer” algorithm (i.e. while falling out of recursion)
   ii. Quicksort does it during the “divide” phase (i.e. on the way into the recursion)