Sorting by Comparison

1. Simple: SelectionSort, BubbleSort
2. Good worst case: MergeSort, HeapSort
3. Good average case: QuickSort
4. Can we do better?

Selection Sort Idea

- Are first 2 elements sorted? If not, swap.
- Are the first 3 elements sorted? If not, move the 3rd element to the left by series of swaps.
- Are the first 4 elements sorted? If not, move the 4th element to the left by series of swaps.
  – etc.

Selection Sort

\[
\text{procedure SelectionSort (Array[1..N])}
\]
\[
\text{For (i=2 to N) { }
  \text{j = i; }
  \text{while (j > 0 && Array[j] < Array[j-1]) { }
    \text{swap (Array[j], Array[j-1])}
    \text{j --;}
  \text{}}
\]

Suppose Array is initially sorted? \(O(n)\)
Suppose Array is reverse sorted? \(O(n^2)\)

Bubble Sort Idea

Slightly rearranged version of selection sort:

- Move smallest element in range 1,……,n to position 1 by a series of swaps
- Move smallest element in range 2,……,n to position 2 by a series of swaps
- Move smallest element in range 3,……,n to position 3 by a series of swaps
  – etc.
Why Selection (or Bubble) Sort is Slow

- **Inversion**: a pair (i,j) such that i < j but Array[i] > Array[j]
- Array of size N can have \( \Theta(N^2) \) inversions
  - average number of inversions in a random set of elements is \( N(N-1)/4 \)
- Selection/Bubble Sort only swaps adjacent elements
  - only removes 1 inversion!

HeapSort: sorting with a priority queue ADT (heap)

Worst Case: \( O(n \log n) \)

Best Case: \( O(n \log n) \)

Shove everything into a queue, take them out smallest to largest.

HeapSort: sorting with a priority queue ADT (heap)

Worst Case: \( O(n \log n) \)

Best Case: \( O(n \log n) \)

Shove everything into a queue, take them out smallest to largest.

MergeSort Running Time

Any difference best/worse case?

\[
T(1) \leq b \\
T(n) \leq 2T(n/2) + cn \quad \text{for } n>1 \\
T(n) \leq 2T(n/2) + cn \leq 2(2T(n/4)+cn/2)+cn \\
= 4T(n/4) + cn = 4(2T(n/8)+cn/4)+cn+cn \\
= 8T(n/8)+cn+cn \leq 2(2T(n/8)) = \text{inductive leap} \\
\leq \ldots nT(1)+cn \log n \text{ where } k = \log n \\
= O(n \log n)
\]

QuickSort

Pick a “pivot”. Divide into less-than & greater-than pivot. Sort each side recursively.
QuickSort Partition

Pick pivot: 7 2 8 3 5 9 6

Partition with cursors:

2 goes to less-than:

QuickSort Partition (cont’d)

6, 8 swap less/greater-than

3, 5 less-than 9 greater-than

Partition done. Recursively sort each side.

Analyzing QuickSort

- Picking pivot: constant time
- Partitioning: linear time
- Recursion: time for sorting left partition (say of size i) + time for right (size N-i-1)

\[
T(1) = b \\
T(N) = T(i) + T(N-i-1) + cN \\
\text{where } i \text{ is the number of elements smaller than the pivot}
\]

Dealing with Slow QuickSorts

- Randomly choose pivot
  - Good theoretically and practically, but call to random number generator can be expensive
- Pick pivot cleverly
  - “Median-of-3” rule takes Median(first, middle, last element elements). Also works well.
QuickSort

Best Case

Pivot is always middle element.
\[ T(N) = T(i) + T(N-i-1) + cN \]
\[ T(N) = 2T(N/2 - 1) + cN \]
\[ < 2T(N/2) + cN \]
\[ < 4T(N/4) + c(2N/2 + N) \]
\[ < 8T(N/8) + cN(1 + 1 + 1) \]
\[ < kT(N/k) + cN\log(k) = O(N\log N) \]

QuickSort

Average Case

- Assume all size partitions equally likely, with probability 1/N

\[ T(N) = T(i) + T(N-i-1) + cN \]
average value of \( T(i) \) or \( T(N-i-1) \) is \( \frac{1}{2} \sum_{j=0}^{i-1} T(j) \)
\[ T(N) = \left( \frac{1}{2} \sum_{j=0}^{i-1} T(j) \right) + cN \]
\[ = O(N\log N) \]

details: Weiss pg 276-279

Could We Do Better?*

- For any possible correct Sorting by Comparison algorithm…
  - What is lowest best case time?
  - What is lowest worst case time?

* (no. sorry.)

Best case time

- How many comparisons does it take before we can be sure of the order?
- This is the minimum # of comparisons that any algorithm could do.

Decision tree to sort list A,B,C

Legend
- Internal node, with facts known so far
- Leaf node, with ordering of A,B,C
- Edge, with result of one comparison
Max depth of the decision tree

- How many permutations are there of $N$ numbers?
- How many leaves does the tree have?
- What’s the shallowest tree with a given number of leaves?
- What is therefore the worst running time (number of comparisons) by the best possible sorting algorithm?

Stirling’s approximation

\[ n! \approx \sqrt{2\pi n} \left( \frac{n}{e} \right)^n \]

\[ \log(n!) = \log\left( \sqrt{2\pi n} \left( \frac{n}{e} \right)^n \right) \]

\[ = \log(\sqrt{2\pi n}) + \log\left( \left( \frac{n}{e} \right)^n \right) = \Omega(n \log n) \]

MergeSort Good for Something!

- Basis for most external sorting routines
- Can sort any number of records using a tiny amount of main memory
  - in extreme case, only need to keep 2 records in memory at any one time!

Not enough RAM – External Sorting

- E.g.: Sort 10 billion numbers with 1 MB of RAM.
- Databases need to be very good at this

External MergeSort

- Split input into two tapes
- Each group of 1 records is sorted by definition, so merge groups of 1 to groups of 2, again split between two tapes
- Merge groups of 2 into groups of 4
- Repeat until data entirely sorted

\[ \log N \text{ passes} \]
Better External MergeSort

- Suppose main memory can hold M records.
- Initially read in groups of M records and sort them (e.g. with QuickSort).
- Number of passes reduced to log(N/M)

Summary

- Sorting algorithms that only compare adjacent elements are \( \Theta(N^2) \) worst case – but may be \( \Theta(N) \) best case
- HeapSort and MergeSort - \( \Theta(N \log N) \) both best and worst case
- QuickSort \( \Theta(N^2) \) worst case but \( \Theta(N \log N) \) best and average case
- Any comparison-based sorting algorithm is \( \Omega(N \log N) \) worst case
- External sorting: MergeSort with \( \Theta(\log N/M) \) passes