Quick Sort

CSE 373 - Data Structures
May 15, 2002

Readings and References

• Reading
  › Section 7.7, Data Structures and Algorithm Analysis in C, Weiss

• Other References
  › C LR

Sorting Ideas - swap adjacent

• Swap adjacent elements
  › Bubble sort
    • it works, but it's always slow
  › Insertion sort
    • works well on already sorted or partially sorted input
    • low overhead so it works well on small inputs or as the basic sorter for a larger algorithm

Sorting Ideas - swap non-adjacent

• Swap non-adjacent elements
  › Shell sort
    • resolves multiple inversions with a single swap
    • does an insertion sort of variable sized sub-arrays
    • choice of increments critical
  › Heap sort
    • resolves multiple inversions with a single swap
    • does insertion sort of paths through a binary heap
Sorting Ideas - recursion and merge

- Merging two sorted arrays is fast
  › Partition the array and sort each part separately, then merge the results
  › The merge can resolve many inversions with each element merged
- Merge sort
  › Fast
  › requires extra O(N) temporary array

Sorting Ideas - recursion and join

- Joining two sorted arrays can be very fast
  › Partition the array into a set of little elements and a set of big elements, sort each partition, and join them
  › The partitioning operation can move elements a long way towards the final location in one move
- Quick Sort
  › Fast
  › in-place sort (no extra space required)

Quicksort

- Quicksort uses a divide and conquer strategy, but does not require the O(N) extra space that MergeSort does
  › Partition array into left and right sub-arrays
    • the elements in left sub-array are all less than pivot
    • elements in right sub-array are all greater than pivot
  › Recursively sort left and right sub-arrays
  › Concatenate left and right sub-arrays in O(1) time

“Four easy steps”

- To sort an array S
  › If the number of elements in S is 0 or 1, then return. The array is sorted.
  › Pick an element v in S. This is the pivot value.
  › Partition S-{v} into two disjoint subsets, S_1 = {all values x≤v}, and S_2 = {all values x≥v}.
  › Return QuickSort(S_1), v, QuickSort(S_2)
The steps of QuickSort

<table>
<thead>
<tr>
<th>S</th>
<th>select pivot value</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>81</td>
</tr>
<tr>
<td>S₁</td>
<td>13</td>
</tr>
<tr>
<td>S₂</td>
<td>81</td>
</tr>
</tbody>
</table>

partition S

QuickSort(S₁) and QuickSort(S₂)

Presto! S is sorted

Quicksort Example

- Sort the array containing:

  9   16   4   15   2   5   17   1

  partition:

  2   5   1   9   16   15   17

  QuickSort(S₁) and QuickSort(S₂)

  Concatenate:

  2   5   1   9   16   15   17

Details, details

- "The algorithm so far lacks quite a few of the details"
- Implementing the actual partitioning
- Picking the pivot
  - want a value that will cause |S₁| and |S₂| to be non-zero, and close to equal in size if possible
- Dealing with cases where the element equals the pivot

Quicksort Partitioning

- Need to partition the array into left and right sub-arrays
  - the elements in left sub-array are ≤ pivot
  - elements in right sub-array are ≥ pivot
- How do the elements get to the correct partition?
  - Choose an element from the array as the pivot
  - Make one pass through the rest of the array and swap as needed to put elements in partitions
Partitioning is done In-Place

- One implementation (there are others)
  - median3 finds pivot and sorts left, center, right
  - Swap pivot with next to last element
  - Set pointers i and j to start and end of array
  - Increment i until you hit element A[i] > pivot
  - Decrement j until you hit element A[j] < pivot
  - Swap A[i] and A[j]
  - Repeat until i and j cross
  - Swap pivot (= A[N-2]) with A[i]

Choosing the Pivot (1)

- First (bad) Idea
  - Pick the first element as pivot
  - What if it is the smallest or largest?
  - What if the array is sorted?
    - How many recursive calls does quicksort make?
      - O(N) calls, and it does O(N) work for each call, so you do O(N^2) work when the array is already sorted!
Choosing the Pivot (2)

- 2\textsuperscript{nd} (okay) Idea:
  - Pick a \textit{random} element to be the pivot
  - Gets rid of asymmetry in left/right sizes
  - Actually works pretty well
  - But it requires calls to pseudo-random number generator
    - expensive in terms of time
    - many implementations are not particularly random

Choosing the Pivot (3a)

- Third idea
  - Pick \textit{median} element (N/2\textsuperscript{th} largest element)
  - This is great … it splits \textit{S} exactly in two
  - But it’s hard to find the median element without sorting the entire array first, which is why we are here in the first place ...

Choosing the Pivot (3b)

- Find the median of the first, middle and last elements - “median of 3”
- If the data in the array is not sorted, median of 3 is similar to picking a random pivot
- If the data in the array is presorted, this will pick a value near the actual median of the entire array, which is good

Median-of-Three Pivot

- Find the median of the first, middle and last element
  \begin{array}{c}
    2 \quad 4 \quad 9 \quad 15 \quad 16 \\
    \downarrow \\
    9
  \end{array}
  \quad
  \begin{array}{c}
    5 \quad 4 \quad 2 \quad 15 \quad 16 \\
    \downarrow \\
    5
  \end{array}

- Takes only O(1) time and not error-prone like the pseudo-random pivot choice
- Less chance of poor performance as compared to looking at only 1 element
- For sorted inputs, splits array nicely in half each recursion
A[i]==pivot?

- Stop and swap
  - while(A[++i]<pivot){}
  - while(A[--j]>pivot){}

- Although this seems a little odd, it moves i and j towards the middle
  - the benefit of balanced partitions when i and j cross in the middle outweighs the extra cost of swapping elements that are equal to the pivot

---

Quicksort Best Case Performance

- Algorithm always chooses best pivot and splits sub-arrays in half at each recursion
  - T(0) = T(1) = O(1)
    - constant time if 0 or 1 element
  - For N > 1, 2 recursive calls plus linear time for partitioning
    - T(N) = 2T(N/2) + O(N)
      - Same recurrence relation as Mergesort
    - T(N) = O(N log N)

---

Quicksort Worst Case Performance

- Algorithm always chooses the worst pivot – one sub-array is empty at each recursion
  - T(0) = T(1) = O(1)
  - T(N) = T(N-1) + O(N)
  - = T(N-2) + O(N-1) + O(N)
  - = T(0) + O(1) + ... + O(N)
  - T(N) = O(N^2)

- Fortunately, *average case performance* is O(N log N) (see text for proof)