Divide and Conquer Sorting

CSE 373
Data Structures
Lecture 14

Readings

- Reading
  - Section 7.6, Mergesort
  - Section 7.7, Quicksort

“Divide and Conquer”

- Very important strategy in computer science:
  - Divide problem into smaller parts
  - Independently solve the parts
  - Combine these solutions to get overall solution
- Idea 1: Divide array into two halves, recursively sort left and right halves, then merge two halves known as Mergesort
- Idea 2: Partition array into small items and large items, then recursively sort the two sets known as Quicksort

Mergesort

- Divide it in two at the midpoint
- Conquer each side in turn (by recursively sorting)
- Merge two halve together

Mergesort Example

Auxiliary Array

- The merging requires an auxiliary array.
Auxiliary Array

- The merging requires an auxiliary array.

2 4 8 9 1 3 5 6

Auxiliary array

1 2 3 4 5

Auxiliary array

Merging

Merge(A[], T[], left, right, target) :
mid := (left + right) \div 2;
i := left; j := mid + 1; target := left;
while i ≤ mid and j ≤ right do
else T[target] := A[j]; j := j + 1;
target := target + 1;
if i > mid then //left completed/
for k := left to target do A[k] := T[k];
if j > right then //right completed/
k := mid; l := right;
while k ≤ l do A[k] := A[k]; k := k+1; l := l-1;
for k := left to target do A[k] := T[k];

Recursive Mergesort

Mergesort(A[], T[], left, right, target) :
mid := (left + right) \div 2;
Mergesort(A[], T[], left, mid);
Mergesort(A[], T[], mid+1, right);
Mergesort(A[], T[], left, right);
MainMergesort(A[], 1, n);
Iterative Mergesort

IterativeMergesort(A[1..n], i, j, k, n; integer array, n: integer) i { precondition: n is a power of 2; / \ i, k, parity: integer /
T[i..n]: integer array /

k := 2; parity := 0 /
while k <= n do /
  for i := 1 to n do A[i] := T[i]; /
  if parity = 0 then Merge(A, i, j, i, j + k - 1); /
  else Merge(A, j + k, i, n); /
  parity := 1 - parity;
  k := 2 * k;
  if parity = 1 then /
  for i := 1 to n do A[i] := T[i]; |

How do you handle non-powers of 2?
How can the final copy be avoided?

Mergesort Analysis

- Let T(N) be the running time for an array of N elements
- Mergesort divides array in half and calls itself on the two halves. After returning, it merges both halves using a temporary array
- Each recursive call takes T(N/2) and merging takes O(N)

Mergesort Recurrence Relation

- The recurrence relation for T(N) is:
  - T(1) ≤ α
    - base case: 1 element array → constant time
  - T(N) ≤ 2T(N/2) + bN
    - Sorting N elements takes
      - the time to sort the left half
      - plus the time to sort the right half
      - plus an O(N) time to merge the two halves
  - T(N) = O(n log n)

Properties of Mergesort

- Not in-place
  - Requires an auxiliary array
- Stable
  - Make sure that left is sent to target on equal values.
- Very few comparisons
  - Iterative Mergesort reduces copying.
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 < v}, and S_2 = {all values ≥ v}.
  - Return QuickSort(S_1), v, QuickSort(S_2)

The steps of QuickSort

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 |S_1| and |S_2| 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]
Example

Choose the pivot as the median of three.
Place the pivot and the largest at the right and the smallest at the left.

Example

Move i to the right to be larger than pivot.
Move j to the left to be smaller than pivot.
Swap.

Recursive Quicksort

Quicksort(A[]): integer array, left, right : integers; |
pivotindex : integer; |
if left ≤ CUTOFF ≤ right then |
pivot := median3(A, left, right); |
pivotindex := Partition(A, left, right-1, pivot); |
Quicksort(A, left, pivotindex - 1); |
Quicksort(A, pivotindex + 1, right); |
else |
Insertionsort(A, left, right); |

Don’t use quicksort for small arrays.
CUTOFF = 10 is reasonable.

Alternative Pivot Rules

• Chose A[left] |
  › Fast, but may be too biased
• Chose A[random], left ≤ random ≤ right |
  › Completely unbiased
  › Will cause relatively even split, but slow
• Median of three, A[left], A[right], A[left+right]/2 |
  › The standard, tends to be unbiased, and does a little sorting on the side.

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
  - $T(N) \leq a$ for $N \leq C$
  - $T(N) \leq T(N-1) + bN$
  - $T(N) \leq T(N-2) + b(N-1) + bN$
  - $T(N) \leq T(C) + b(C+1) + \ldots + bN$
  - $T(N) \leq a + b(C + C+1 + C+2 + \ldots + N)$
  - $T(N) = O(N^2)$
- Fortunately, average case performance is $O(N \log N)$ (see text for proof)

Properties of QuickSort

- Not stable because of long distance swapping.
- No iterative version (without using a stack).
- Pure quicksort not good for small arrays.
- "In-place", but uses auxiliary storage because of recursive calls.
- $O(n \log n)$ average case performance, but $O(n^2)$ worst case performance.

Folklore

- "QuickSort is the best in-memory sorting algorithm."
- Truth
  - QuickSort uses very few comparisons on average.
  - QuickSort does have good performance in the memory hierarchy.
    - Small footprint
    - Good locality