

CSE 332: Data Structures and Parallelism

Spring 2022
Richard Anderson
Lecture 13: Sorting I

Announcements

Sorting

- Input
 - an array A of data records
 - a key value in each data record
 - a comparison function which imposes a consistent ordering on the keys
- Output
 - “sorted” array A such that
 - For any i and j , if $i < j$ then $A[i] \leq A[j]$

Space

- How much space does the sorting algorithm require?
 - In-place: no more than the array or at most $O(1)$ addition space
 - out-of-place: use separate data structures, copy back
 - External memory sorting – data so large that does not fit in memory

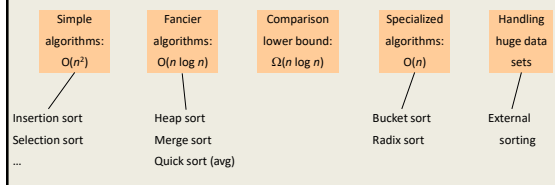
Time

How fast is the algorithm?

- requirement: for any $i < j$, $A[i] \leq A[j]$
- This means that you need to at least check on each element at the very minimum
 - Complexity is at least:
- And you could end up checking each element against every other element
 - Complexity could be as bad as:

The big question: How close to $O(n)$ can you get?

Sorting: *The Big Picture*



Insertion Sort

1. Sort first 2 elements.
2. Insert 3rd element in order.
 - (First 3 elements are now sorted.)
3. Insert 4th element in order
 - (First 4 elements are now sorted.)
4. And so on...

4/25/2022

CSE 332

7

How to do the insertion?

Suppose my sequence is:

16, 31, 54, 78, 32, 17, 6

And I've already sorted up to 78. How to insert 32?

4/25/2022

CSE 332

8

Try it out: Insertion sort

- 31, 16, 54, 4, 2, 17, 6

4/25/2022

CSE 332

9

Insertion Sort: Code

```
void InsertionSort (Array a[0..n-1]) {
    for (i=1; i<n; i++) {
        for (j=i; j>0; j--) {
            if (a[j] < a[j-1])
                Swap(a[j],a[j-1])
            else
                break
        }
    }
}
```

4/25/2022

CSE 332

10

Insertion Sort

- Worst case $O(n^2)$
- It can be shown that the runtime is related to how sorted the data is
 - Run time proportional to number of pairs of out of order items
- Insertion sort is useful when there is a small number of items to sort

4/25/2022

CSE 332

11

Heap Sort: Sort with a Binary Heap

Worst Case Runtime:

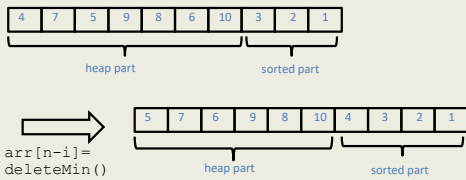
4/25/2022

CSE 332

12

In-place heap sort

- Treat the initial array as a heap (via **buildHeap**)
- When you delete the i^{th} element, put it at **arr[n-i]**
 - It's not part of the heap anymore!



4/25/2022

CSE 332

13

“Divide and Conquer”

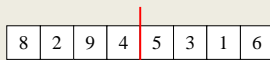
- Very important strategy in computer science:
 - Divide problem into smaller parts
 - Independently solve the parts
 - Combine these solutions to get overall solution
- **Idea 1:** Divide array in half, *recursively* sort left and right halves, then *merge* two halves
→ known as **Mergesort**
- **Idea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

4/25/2022

CSE 332

14

Mergesort



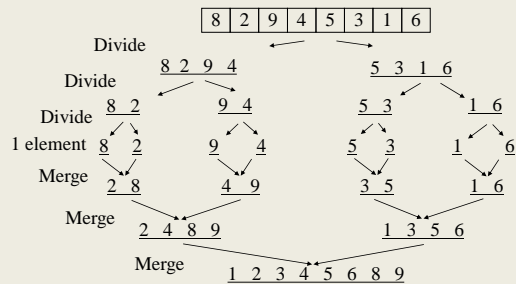
- Divide it in two at the midpoint
- Sort each half (recursively)
- Merge two halves together

4/25/2022

CSE 332

15

Mergesort Example



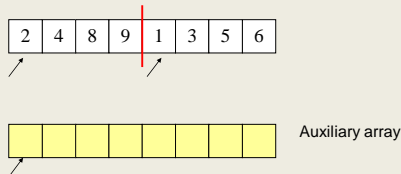
4/25/2022

CSE 332

16

Merging: Two Pointer Method

- Perform merge using an auxiliary array



4/25/2022

CSE 332

17

Merging

```

Merge(A[], Temp[], left, mid, right) {
    Int i, j, k, l, target
    i = left
    j = mid + 1
    target = left
    while (i < mid && j <= right) {
        if (A[i] <= A[j])
            Temp[target] = A[i++]
        else
            Temp[target] = A[j++]
        target++
    }
    if (i > mid) //left completed//
        for (k = left to target-1)
            A[k] = Temp[k];
    if (j > right) //right completed//
        k = mid
        l = right
        while (k >= l)
            A[l--] = A[k--]
        for (k = left to target-1)
            A[k] = Temp[k]
    }
    }
    
```

4/25/2022

CSE 332

18

Recursive Mergesort

```

MainMergesort(A[1..n], n) {
  Array Temp[1..n]
  Mergesort[A, Temp, 1, n]
}

Mergesort(A[], Temp[], left, right) {
  if (left < right) {
    mid = (left + right)/2
    Mergesort(A, Temp, left, mid)
    Mergesort(A, Temp, mid+1, right)
    Merge(A, Temp, left, mid, right)
  }
}
    
```

What is the recurrence relation?

4/25/2022

CSE 332

19

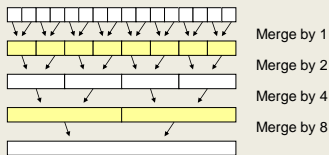
Mergesort: Complexity

4/25/2022

CSE 332

20

Iterative Mergesort



4/25/2022

CSE 332

21

Properties of Mergesort

- In-place?
- Sorted list complexity?
- Nicely extends to handle linked lists.
- Multi-way merge is basis of big data sorting.
- Java uses Mergesort on Collections and on Arrays of Objects.

4/25/2022

CSE 332

22

Quicksort

Quicksort uses a divide and conquer strategy, but does not require the $O(N)$ extra space that MergeSort does.

Here's the idea for sorting array S :

1. Pick an element v in S . This is the *pivot* value.
2. Partition $S - \{v\}$ into two disjoint subsets, S_1 and S_2 such that:
 - elements in S_1 are all $\leq v$
 - elements in S_2 are all $\geq v$
3. Return concatenation of $\text{QuickSort}(S_1)$, v , $\text{QuickSort}(S_2)$

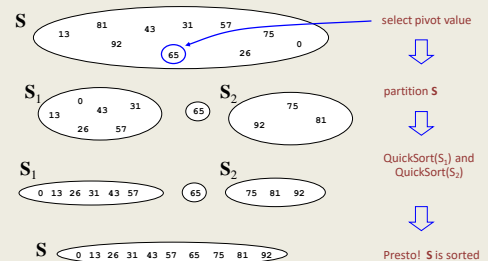
Recursion ends if $\text{QuickSort}()$ receives an array of length 0 or 1.

4/25/2022

CSE 332

23

The steps of Quicksort

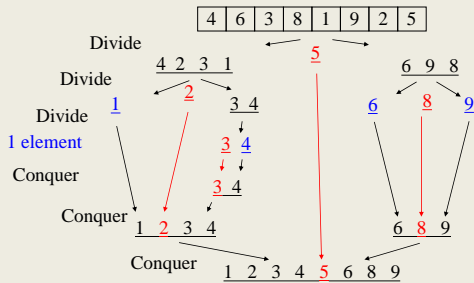


4/25/2022

CSE 332

24

Quicksort Example



4/25/2022

CSE 332

25

Pivot Picking and Partitioning

The tricky parts are:

- **Picking the pivot**
 - Goal: pick a pivot value so that $|S_1|$ and $|S_2|$ are roughly equal in size.
- **Partitioning**
 - Preferably in-place
 - Dealing with duplicates

4/25/2022

CSE 332

26

Picking the pivot

- Choose the first element in the subarray
- Choose a value that might be close to the middle
 - Median of three
- Choose a random element

4/25/2022

CSE 332

27

Quicksort Partitioning

- Partition the array into left and right sub-arrays such that:
 - elements in left sub-array are \leq pivot
 - elements in right sub-array are \geq pivot
- Can be done in-place with another “two pointer method”
 - Sounds like mergesort, but here we are *partitioning*, not sorting...
 - ...and we can do it in-place.
- Lots of work has been invested in engineering quicksort

4/25/2022

CSE 332

28

Quicksort Pseudocode

Putting the pieces together:

```

Quicksort(A[], left, right) {
  if (left < right) {
    medianOf3Pivot(A, left, right);
    pivotIndex = Partition(A, left+1, right-1);

    Quicksort(A, left, pivotIndex - 1);
    Quicksort(A, pivotIndex + 1, right);
  }
}
    
```

4/25/2022

CSE 332

29

Important Tweak

Insertion sort is actually better than quicksort on small arrays. Thus, a better version of quicksort:

```

Quicksort(A[], left, right) {
  if (right - left >= CUTOFF) {
    medianOf3Pivot(A, left, right);
    pivotIndex = Partition(A, left+1, right-1);

    Quicksort(A, left, pivotIndex - 1);
    Quicksort(A, pivotIndex + 1, right);
  } else {
    InsertionSort(A, left, right);
  }
}
    
```

CUTOFF = 16 is reasonable.

4/25/2022

CSE 332

30

Quicksort run time

- What is the best case behavior?

4/25/2022

CSE 332

31

Worst case run time

- What is the bad case for partitioning?
- Design a bad case input (assume first element is chosen as pivot)

4/25/2022

CSE 332

32

Average case performance

- Assume all permutations of the data are equally likely
 - Or equivalently, a random pivot is chosen
- The math gets messy, but doable

4/25/2022

CSE 332

33

Properties of Quicksort

- $O(N^2)$ worst case performance, but $O(N \log N)$ average case performance.
- Pure quicksort not good for small arrays.
- No iterative version (without using a stack).
- “In-place,” but uses auxiliary storage because of recursive calls.
- Used by Java for sorting arrays of primitive types.

4/25/2022

CSE 332

34