

Sorting Introduction

CSE 326
Data Structures
Lecture 3

Plan

- Look at three sorting algorithms in detail
 - › Insertion Sort
 - › Mergesort
 - › Quicksort

Sort Intro - Lecture 3

2

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
 - › reorganize the elements of A such that
 - For any i and j , if $i < j$ then $A[i] \leq A[j]$

Sort Intro - Lecture 3

3

Consistent Ordering

- The comparison function must provide a consistent *ordering* on the set of possible keys
 - › You can compare any two keys and get back an indication of $a < b$, $a > b$, or $a = b$ (tricotomy)
 - › The comparison functions must be consistent
 - If $\text{compare}(a, b)$ says $a < b$, then $\text{compare}(b, a)$ must say $b > a$
 - If $\text{compare}(a, b)$ says $a = b$, then $\text{compare}(b, a)$ must say $b = a$
 - If $\text{compare}(a, b)$ says $a = b$, then $\text{equals}(a, b)$ and $\text{equals}(b, a)$ must say $a = b$

Sort Intro - Lecture 3

4

Why Sort?

- Allows binary search of an N -element array in $O(\log N)$ time
- Allows $O(1)$ time access to k th largest element in the array for any k
- Allows easy detection of any duplicates
- Sorting algorithms are among the most frequently used algorithms in computer science

Sort Intro - Lecture 3

5

Time

- How fast is the algorithm?
 - › The definition of a sorted array A says that 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
 - which is $O(N)$
 - › And you could end up checking each element against every other element
 - which is $O(N^2)$
 - › The big question is: How close to $O(N)$ can you get?

Sort Intro - Lecture 3

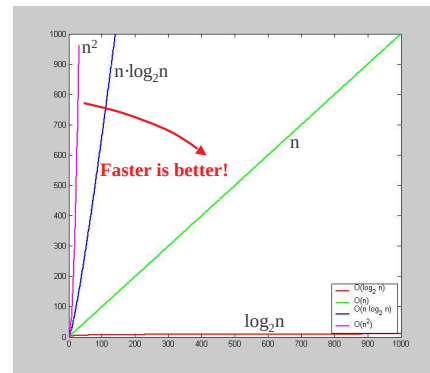
6

Space

- How much space does the sorting algorithm require in order to sort the collection of items?
 - Is copying needed
 - In-place sorting – no copying – $O(1)$ additional space.
 - External memory sorting – data so large that does not fit in memory

Sort Intro - Lecture 3

7



Insertion Sort

- What if first k elements of array are already sorted?
 - 4, 7, 12, 5, 19, 16
- We can shift the tail of the sorted elements list down and then *insert* next element into proper position and we get $k+1$ sorted elements
 - 4, 5, 7, 12, 19, 16

Sort Intro - Lecture 3

9

Insertion Sort Psuedocode

- Sort(array A of length N)
 - For $i = 2 \dots N$
 - While $A[i] > A[i-1]$ and $i >= 0$
 - Swap $A[i]$ and $A[i-1]$
 - $i = i - 1$

Sort Intro - Lecture 3

10

Insertion Sort

```

InsertionSort(A[1..N]: integer array, N: integer)
{
    j, P, temp: integer ;
    for P = 2 to N {
        temp := A[P];
        j := P;
        while j > 1 and A[j-1] > temp do
            A[j] := A[j-1]; j := j-1;
        A[j] := temp;
    }
}

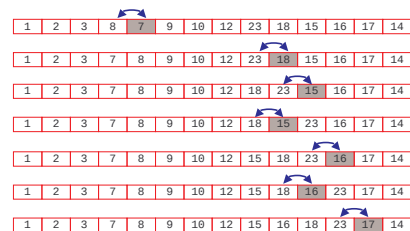
```

- Is Insertion sort in-place?
- Running time = ?

Sort Intro - Lecture 3

11

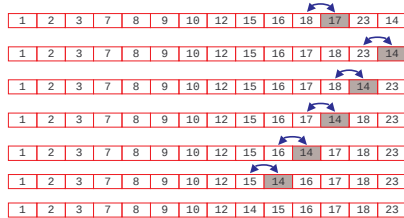
Example



Sort Intro - Lecture 3

12

Example



Sort Intro - Lecture 3

13

Insertion Sort Characteristics

- In-place
- Running time
 - › Worst case is $O(N^2)$
 - reverse order input
 - must copy every element every time
- Good sorting algorithm for almost sorted data
 - › Each item is close to where it belongs in sorted order.

Sort Intro - Lecture 3

14

More Analysis

- Best Case?
 - › $O(N)$
- Average Case? (assume no duplicates)
 - › How many inversions?
 - How many unordered pairs? $N(N+1)/2$
 - Consider any A. Then pairs in A^R are inverted iff not inverted in A. So total number in A and A^R is $N(N+1)/2$. So average is $N(N+1)/4$.
 - › Swap removes exactly one inversion, so $\Omega(N^2)$

Sort Intro - Lecture 3

15

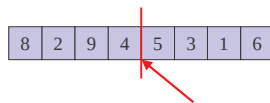
“Divide and Conquer”

- 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 à known as **Mergesort**
- **Idea 2:** Partition array into small items and large items, then recursively sort the two sets à known as **Quicksort**

Sort Intro - Lecture 3

16

Mergesort

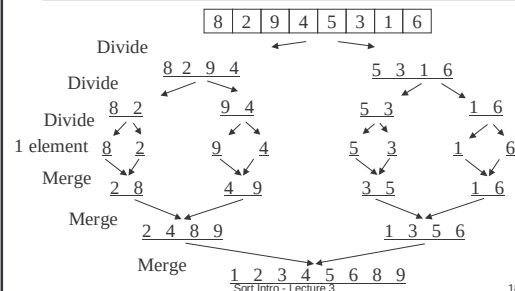


- Divide it in two at the midpoint
- Conquer each side in turn (by recursively sorting)
- Merge two halves together

Sort Intro - Lecture 3

17

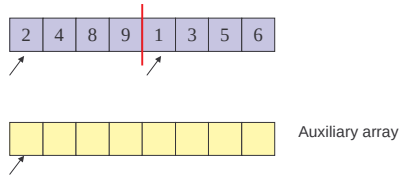
Mergesort Example



18

Auxiliary Array

- The merging requires an auxiliary array.

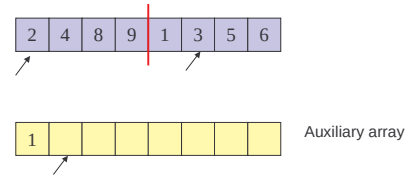


Sort Intro - Lecture 3

19

Auxiliary Array

- The merging requires an auxiliary array.

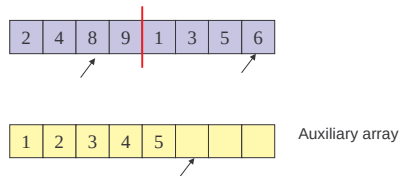


Sort Intro - Lecture 3

20

Auxiliary Array

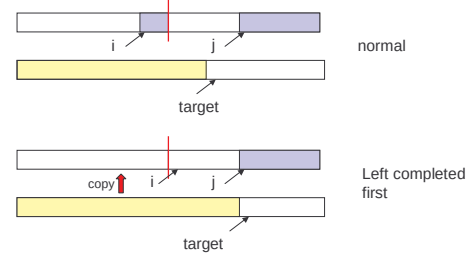
- The merging requires an auxiliary array.



Sort Intro - Lecture 3

21

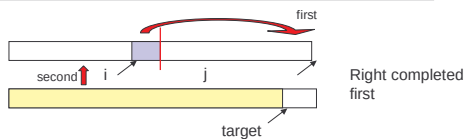
Merging



Sort Intro - Lecture 3

22

Merging



Sort Intro - Lecture 3

23

Merging

```

Merge(A[], T[] : integer array, left, right : integer) : {
    mid, i, j, k, l, target : integer;
    mid := (right + left)/2;
    i := left; j := mid + 1; target := left;
    while i <= mid and j <= right do
        if A[i] <= A[j] then T[target] := A[i]; i := i + 1;
        else T[target] := A[j]; j := j + 1;
        target := target + 1;
    if i > mid then //left completed//
        for k := left to target-1 do A[k] := T[k];
    if j > right then //right completed//
        k := mid; l := right;
        while k >= i do A[l] := A[k]; k := k-1; l := l-1;
        for k := left to target-1 do A[k] := T[k];
}
    
```

Sort Intro - Lecture 3

24

Recursive Mergesort

```

Mergesort(A[], T[] : integer array, left : integer, right : integer) : {
  if left < right then
    mid := (left + right)/2;
    Mergesort(A, T, left, mid);
    Mergesort(A, T, mid+1, right);
    Merge(A, T, left, right);
  }
}

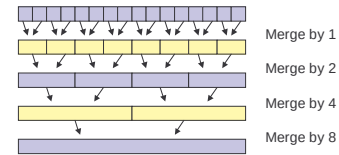
MainMergesort(A[1..n]: integer array, n : integer) : {
  T[1..n]: integer array;
  Mergesort(A, T, 1, n);
}

```

Sort Intro - Lecture 3

25

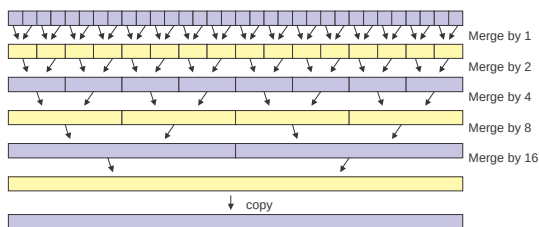
Iterative Mergesort



Sort Intro - Lecture 3

26

Iterative Mergesort



Sort Intro - Lecture 3

27

Iterative pseudocode

- Sort(array A of length N)
 - › Let $m = 2$, let B be temp array of length N
 - › While $m < N$
 - For $i = 1 \dots N$ in increments of m
 - merge $A[i \dots i+m/2]$ and $A[i+m/2 \dots i+m]$ into $B[i \dots i+m]$
 - Swap role of A and B
 - $m = m * 2$
 - › If needed, copy B back to A

Sort Intro - Lecture 3

28

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 $O(N)$

Sort Intro - Lecture 3

29

Mergesort Recurrence Relation

- The recurrence relation for $T(N)$ is:
 - › $T(1) \leq c$
 - base case: 1 element array à constant time
 - › $T(N) \leq 2T(N/2) + dN$
 - 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
- $T(N) = O(N \log N)$

Sort Intro - Lecture 3

30

Solving the Recurrence

$$\begin{aligned}
 T(n) &\leq 2T(n/2) + dn && \text{Assuming } n \text{ is a power of } 2 \\
 &\leq 2(2T(n/4) + dn/2) + dn \\
 &= 4T(n/4) + 2dn \\
 &\leq 4(2T(n/8) + dn/4) + 2dn \\
 &= 8T(n/8) + 3dn \\
 &\vdots \\
 &\leq 2^k T(n/2^k) + kdn \\
 &= nT(1) + kdn && \text{if } n = 2^k \\
 &\leq cn + dn \log_2 n \\
 &= O(n \log n)
 \end{aligned}$$

Sort Intro - Lecture 3

31

Properties of Mergesort

- Not in-place
 - › Requires an auxiliary array
- Very few comparisons
- Iterative Mergesort reduces copying.

Sort Intro - Lecture 3

32

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

Sort Intro - Lecture 3

33

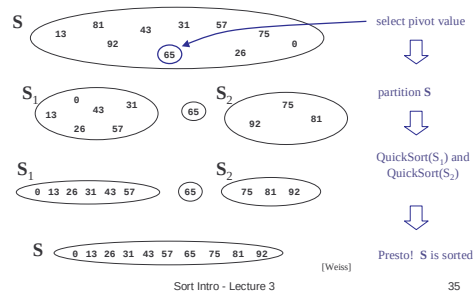
“Four easy steps”

- 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 $x \leq v$ }, and **S**₂ = {all values $x \geq v$ }.
 - › Return QuickSort(**S**₁), v , QuickSort(**S**₂)

Sort Intro - Lecture 3

34

The steps of QuickSort



Sort Intro - Lecture 3

35

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

Sort Intro - Lecture 3

36

Quicksort Partitioning

- Need to partition the array into left and right sub-arrays
 - the elements in left sub-array are $\leq$ pivot
 - elements in right sub-array are $\geq$ 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

Sort Intro - Lecture 3

37

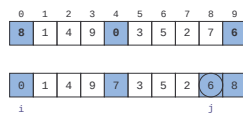
Partitioning is done In-Place

- 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 $A[i] > \text{pivot}$
 - Decrement j until you hit element $A[j] < \text{pivot}$
 - Swap $A[i]$ and $A[j]$
 - Repeat until i and j cross
 - Swap pivot ($= A[N-2]$) with $A[i]$

Sort Intro - Lecture 3

38

Example



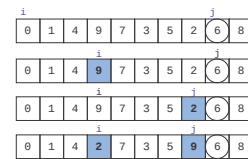
Choose the pivot as the median of three.

Place the pivot and the largest at the right and the smallest at the left

Sort Intro - Lecture 3

39

Example

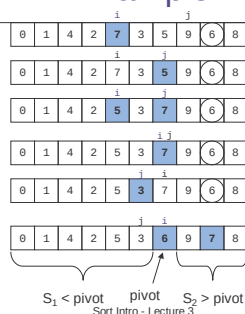


Move i to the right to be larger than pivot.
Move j to the left to be smaller than pivot.
Swap

Sort Intro - Lecture 3

40

Example



$S_1 < \text{pivot}$ pivot $S_2 > \text{pivot}$

Sort Intro - Lecture 3

41

Recursive Quicksort

```

Quicksort(A[]: integer array, left, right : integer): {
    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.

Sort Intro - Lecture 3

42

Alternative Pivot Rules

- Chose $A[\text{left}]$
 - › Fast, but may be too biased
- Chose $A[\text{random}]$, $\text{left} \leq \text{random} \leq \text{right}$
 - › Completely unbiased
 - › Will cause relatively even split, but slow
- Median of three, $A[\text{left}]$, $A[\text{right}]$, $A[(\text{left}+\text{right})/2]$
 - › The standard, tends to be unbiased, and does a little sorting on the side.

Sort Intro - Lecture 3

43

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
 - › $T(N) = O(N \log N)$

Sort Intro - Lecture 3

44

Quicksort Worst Case Performance

- Algorithm always chooses the worst pivot – one sub-array is empty at each recursion
 - › $T(N) \leq a$ for $N \leq C$
 - › $T(N) \leq T(N-1) + bN$
 - › $\leq T(N-2) + b(N-1) + bN$
 - › $\leq T(C) + b(C+1) + \dots + bN$
 - › $\leq a + b(C + C+1 + C+2 + \dots + N)$
 - › $T(N) = O(N^2)$
- Fortunately, *average case performance* is $O(N \log N)$ (see text for proof)

Sort Intro - Lecture 3

45

Properties of Quicksort

- 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.

Sort Intro - Lecture 3

46

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

Sort Intro - Lecture 3

47