CSE 332: Sorting

Richard Anderson, Steve Seitz
Winter 2014

Announcements (2/3/14)

- Reading for this lecture: Chapter 7.
- HW 4 due Wednesday
  - no new HW out this week
- Midterm next Monday

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] ≤ A[j]

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 (trichotomy)
  - The comparison functions must be consistent
    - If `compare(a,b)` says a<b, then `compare(b,a)` must say b>a
    - If `compare(a,b)` says a=b, then `compare(b,a)` must say b=a
    - If `compare(a,b)` says a=b, then `equals(a,b) and equals(b,a)` must say a=b

Why Sort?

- Provides fast search:
- Find *k*th largest element in:

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

Stability

A sorting algorithm is **stable** if:

– Items in the input with the same value end up in the same order as when they began.

Input		**Unstable sort**		**Stable Sort**	
Adams	1	Adams	1	Adams	1
Black	2	Smith	1	Smith	1
Brown	4	Washington	2	Black	2
Jackson	2	Jackson	2	Jackson	2
Jones	4	Black	2	Washington	2
Smith	1	White	3	White	3
Thompson	4	Wilson	3	Wilson	3
Washington	2	Thompson	4	Brown	4
White	3	Brown	4	Jones	4
Wilson	3	Jones	4	Thompson	4

[Sedgewick]

Time

How fast is the algorithm?

– requirement: for any i<j, A[i] ≤ 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*

Simple algorithms: $O(n^2)$	Fancier algorithms: $O(n \log n)$	Comparison lower bound: $\Omega(n \log n)$	Specialized algorithms: $O(n)$	Handling huge data sets
Insertion sort Selection sort …	Heap sort Merge sort Quick sort (avg) …		Bucket sort Radix sort	External sorting

Demo (with sound!)

• http://www.youtube.com/watch?v=kPRA0W1kECg

Selection Sort: idea

1. Find the smallest element, put it 1st
2. Find the next smallest element, put it 2nd
3. Find the next smallest, put it 3rd
4. And so on …

Try it out: Selection Sort

• 31, 16, 54, 4, 2, 17, 6

Selection Sort: Code

```
void SelectionSort (Array a[0..n-1]) {
    for (i=0; i<n; ++i) {
        j = Find index of
                smallest entry in a[i..n-1]
        Swap(a[i],a[j])
    }
}
```

Runtime:

worst case :
best case :
average case :

Bubble Sort

- Take a pass through the array
  - If neighboring elements are out of order, swap them.
- Repeat until no swaps needed

- Wost & avg case: $O(n^2)$
  - pretty much no reason to ever use this algorithm

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…

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?

Try it out: Insertion sort

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

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
        }

}
```

Note: can instead move the "hole" to minimize copying, as with a binary heap.

Runtime:

worst case :
best case :
average case :

Insertion Sort vs. Selection Sort

- Same worst case, avg case complexity
- Insertion better best-case
  - preferable when input is "almost sorted"
    - one of the best sorting algs for almost sorted case (also for small arrays)

19

Sorting: *The Big Picture*

Simple algorithms: $O(n^2)$	Fancier algorithms: $O(n \log n)$	Comparison lower bound: $\Omega(n \log n)$	Specialized algorithms: $O(n)$	Handling huge data sets
Insertion sort Selection sort …	Heap sort Merge sort Quick sort (avg) …		Bucket sort Radix sort	External sorting

20

Heap Sort: Sort with a Binary Heap

Worst Case Runtime:

21

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	7	5	9	8	6	10	3	2	1

$\underbrace{4\ 7\ 5\ 9\ 8\ 6\ 10}_{\text{heap part}}\ \underbrace{3\ 2\ 1}_{\text{sorted part}}$

```
arr[n-i]=
deleteMin()
```

5	7	6	9	8	10	4	3	2	1

$\underbrace{5\ 7\ 6\ 9\ 8\ 10}_{\text{heap part}}\ \underbrace{4\ 3\ 2\ 1}_{\text{sorted part}}$

10/21/2013 22

AVL Sort

Worst Case Runtime:

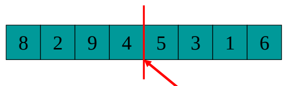
23

"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

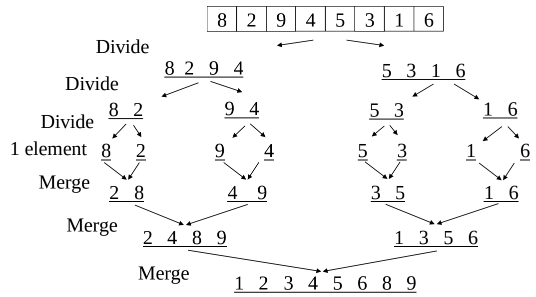
24

Mergesort



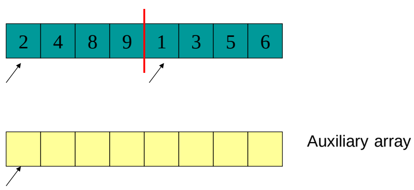
- Divide it in two at the midpoint
- Sort each half (recursively)
- Merge two halves together

Mergesort Example



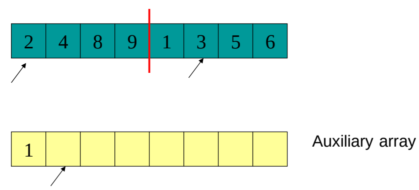
Merging: Two Pointer Method

- Perform merge using an auxiliary array



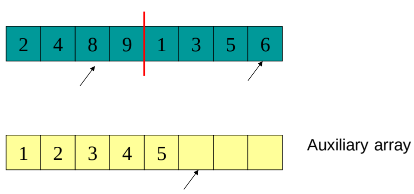
Merging: Two Pointer Method

- Perform merge using an auxiliary array

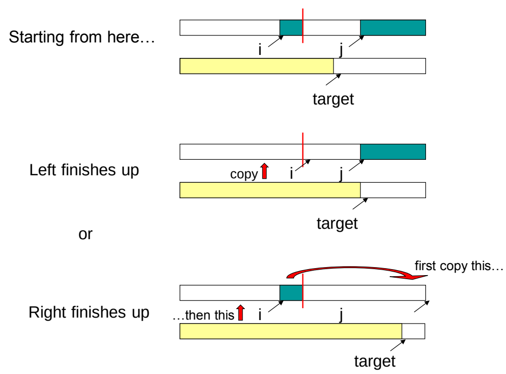


Merging: Two Pointer Method

- Perform merge using an auxiliary array

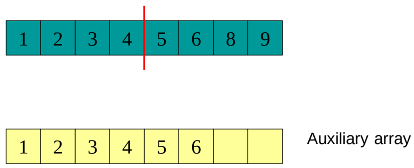


Merging: Finishing Up



Merging: Two Pointer Method

- Final result



Complexity? Stability?

31

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 ≥ i)
      A[l--] = A[k--]
    for (k = left to target-1)
      A[k] = Temp[k]
}
```

32

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)
  }
}
```

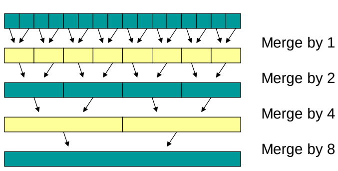
What is the recurrence relation?

33

Mergesort: Complexity

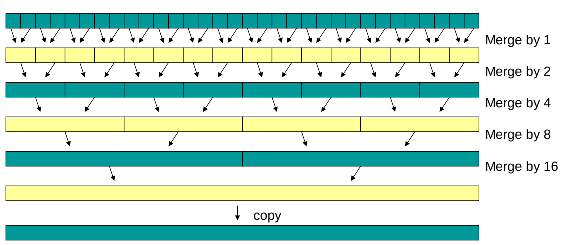
34

Iterative Mergesort



35

Iterative Mergesort



Iterative Mergesort reduces copying
Complexity?

36

Properties of Mergesort

- In-place?
- Stable?
- 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.

37

Quicksort

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

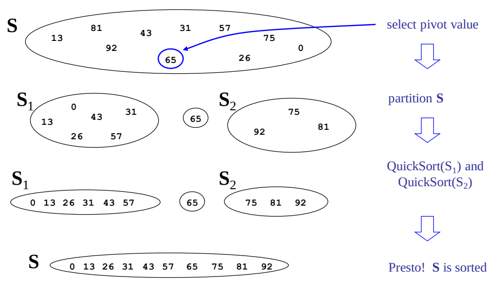
Here's the idea for sorting array **S**:

1. Pick an element *v* in **S**. This is the ***pivot*** value.
2. Partition **S**-{*v*} into two disjoint subsets, $\mathbf{S}_1$ and $\mathbf{S}_2$ such that:
   - elements in $\mathbf{S}_1$ are all $\leq v$
   - elements in $\mathbf{S}_2$ are all $\geq v$
3. Return concatenation of QuickSort($\mathbf{S}_1$), *v*, QuickSort($\mathbf{S}_2$)

Recursion ends when Quicksort( ) receives an array of length 0 or 1.

38

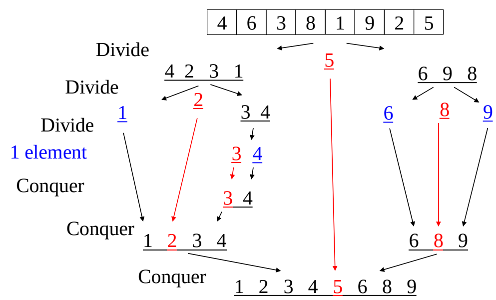
The steps of Quicksort



[Weiss]

39

Quicksort Example



40

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

41

Picking the Pivot

42

Median of Three Pivot

0	1	2	3	4	5	6	7	8	9
8	1	4	9	**6**	3	5	2	7	**0**

medianOf3Pivot(…)

0	1	4	9	7	3	5	2	6	8

Choose the pivot as the median of three.

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

43

Quicksort Partitioning

- Partition the array into left and right sub-arrays such that:
  - elements in left sub-array are ≤ pivot
  - elements in right sub-array are ≥ 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.

44

Partioning In-place

Setup: `i` = start and `j` = end of un-partioned elements:

	i						j		
0	1	4	9	7	3	5	2	6	8

Advance `i` until element ≥ pivot:

			i				j		
0	1	4	**9**	7	3	5	2	6	8

Advance `j` until element ≤ pivot:

			i				j		
0	1	4	9	7	3	5	**2**	6	8

If `j` > `i`, then swap:

			i				j		
0	1	4	**2**	7	3	5	**9**	6	8

45

Partioning In-place

Advance `i` :

				i			j		
0	1	4	2	**7**	3	5	9	6	8

Advance `j` :

				i		j			
0	1	4	2	7	3	**5**	9	6	8

Swap :

				i		j			
0	1	4	2	**5**	3	**7**	9	6	8

Advance `i` :

						i j			
0	1	4	2	5	3	**7**	9	6	8

Advance `j` :

					j	i			
0	1	4	2	5	**3**	7	9	6	8

`i` > `j`, swap in pivot, partition done!

					j	i			
0	1	4	2	5	3	**6**	9	**7**	8

$S_1 \leq$ pivot　　pivot　　$S_2 \geq$ pivot

46

Partition Pseudocode

```
Partition(A[], left, right) {
  v = A[right]; // Assumes pivot value currently at right
  i = left;     // Initialize left side, right side pointers
  j = right-1;

  // Do i++, j-- until they cross, swapping values as needed
  while (1) {
    while (A[i] < v) i++;
    while (A[j] > v) j--;
    if (i < j) {
      Swap(A[i], A[j]);
      i++; j--;
    }
    else
      break;
  }
  Swap(A[i], A[right]); // Swap pivot value into position
  return i;             // Return the final pivot position
}
```

Complexity for input size *n*?

47

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);
  }
}
```

48

QuickSort:
Best case complexity

```
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);
  }
}
```


QuickSort:
Worst case complexity

```
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);
  }
}
```


QuickSort:
Average case complexity

Turns out to be **O(*n* log *n*)**.

See Section 7.7.5 for an idea of the proof.
Don't need to know proof details for this course.

Many Duplicates?

An important case to consider is when an array has many duplicates.

0	1	2	3	4	5	6	7	8	9
8	6	6	6	**6**	6	6	6	6	**0**

$\downarrow$ `medianOf3Pivot(…)`

0	6	6	6	6	6	6	6	6	8

Partitioning with Duplicates

Setup: `i` = start and `j` = end of un-partioned elements:

	i						j		
0	6	6	6	6	6	6	6	6	8

Advance `i` until element ≥ pivot:

	i						j		
0	**6**	6	6	6	6	6	6	6	8

Advance `j` until element ≤ pivot:

	i						j		
0	6	6	6	6	6	6	**6**	6	8

If `j` > `i`, then swap:

	i						j		
0	**6**	6	6	6	6	6	**6**	6	8

Partitioning with Duplicates

Advance `i,j`:

		i				j			
0	6	**6**	6	6	6	**6**	6	6	8

Swap:

		i				j			
0	6	**6**	6	6	6	**6**	6	6	8

Advance `i,j`:

			i		j				
0	6	6	**6**	6	**6**	6	6	6	8

Swap:

			i		j				
0	6	6	**6**	6	**6**	6	6	6	8

Advance `i,j`:

				i j					
0	6	6	6	**6**	6	6	6	6	8

Finish:

				i j					
0	6	6	6	**6**	6	6	6	**6**	8

Partitioning with Duplicates:Take Two

Start `i` = start and `j` = end of un-partioned elements:

	i						j		
0	6	6	6	6	6	6	6	6	8

Advance `i` until element **>** pivot (and in bounds):

							i j		
0	6	6	6	6	6	6	**6**	6	8

Advance `j` until element **<** pivot (and in bounds):

	j						i		
0	**6**	6	6	6	6	6	**6**	6	8

Finish:

	j						i		
0	6	6	6	6	6	6	**6**	**6**	8

Is this better?

Partitioning with Duplicates: Upshot

It's better to stop advancing pointers when elements are equal to pivot, and then just do swaps.

Complexity of quicksort on an array of identical values?

Can we do better?

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 = 10 is reasonable.

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.
- Stable?
- Used by Java for sorting arrays of primitive types.