File searchSortAlgorithms.zip on course website (lecture notes for lectures 12, 13) contains ALL searching/ sorting algorithms. Download it and look at algorithms

SORTING AND ASYMPTOTIC COMPLEXITY

Lecture 12
CS2110 – Spring 2014

Execution of logarithmic-space Quicksort

2

```
/** Sort b[h..k]. */
public static void QS(int[] b, int h, int k) {
   int h1= h; int k1= k;
   // inv; b[h..k] is sorted if b[h1..k1] is
   while (size of b[h1..k1] > 1) {
       int j= partition(b, h1, k1);
       // b[h1..j-1] <= b[j] <= b[j+1..k1]
       if (b[h1..j-1] smaller than b[j+1..k1])
           { QS(b, h, j-1); h1= j+1; }
       else
           {QS(b, j+1, k1); k1= j-1; }
   }
}
```

Last lecture ended with presenting this algorithm. There was no time to explain it. We now show how it is executed in order to illustrate how the invariant is maintained

Call QS(b, 0, 11);

3

```
public static void QS(int[] b, int h, int k) {
  int h1= h; int k1= k;
  // inv; b[h..k] is sorted if b[h1..k1] is
  while (size of b[h1..k1] > 1) {
    int j= partition(b, h1, k1);
    // b[h1..j-1] <= b[j] <= b[j+1..k1]
    if (b[h1..j-1] smaller than b[j+1..k1])
       { QS(b, h, j-1); h1= j+1; }
    else {QS(b, j+1, k1); k1= j-1; }
  }
}
```

Initially, h is 0 and k is 11. The initialization stores 0 and 11 in h1 and k1. The invariant is true since h = h1 and k = k1.

j ?
h 0   k 11
h1 0   k1 11

0 11
3 4 8 7 6 8 9 1 2 5 7 9

Call QS(b, 0, 11);

4

```
public static void QS(int[] b, int h, int k) {
  int h1= h; int k1= k;
  // inv; b[h..k] is sorted if b[h1..k1] is
  while (size of b[h1..k1] > 1) {
    int j= partition(b, h1, k1);
    // b[h1..j-1] <= b[j] <= b[j+1..k1]
    if (b[h1..j-1] smaller than b[j+1..k1])
       { QS(b, h, j-1); h1= j+1; }
    else {QS(b, j+1, k1); k1= j-1; }
  }
}
```

The assignment to j partitions b, making it look like what is below. The two partitions are underlined

j 2
h 0   k 11
h1 0   k1 11

0 j 11
2 1 3 7 6 8 9 4 8 5 7 9

Call QS(b, 0, 11);

5

```
public static void QS(int[] b, int h, int k) {
  int h1= h; int k1= k;
  // inv; b[h..k] is sorted if b[h1..k1] is
  while (size of b[h1..k1] > 1) {
    int j= partition(b, h1, k1);
    // b[h1..j-1] <= b[j] <= b[j+1..k1]
    if (b[h1..j-1] smaller than b[j+1..k1])
       { QS(b, h, j-1); h1= j+1; }
    else {QS(b, j+1, k1); k1= j-1; }
  }
}
```

The left partition is smaller, so it is sorted recursively by this call. We have changed the partition to the result.

h 0   k 11
h1 0   k1 11

0 j 11
1 2 3 7 6 8 9 4 8 5 7 9

Call QS(b, 0, 11);

6

```
public static void QS(int[] b, int h, int k) {
  int h1= h; int k1= k;
  // inv; b[h..k] is sorted if b[h1..k1] is
  while (size of b[h1..k1] > 1) {
    int j= partition(b, h1, k1);
    // b[h1..j-1] <= b[j] <= b[j+1..k1]
    if (b[h1..j-1] smaller than b[j+1..k1])
       { QS(b, h, j-1); h1= j+1; }
    else {QS(b, j+1, k1); k1= j-1; }
  }
}
```

The assignment to h1 is done.

Do you see that the inv is true again? If the underlined partition is sorted, then so is b[h..k]. Each iteration of the loop keeps inv true and reduces size of b[h1..k1].

j 2
h 0   k 11
h1 3   k1 11

0 j 11
1 2 3 7 6 8 9 4 8 5 7 9

Divide & Conquer!

It often pays to

- Break the problem into smaller subproblems,
- Solve the subproblems separately, and then
- Assemble a final solution

This technique is called *divide-and-conquer*

- Caveat: It won't help unless the *partitioning* and *assembly* processes are inexpensive

We did this in Quicksort: Partition the array and then sort the two partitions.

MergeSort

Quintessential divide-and-conquer algorithm:

Divide array into equal parts, sort each part (recursively), then merge

Questions:

- Q1: How do we divide array into two equal parts?
  A1: Find middle index: b.length/2
- Q2: How do we sort the parts?
  A2: Call MergeSort recursively!
- Q3: How do we merge the sorted subarrays?
  A3: It takes linear time.

Merging Sorted Arrays A and B into C

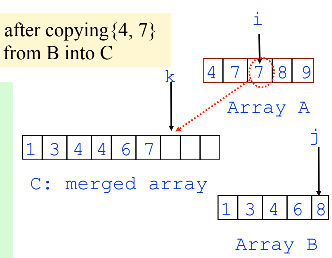
Picture shows situation after copying {4, 7} from A and {1, 3, 4, 6} from B into C

A[0..i-1] and B[0..j-1] have been copied into C[0..k-1].

C[0..k-1] is sorted.

Next, put a[i] in c[k], because a[i] < b[j].

Then increase k and i.



Merging Sorted Arrays A and B into C

- Create array C of size: size of A + size of B
- i= 0; j= 0; k= 0; // initially, nothing copied
- Copy smaller of A[i] and B[j] into C[k]
- Increment i or j, whichever one was used, and k
- When either A or B becomes empty, copy remaining elements from the other array (B or A, respectively) into C

This tells what has been done so far:

A[0..i-1] and B[0..j-1] have been placed in C[0..k-1].

C[0..k-1] is sorted.

MergeSort


```
/** Sort b[h..k] */
public static void MS
      (int[] b, int h, int k) {
   if (k – h <= 1) return;
   MS(b, h, (h+k)/2);
   MS(b, (h+k)/2 + 1, k);
   merge(b, h, (h+k)/2, k);
}
```

merge 2 sorted arrays

QuickSort versus MergeSort


```
/** Sort b[h..k] */
public static void QS
      (int[] b, int h, int k) {
   if (k – h <= 1) return;
   int j= partition(b, h, k);
   QS(b, h, j-1);
   QS(b, j+1, k);
}
```

```
/** Sort b[h..k] */
public static void MS
      (int[] b, int h, int k) {
   if (k – h <= 1) return;
   MS(b, h, (h+k)/2);
   MS(b, (h+k)/2 + 1, k);
   merge(b, h, (h+k)/2, k);
}
```

One processes the array then recurses.
One recurses then processes the array.

merge 2 sorted arrays

MergeSort Analysis

13

Outline

- Split array into two halves
- Recursively sort each half
- Merge two halves

Merge: combine two sorted arrays into one sorted array:

- Time: O(n) where n is the total size of the two arrays

Runtime recurrence

T(n): time to sort array of size n

$T(1) = 1$

$T(n) = 2T(n/2) + O(n)$

Can show by induction that

T(n) is O(n log n)

Alternatively, can see that T(n) is O(n log n) by looking at tree of recursive calls

MergeSort Notes

14

- Asymptotic complexity: O(n log n)
  - Much faster than $O(n^2)$
- Disadvantage
  - Need extra storage for temporary arrays
  - In practice, can be a disadvantage, even though MergeSort is *asymptotically optimal for sorting*
  - Can do MergeSort in place, but *very* tricky (and slows execution significantly)
- Good sorting algorithm that does not use so much extra storage? Yes: QuickSort —when done properly, uses log n space.

QuickSort Analysis

15

Runtime analysis (worst-case)

- Partition can produce this: | p | ≥ p |
- Runtime recurrence: $T(n) = T(n-1) + n$
- Can be solved to show worst-case T(n) is $O(n^2)$
- Space can be O(n) —max depth of recursion

Runtime analysis (expected-case)

- More complex recurrence
- Can be solved to show *expected* T(n) is O(n log n)

Improve constant factor by avoiding QuickSort on *small* sets

- Use InsertionSort (for example) for sets of size, say, ≤ 9
- Definition of *small* depends on language, machine, etc.

Sorting Algorithm Summary

16

We discussed

- InsertionSort
- SelectionSort
- MergeSort
- QuickSort

Other sorting algorithms

- HeapSort (will revisit)
- ShellSort (in text)
- BubbleSort (nice name)
- RadixSort
- BinSort
- CountingSort

Why so many? Do computer scientists have some kind of sorting fetish or what?

Stable sorts: **Ins, Sel, Mer**

Worst-case O(n log n): **Mer, Hea**

Expected O(n log n): **Mer, Hea, Qui**

Best for nearly-sorted sets: **Ins**

No extra space: **Ins, Sel, Hea**

Fastest in practice: **Qui**

Least data movement: **Sel**

A sorting algorithm is stable if: equal values stay in same order: b[i] = b[j] and i < j means that b[i] will precede b[j] in result

Lower Bound for Comparison Sorting

17

Goal: Determine minimum time *required* to sort *n* items

Note: we want *worst-case*, not *best-case* time

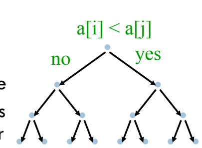
- Best-case doesn't tell us much. E.g. Insertion Sort takes O(n) time on already-sorted input
- Want to know *worst-case time* for *best possible* algorithm

- How can we prove anything about the *best possible* algorithm?
- Want to find characteristics that are common to *all* sorting algorithms
- Limit attention to *comparison-based algorithms* and try to count number of comparisons

Comparison Trees

18

- Comparison-based algorithms make decisions based on comparison of data elements
- Gives a *comparison tree*
- If algorithm fails to terminate for some input, comparison tree is infinite
- Height of comparison tree represents *worst-case number of comparisons* for that algorithm
- Can show: *Any* correct comparison-based algorithm must make at least n log n comparisons in the worst case



Lower Bound for Comparison Sorting

19

- Say we have a correct comparison-based algorithm
- Suppose we want to sort the elements in an array b[]
- Assume the elements of b[] are distinct
- Any permutation of the elements is initially possible
- When done, b[] is sorted
- But the algorithm could not have taken the same path in the comparison tree on different input permutations

Lower Bound for Comparison Sorting

20

How many input permutations are possible? $n! \sim 2^{n \log n}$

For a comparison-based sorting algorithm to be correct, it must have at least that many leaves in its comparison tree

To have at least $n! \sim 2^{n \log n}$ leaves, it must have height at least n log n (since it is only binary branching, the number of nodes at most doubles at every depth)

Therefore its longest path must be of length at least n log n, and that it its worst-case running time

Interface java.lang.Comparable<T>

21

```
public int compareTo(T x);
```

- Return a negative, zero, or positive value
  - negative if **this** is before **x**
  - 0 if **this.equals(x)**
  - positive if **this** is after **x**

Many classes implement `Comparable`

- `String, Double, Integer, Character, Date, …`
- Class implements Comparable? Its method compareTo is considered to define that class's *natural ordering*

Comparison-based sorting methods should work with `Comparable` for maximum generality