"Organizing is what you do before you do something, so that when you do it, it is not all mixed up."
~ A. A. Milne

SORTING

Lecture 11
CS2110 – Fall 2017

Announcements

2

SPLASH! at Cornell is a program with a *"teach anything, learn anything"* philosophy. You will be able to provide high schoolers with instruction in the topic of your choice.

This semester's event is on Saturday, November 4

Apply to be a teacher!

If you are interested, please email us at: splashcornell@gmail.com.

Announcements

3

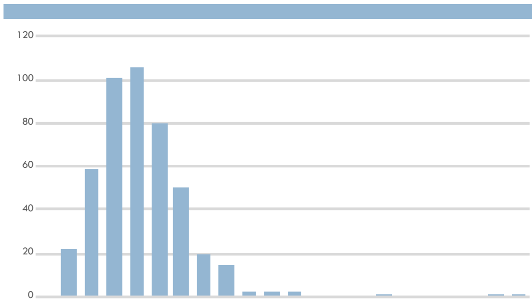
Prelim 1

4

- It's on Thursday Evening (9/28)
- Two Sessions:
  - 5:30-7:00PM: A..Lid
  - 7:30-9:00PM: Lie..Z
- Three Rooms:
  - We will email you Thursday morning with your room
- Bring your Cornell ID!!!

A3

5



A3 Comments

6

A3 Comments

7

/* Mini lecture on linked lists would have been very helpful. I still do not
* know when we covered this topic in class. It was initially difficult to
* understand what we were meant to do without having learned the topic
* in depth before */

/* Maybe the assignment guide could explain a bit more about how to
* thoroughly test the methods though. Testing is still a bit difficult and I
* wish we had an assignment which covered that more. The instructions
* could have been more specific about what is expected from the test
* cases though. */

/* It also showed me how important it is to test after writing a method. I
* had messed up on one of the earlier methods and if I had waited to test
* I would have had a lot of trouble figuring out what went wrong. This
* assignment showed me how vital it is to test not at the end but
* incrementally. I feel more careful, efficient, and organized. */

Why Sorting?

8

- Sorting is useful
  - Database indexing
  - Operations research
  - Compression
- There are lots of ways to sort
  - There isn't one right answer
  - You need to be able to figure out the options and decide which one is right for your application.
  - Today, we'll learn about several different algorithms (and how to derive them)

Some Sorting Algorithms

9

- Insertion sort
- Selection sort
- Merge sort
- Quick sort

InsertionSort

10

pre: b [0 .. b.length: ?] post: b [0 .. b.length: sorted]

inv: b [0 .. i: sorted | i .. b.length: ?]

or: b[0..i-1] is sorted

A loop that processes elements of an array in increasing order has this invariant

Each iteration, i= i+1; How to keep inv true?

11

inv: b [0 .. i: sorted | i .. b.length: ?]

e.g. b [0: 2 5 5 5 7 | i: 3 ? | b.length]

b [0: 2 3 5 5 5 7 | i: ? | b.length]

What to do in each iteration?

12

inv: b [0 .. i: sorted | i .. b.length: ?]

e.g. b [0: 2 5 5 5 7 | i: 3 ? | b.length]

Loop body (inv true before and after):

2 5 5 5 3 | 7 ?
2 5 5 3 5 | 7 ?
2 5 3 5 5 | 7 ?
2 3 5 5 5 | 7 ?

Push b[i] to its sorted position in b[0..i], then increase i

This will take time proportional to the number of swaps needed

b [2 3 5 5 5 7 | ?]

Insertion Sort

13

```
// sort b[], an array of int
// inv: b[0..i-1] is sorted
for (int i= 0; i < b.length; i= i+1) {
    // Push b[i] down to its sorted
    // position in b[0..i]

        Present algorithm like this

}
```

Note English statement in body.

Abstraction. Says **what** to do, not **how.**

This is the best way to present it. We expect you to present it this way when asked.

Later, can show how to implement that with an inner loop

Insertion Sort

14

```
// sort b[], an array of int
// inv: b[0..i-1] is sorted
for (int i= 0; i < b.length; i= i+1) {
    // Push b[i] down to its sorted
    // position in b[0..i]
    int k= i;
    while (k > 0  &&  b[k] < b[k-1]) {
        Swap b[k] and b[k-1]
        k= k–1;
    }
}
```

invariant P: b[0..i] is sorted
except that b[k] may be < b[k-1]

k				i	
2 5	3	5	5	7	?

example

start?

stop?

progress?

maintain invariant?

Insertion Sort

15

```
// sort b[], an array of int
// inv: b[0..i-1] is sorted
for (int i= 0; i < b.length; i= i+1) {
    Push b[i] down to its sorted position
    in b[0..i]
}
```

Pushing b[i] down can take i swaps.
Worst case takes

1 + 2 + 3 + … n-1 = (n-1)*n/2

Swaps.

Let n = b.length

- Worst-case: $O(n^2)$ (reverse-sorted input)
- Best-case: $O(n)$ (sorted input)
- Expected case: $O(n^2)$

Performance

16

Algorithm	Time	Space	Stable?
Insertion Sort	$O(n)$ to $O(n^2)$	$O(1)$	Yes
Selection Sort	$O(n^2)$	$O(1)$	No
Merge Sort			
Quick Sort			

SelectionSort

17

pre: b [0 ? b.length]   post: b [0 sorted b.length]

inv: b [0 sorted , <= b[i..] | i >= b[0..i-1] b.length]   Additional term in invariant

Keep invariant true while making progress?

e.g.: b [0 1 2 3 4 5 6 | i 9 9 9 7 8 6 9 b.length]

Increasing i by 1 keeps inv true only if b[i] is min of b[i..]

SelectionSort

```
//sort b[], an array of int
// inv: b[0..i-1] sorted  AND
//      b[0..i-1]  <=  b[i..]
for (int i= 0; i < b.length; i= i+1) {
   int m= index of minimum of b[i..];
   Swap b[i] and b[m];
}
```

Another common way for people to sort cards

Runtime with n = b.length

- Worst-case $O(n^2)$
- Best-case $O(n^2)$
- Expected-case $O(n^2)$

b [0 sorted, smaller values | i larger values length]

Each iteration, swap min value of this section into b[i]

Performance

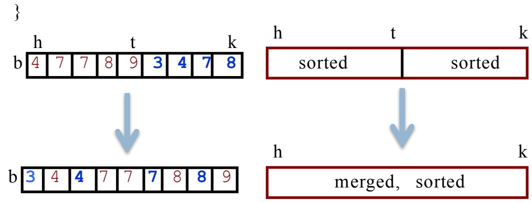
19

Algorithm	Time	Space	Stable?
Insertion Sort	$O(n)$ to $O(n^2)$	$O(1)$	Yes
Selection Sort	$O(n^2)$	$O(1)$	No
Merge Sort			
Quick Sort			

Merge two adjacent sorted segments

20

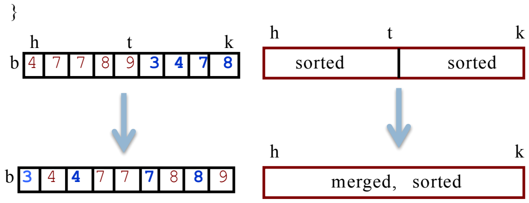
```
/* Sort b[h..k]. Precondition: b[h..t] and b[t+1..k] are sorted.  */
public static merge(int[] b, int h, int t, int k) {

}
```



Merge two adjacent sorted segments

21

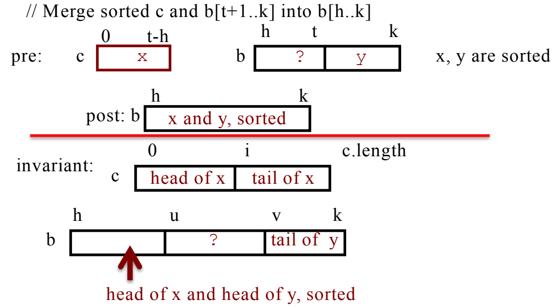
```
/* Sort b[h..k]. Precondition: b[h..t] and b[t+1..k] are sorted.  */
public static merge(int[] b, int h, int t, int k) {
    Copy b[h..t] into a new array c;
    Merge c and b[t+1..k] into b[h..k];
}
```



Merge two adjacent sorted segments

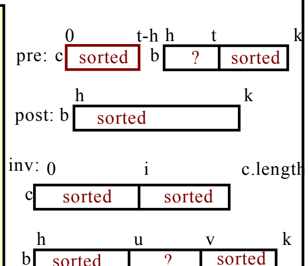
22

// Merge sorted c and b[t+1..k] into b[h..k]



Merge

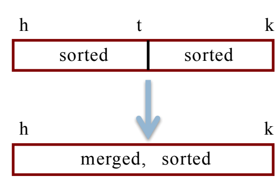
```
int i = 0;
int u = h;
int v = t+1;
while( i < t-h){
    if(v < k && b[v] < c[i]) {
        b[u] = b[v];
        u++; v++;
    }else {
        b[u] = c[i];
        u++; i++;
    }
}
```



Mergesort

24

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



Performance

25

Algorithm	Time	Space	Stable?
Insertion Sort	$O(n)$ to $O(n^2)$	$O(1)$	Yes
Selection Sort	$O(n^2)$	$O(1)$	No
Merge Sort	$n\log(n)$	$O(n)$	Yes
Quick Sort			

QuickSort

26

Quicksort developed by Sir Tony Hoare (he was knighted by the Queen of England for his contributions to education and CS).

83 years old.

Developed Quicksort in 1958. But he could not explain it to his colleague, so he gave up on it.

Later, he saw a draft of the new language Algol 58 (which became Algol 60). It had recursive procedures. First time in a procedural programming language. "Ah!," he said. "I know how to write it better now." 15 minutes later, his colleague also understood it.

Partition algorithm of quicksort

27

```
        h  h+1                          k
pre:   | x |            ?              |        x is called
                                                 the pivot
```

Swap array values around until b[h..k] looks like this:

```
        h                 j             k
post:  |      <= x      | x |   >= x   |
```

28

```
| 20 | 31 | 24 | 19 | 45 | 56 | 4 | 20 | 5 | 72 | 14 | 99 |
```

pivot

partition

```
                        j
| 19 | 4 | 5 | 14 | 20 | 31 | 24 | 45 | 56 | 20 | 72 | 99 |
```

Not yet sorted

these can be in any order

these can be in any order

Not yet sorted

The 20 could be in the other partition

Partition algorithm

29

```
         h  h+1                          k
pre:   b | x |            ?              |

         h                 j             k
post:  b |      <= x      | x |   >= x   |
```

Combine pre and post to get an invariant

invariant needs at least 4 sections

```
         h             j          t          k
       b |   <= x    | x |   ?   |   >= x   |
```

Partition algorithm

30

```
   h             j          t          k
 b |   <= x    | x |   ?   |   >= x   |
```

Initially, with j = h and t = k, this diagram looks like the start diagram

```
j= h; t= k;
while (j < t) {
   if (b[j+1] <= b[j]) {
      Swap b[j+1] and b[j];   j= j+1;
   } else {
      Swap b[j+1] and b[t];   t= t-1;
   }
}
```

Terminate when j = t, so the "?" segment is empty, so diagram looks like result diagram

Takes linear time: O(k+1-h)

QuickSort procedure

```
/** Sort b[h..k]. */
public static void QS(int[] b, int h, int k) {
  if (b[h..k] has < 2 elements) return;   Base case

  int j= partition(b, h, k);
    // We know b[h..j–1] <= b[j] <= b[j+1..k]

    // Sort b[h..j-1] and b[j+1..k]
    QS(b, h, j-1);
    QS(b, j+1, k);
}
```

Function does the partition algorithm and returns position j of pivot

h	j	k
<= x	x	>= x

Worst case quicksort: pivot always smallest value

j n
| x0 | >= x0 | — partioning at depth 0

j
| x0 | x1 | >= x1 | — partioning at depth 1

j
| x0 | x1 | x2 | >= x2 | — partioning at depth 2

```
/** Sort b[h..k]. */
public static void QS(int[] b, int h, int k) {
  if (b[h..k] has < 2 elements) return;
  int j= partition(b, h, k);
  QS(b, h, j-1);    QS(b, j+1, k);
```

Depth of recursion: O(n)

Processing at depth i: O(n-i)

O(n*n)

Best case quicksort: pivot always middle value

0 j n
| <= x0 | x0 | >= x0 | — depth 0. 1 segment of size ~n to partition.

| <=x1 | x1 | >= x1 | x0 | <=x2 | x2 | >=x2 | — Depth 2. 2 segments of size ~n/2 to partition.

Depth 3. 4 segments of size ~n/4 to partition.

Max depth: O(log n). Time to partition on each level: O(n)
Total time: O(n log n).

Average time for Quicksort: n log n. Difficult calculation

QuickSort complexity to sort array of length n

Time complexity
Worst-case: O(n*n)
Average-case: O(n log n)

```
/** Sort b[h..k]. */
public static void QS(int[] b, int h, int k) {
  if (b[h..k] has < 2 elements) return;
  int j= partition(b, h, k);
  // We know b[h..j–1] <= b[j] <= b[j+1..k]
  // Sort b[h..j-1] and b[j+1..k]
 QS(b, h, j-1);
 QS(b, j+1, k);
}
```

Worst-case space: ?
What's depth of recursion?

Worst-case space: O(n)!
--depth of recursion can be n
Can rewrite it to have space O(log n)
Show this at end of lecture if we have time

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.

Partition. Key issue. How to choose pivot

pre: b — h h k: | x | ? |

post: b — h j k: | <= x | x | >= x |

Choosing pivot
Ideal pivot: the median, since it splits array in half
But computing is O(n), quite complicated

Popular heuristics: Use
- first array value (not so good)
- middle array value (not so good)
- Choose a random element (not so good)
- median of first, middle, last, values (often used)!

Performance

37

Algorithm	Time	Space	Stable?
Insertion Sort	$O(n)$ to $O(n^2)$	$O(1)$	Yes
Selection Sort	$O(n^2)$	$O(1)$	No
Merge Sort	$n\log(n)$	$O(n)$	Yes
Quick Sort	$n\log(n)$ to $O(n^2)$	$O(\log(n))$	No

Sorting in Java

38

- Java.util.Arrays has a method Sort()
  - implemented as a collection of overloaded methods
  - for primitives, Sort is implemented with a version of quicksort
  - for Objects that implement Comparable, Sort is implemented with mergesort
- Tradeoff between speed/space and stability/performance guarantees

Quicksort with logarithmic space

39

Problem is that if the pivot value is always the smallest (or always the largest), the depth of recursion is the size of the array to sort.

Eliminate this problem by doing some of it iteratively and some recursively. We may show you this later. Not today!

QuickSort with logarithmic space

40

```
/** Sort b[h..k]. */
public static void QS(int[] b, int h, int k) {
   int h1= h; int k1= k;
   // invariant b[h..k] is sorted if b[h1..k1] is sorted
   while (b[h1..k1] has more than 1 element) {
        Reduce the size of b[h1..k1], keeping inv true
   }
}
```

QuickSort with logarithmic space

41

```
/** Sort b[h..k]. */
public static void QS(int[] b, int h, int k) {
   int h1= h; int k1= k;
   // invariant b[h..k] is sorted if b[h1..k1] is sorted
   while (b[h1..k1] has more than 1 element) {
        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; }
   }
}
```

Only the smaller segment is sorted recursively. If b[h1..k1] has size n, the smaller segment has size < n/2. Therefore, depth of recursion is at most log n