We may not cover
all this material

SEARCHING AND SORTING
HINT AT ASYMPTOTIC COMPLEXITY

Lecture 9
CS2110 – Fall 2015

Miscellaneous

- Prelim a week from now. Thursday night. By tonight, all people with conflicts should either have emailed Megan or completed assignment P1Conflict. (36 did so, till now.)
  Review session Sunday 1-3PM, Kimball B11. Next week's recitation also a review.
- A3 due Monday night. Group early! Only 328 views of the piazza A3 FAQ.
- Piazza Supplemental study material. We will be putting something on it soon about loop invariants –up to last lecture.
- Sorry for the mistakes in uploading todays' lecture to the CMS. My mistake. Usually I check when I upload something. This time, in a hurry, I didn't.

Last lecture: binary search

pre: b [0 … b.length: ?]

post: b [0 … h: <= v | > v … b.length]

inv: b [0 … h: <= v | ? | t … b.length: > v]

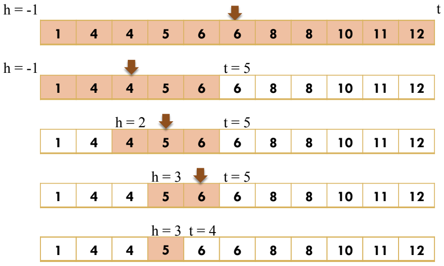
```
h= –1;  t= b.length;
while  (h != t–1) {
    int  e=  (h+t)/2;
    if (b[e] <= v)  h=  e;
    else  t=  e;
}
```

Methodology:
1. Draw the invariant as a combination of pre and post
2. Develop loop using 4 loopy questions.

Practice doing this!

Binary search: find position *h* of *v* = 5

pre: array is sorted



h = -1 t = 11
1 4 4 5 6 6 8 8 10 11 12

h = -1 t = 5
1 4 4 5 6 6 8 8 10 11 12

h = 2 t = 5
1 4 4 5 6 6 8 8 10 11 12

h = 3 t = 5
1 4 4 5 6 6 8 8 10 11 12

h = 3 t = 4
1 4 4 5 6 6 8 8 10 11 12

post: <= v h > v

Loop invariant:

entries h and below are <= v

entries t and above are > v

entries between h and t are sorted

Binary search: an O(log n) algorithm

inv: b [0 … h: <= v | ? | t … b.length = n: > v]

```
h= –1;  t= b.length;
while  (h != t–1) {
    int  e=  (h+t)/2;
    if (b[e] <= v)  h=  e;
    else  t=  e;
}
```

Initially $t - h = 2^k$
Loop iterates exactly k times

Suppose initially: b.length = $2^k – 1$
Initially, h = -1, t = 2^k -1, $t - h = 2^k$
Can show that one iteration sets h or t so that $t - h = 2^{k-1}$
e.g. Set e to $(h+t)/2 = (2^k – 2)/2 = 2^{k-1} – 1$
Set t to e, i.e. to $2^{k-1} – 1$
Then $t - h = 2^{k-1} – 1 + 1 = 2^{k-1}$
Careful calculation shows that:
each iteration halves t – h !!

Binary search: an O(log n) algorithm
Search array with 32767 elements, only 15 iterations!

Bsearch:

```
h= –1;  t= b.length;
while  (h != t–1) {
    int  e=  (h+t)/2;
    if (b[e] <= v)  h=  e;
    else  t=  e;
}
```

If $n = 2^k$, k is called log(n)
That's the base 2 logarithm

n	**log(n)**
$1 = 2^0$	0
$2 = 2^1$	1
$4 = 2^2$	2
$8 = 2^3$	3
$31768 = 2^{15}$	15

Each iteration takes constant time (a few assignments and an if).
Bsearch executes ~log n iterations for an array of size n. So the number of assignments and if-tests made is proportional to log n. Therefore, Bsearch is called an order log n algorithm, written O(log n). (We'll formalize this notation later.)

Linear search: Find first position of v in b (if in)

Store in h to truthify:

pre: b | 0 … b.length: ?

post: b | 0 … h: v not here | h … b.length: ? | and h = b.length or b[h] = v

inv: b | 0 … h: v not here | h … b.length: ?

```
h= 0;
while (h != b.length && b[h] != v)
    h= h+1;
```

B: h != b.length && b[h] != v

loopy question 1? h= 0;

loopy question 2?
Stop when this is true

loopy question 3? h= h+1;

loopy question 4? OK!

Linear search: Find first position of v in b (if in)

Store in h to truthify:

pre: b | 0 … b.length: ?

post: b | 0 … h: v not here | h … b.length: ? | and h = b.length or b[h] = v

inv: b | 0 … h: v not here | h … b.length: ?

```
h= 0;
while (h != b.length && b[h] != v)
    h= h+1;
```

Worst case: for array of size n, requires n iterations, each taking constant time.
Worst-case time: O(n).

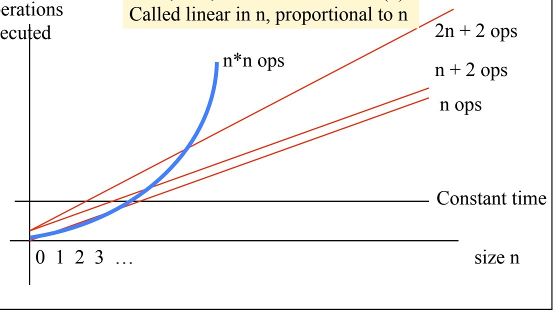
Expected or average time? n/2 iterations. O(n/2) —is also O(n)

Looking at execution speed

Process an array of size n

Number of operations executed

2n+2, n+2, n are all “order n” O(n)
Called linear in n, proportional to n



InsertionSort

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

inv: b | 0 … i: processed | i … b.length: ?

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

---

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

inv: b | 0 … i: sorted | i … b.length: ?

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

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

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

Will take time proportional to the number of swaps needed

What to do in each iteration?

inv: b | 0 … i: sorted | i … b.length: ?

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

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

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

InsertionSort

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

Many people sort cards this way

Works well when input is *nearly sorted*

Note English statement in body. **Abstraction**. Says **what** to do, not **how.**

This is the best way to present it. Later, show how to implement that with a loop

InsertionSort

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

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

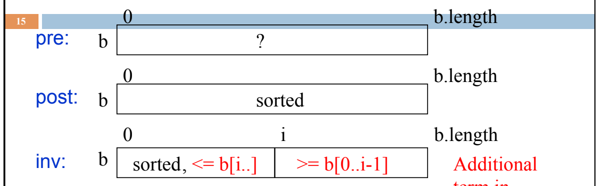
Pushing b[i] down can take i swaps. Worst case takes

1 + 2 + 3 + … n-1 = (n-1)*n/2

Swaps.

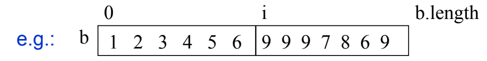
Let n = b.length

SelectionSort



Additional term in invariant

Keep invariant true while making progress?



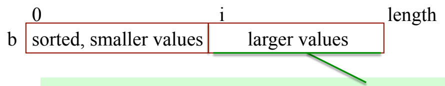
Increasing i by 1 keeps inv true only if b[i] is min of b[i..]

SelectionSort

```
//sort b[], an array of int
// inv: b[0..i-1] sorted
//       b[0..i-1]  <=  b[i..]
for (int i= 1; 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
- Worst-case O(n^2)
- Best-case O(n^2)
- Expected-case O(n^2)

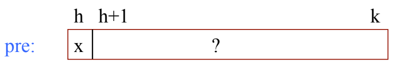


Each iteration, swap min value of this section into b[i]

Swapping b[i] and b[m]

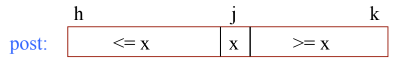
```
// Swap b[i] and b[m]
int t= b[i];
b[i]= b[m];
b[m]= t;
```

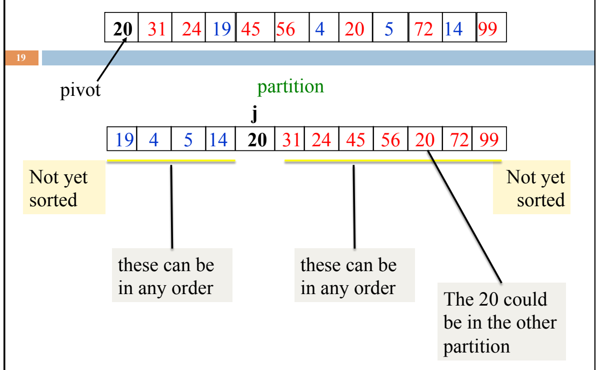
Partition algorithm of quicksort



x is called the pivot

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





Not yet sorted

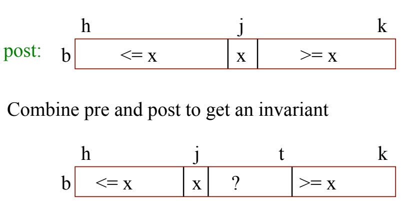
Not yet sorted

these can be in any order

these can be in any order

The 20 could be in the other partition

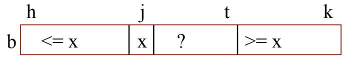
Partition algorithm



Combine pre and post to get an invariant

invariant needs at least 4 sections

Partition algorithm



```
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;
   }
}
```

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

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

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

---

QuickSort procedure

```
/** 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);
}
```

Base case

Function does the partition algorithm and returns position j of pivot

---

QuickSort

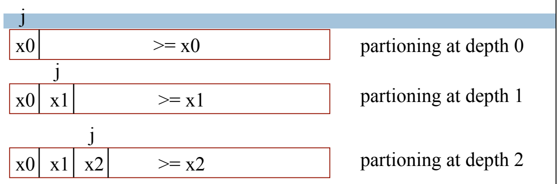
Quicksort developed by Sir Tony Hoare (he was knighted by the Queen of England for his contributions to education and CS).

81 years old.

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

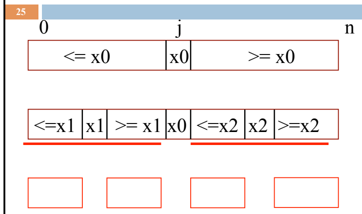
Later, he saw a draft of the new language Algol 58 (which became Algol 60). It had recursive procedures. First time in a procedural programming language. "Ah!," he said. "I know how to write it better now." 15 minutes later, his colleague also understood it.

Worst case quicksort: pivot always smallest value



```
/** 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);
```

Best case quicksort: pivot always middle value



depth 0. 1 segment of size ~n to partition.

Depth 2. 2 segments of size ~n/2 to partition.

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

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

Average time for Quicksort: n log n. Difficult calculation

QuickSort procedure

```
/** 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: quadratic
Average-case: O(n log n)

Worst-case space: O(n*n)! --depth of recursion can be n
Can rewrite it to have space O(log n)
Average-case: O(n * log n)

Partition algorithm

Key issue:
How to choose a *pivot*?

Choosing pivot
- Ideal pivot: the median, since it splits array in half

But computing median of unsorted array is O(n), quite complicated

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

Quicksort with logarithmic space

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

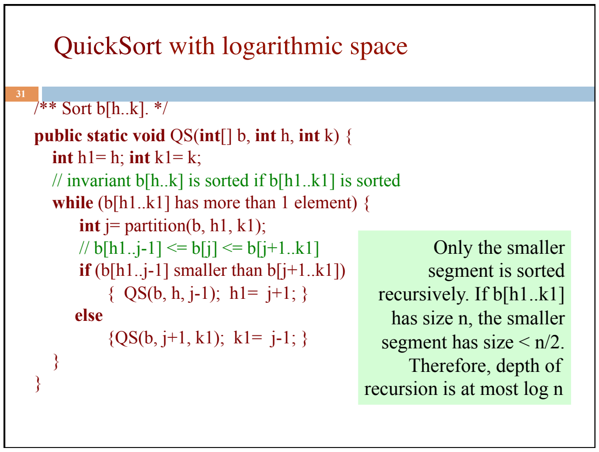
Quicksort with logarithmic space

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

QuickSort with logarithmic space

```
/** 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

```
/** 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