SEARCHING AND SORTING
HINT AT ASYMPTOTIC COMPLEXITY

Lecture 10
CS2110 – Fall 2016

Miscellaneous

- A3 due Monday night. Group early! Only 325 views of the piazza A3 FAQ yesterday morning. Everyone should look at it.
- Pinned Piazza note on Supplemental study material. @281. Contains material that may help you study certain topics. It also talks about *how* to study.

Developing methods

We use Eclipse to show the development of A2 function evaluate. Here are important points to take away from it.

1. If similar code will appear in two or more places, consider writing a method to avoid that duplication.
2. If you introduce a new method, write a specification for it!
3. Before writing a loop, write a loop invariant for it.
4. Have a loop exploit the structure of the data it processes.
5. Don't expect your first attempt to be perfect. Just as you rewrite and rewrite an essay, we rewrite programs.

Search as in problem set: b is sorted

pre: b

0	b.length
?	

post: b

0	h	b.length
<= v	> v	

inv: b

0	h	t	b.length
<= v	?	> v	

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

Methodology:

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

Practice doing this!

Search as in problem set: b is sorted

pre: b

0	b.length
?	

post: b

0	h	b.length
$\leq$ v	> v	

inv: b

0	h	t	b.length
$\leq$ v	?	> v	

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

b[0] > v?   one iteration.

b[b.length-1] $\leq$ 0?
b.length iterations
Worst case: time is proportional to size of b

Since b is sorted, can cut ? segment in half. As a dictionary search

Search as in problem set: b is sorted

	0		b.length
pre: b	?		

	0	h		b.length
post: b	<= v	> v		

	0	h	t		b.length
inv: b	<= v	?	> v		

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

	0	h	e		t
inv: b	<= v	?		?	> v

	0	h	e		t
b	<= v	≤ v	≤v	?	> v

	0	h	e		t
b	<= v	?	>v	> v	> v

Binary search: an O(log n) algorithm

inv: b

0 h	t	
<= v	?	> v

b.length = n

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

inv: b

0 h	e		t	
<= v	?		?	> v

n = 2**k ?   About k iterations

Each iteration cuts the size of the ? segment in half.

Time taken is proportional to k, or log n.

A *logarithmic algorithm*

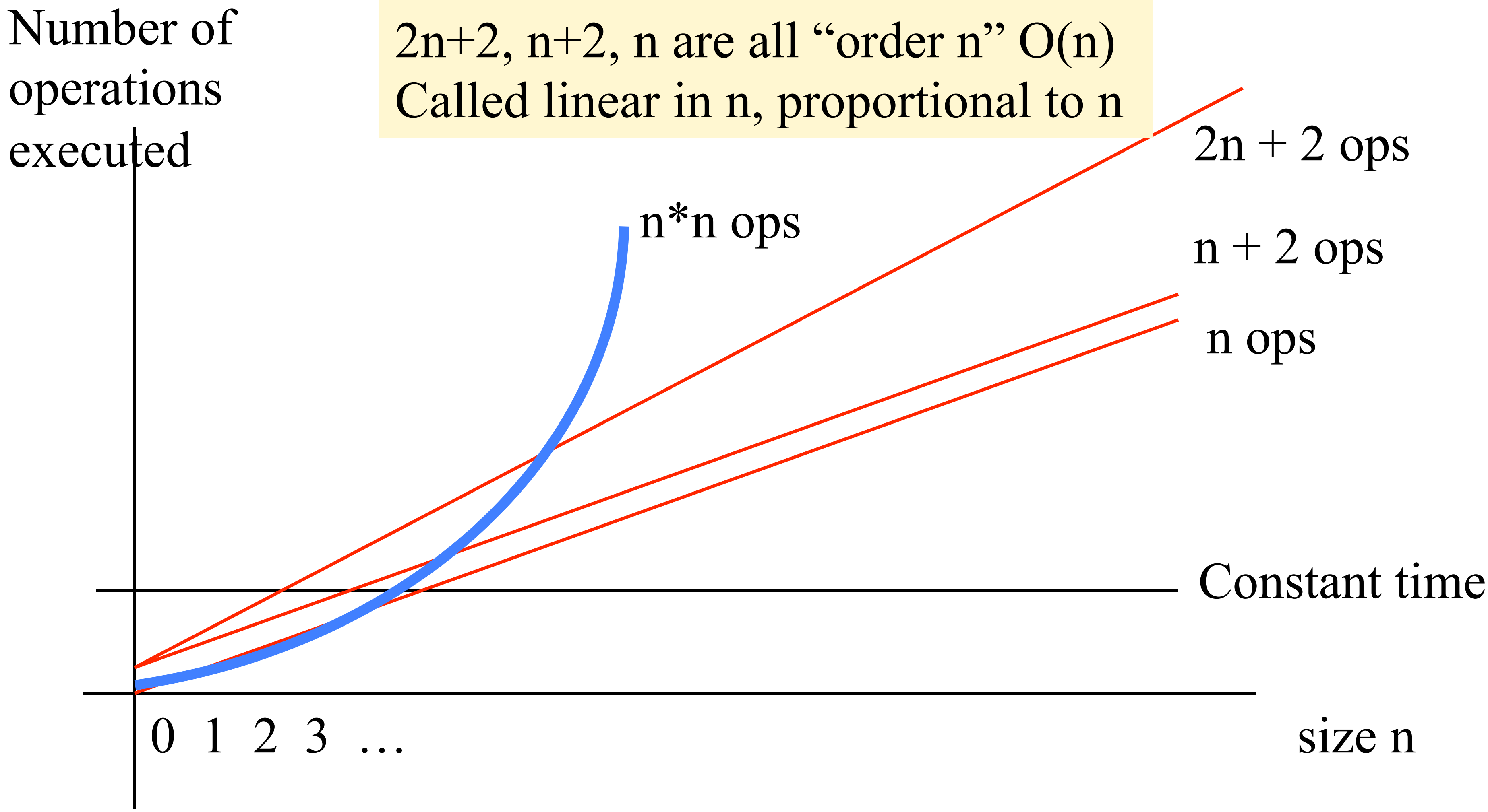
Write as O(log n)

[explain notation next lecture]

Looking at execution speed

Process an array of size n

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



InsertionSort

	0		b.length
pre: b	?		

	0		b.length
post: b	sorted		

	0	i	b.length
inv: b	sorted	?	

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

	0	i	b.length
inv: b	processed	?	

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

for (int i= 0; i < b.length; i= i+1) { maintain invariant }

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

		0	i	b.length
inv:	b	sorted	?	

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

		0	i	b.length
	b	2 3 5 5 5	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?

	0	i	b.length
inv: b	sorted	?	

	0	i	b.length
e.g. b	2 5 5 5 7	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

	0	i	b.length
b	2 3 5 5 5 7	?	

InsertionSort


```
// 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]
}
```

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. We expect you to present it this way when asked.

Later, show how to implement that with a loop

InsertionSort


```
// Q: b[0..i-1] is sorted
// 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;
}
// R: b[0..i] is sorted
```

start?

stop?

progress?

maintain invariant?

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

How to write nested loops


```
// 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]
}
```

```
// 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;
    }
}
```

Present algorithm like this

If you are going to show implementation, *put in "WHAT IT DOES" as a comment*

InsertionSort


```
// 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

$$1 + 2 + 3 + \dots n\text{-}1 = (n-1)*n/2$$

Swaps.

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

O(f(n)) : Takes time proportional to f(n). Formal definition later

Let n = b.length

SelectionSort

	0	b.length
pre: b	?	

	0	b.length
post: b	sorted	

	0	i	b.length
inv: b	sorted , <= b[i..]	>= b[0..i-1]	

Additional term in invariant

Keep invariant true while making progress?

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

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

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

	0	i	length
b	sorted, smaller values	larger values	

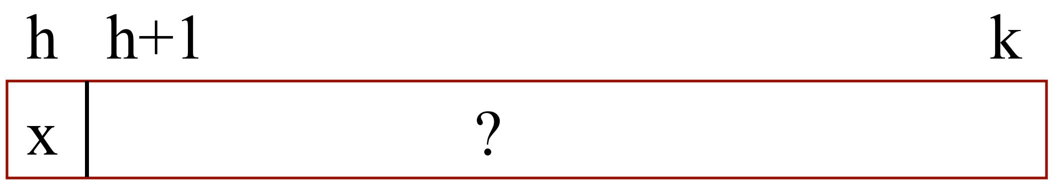
Each iteration, swap min value of this section into b[i]

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

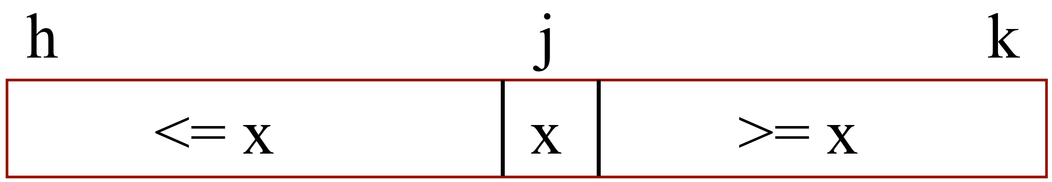
pre:



x is called the pivot

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

post:



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

pivot

partition

19	4	5	14	**20** (**j**)	31	24	45	56	20	72	99

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

		h	h+1	k
pre:	b	x	?	

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

Combine pre and post to get an invariant

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

invariant needs at least 4 sections

Partition algorithm

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

```
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

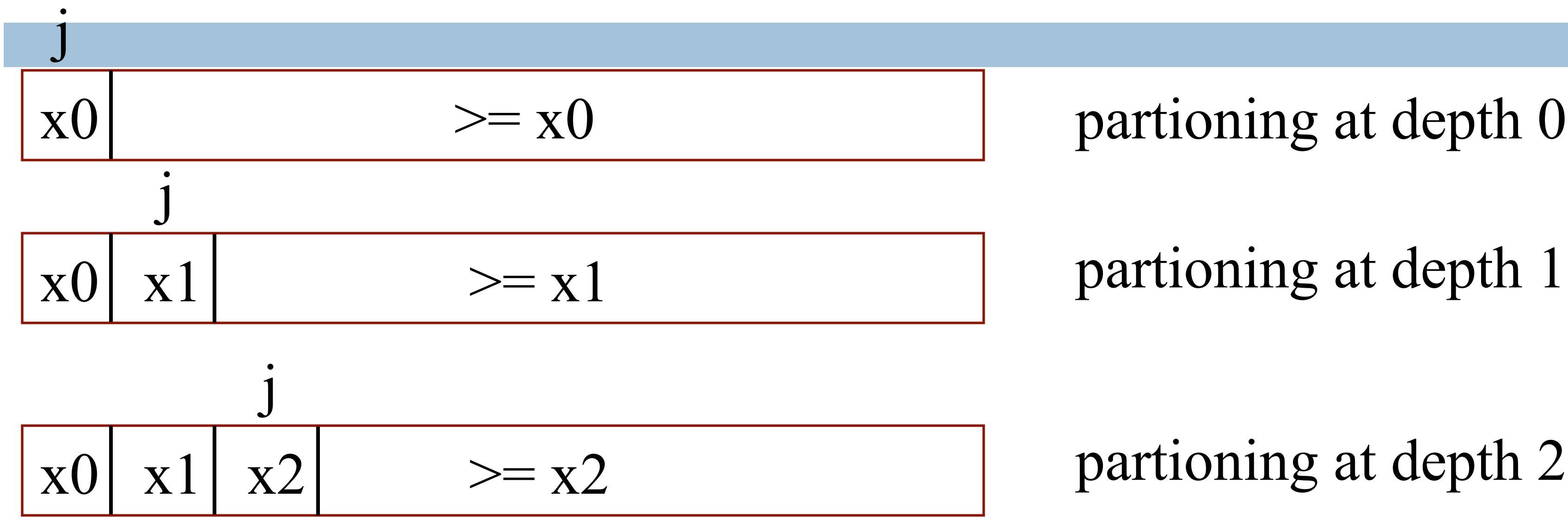
Quicksort developed by Sir Tony Hoare (he was knighted by the Queen of England for his contributions to education and CS).

81 years old.

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

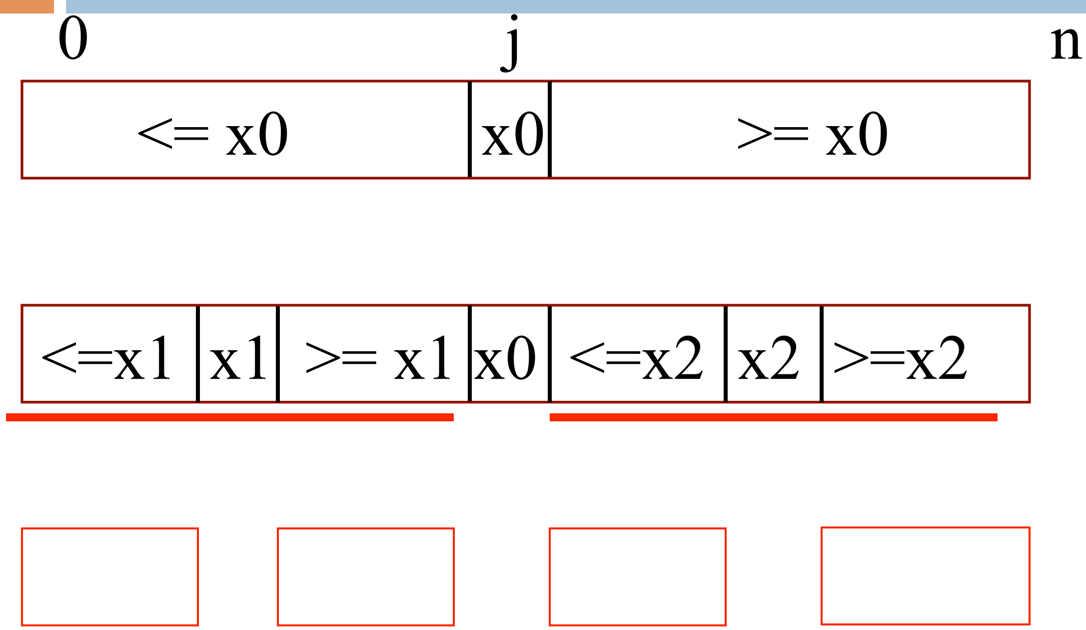
Later, he saw a draft of the new language Algol 58 (which became Algol 60). It had recursive procedures. First time in a procedural programming language. “Ah!,” he said. “I know how to write it better now.” 15 minutes later, his colleague also understood it.

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. We may show you this later. Not today!

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
    }
}
```

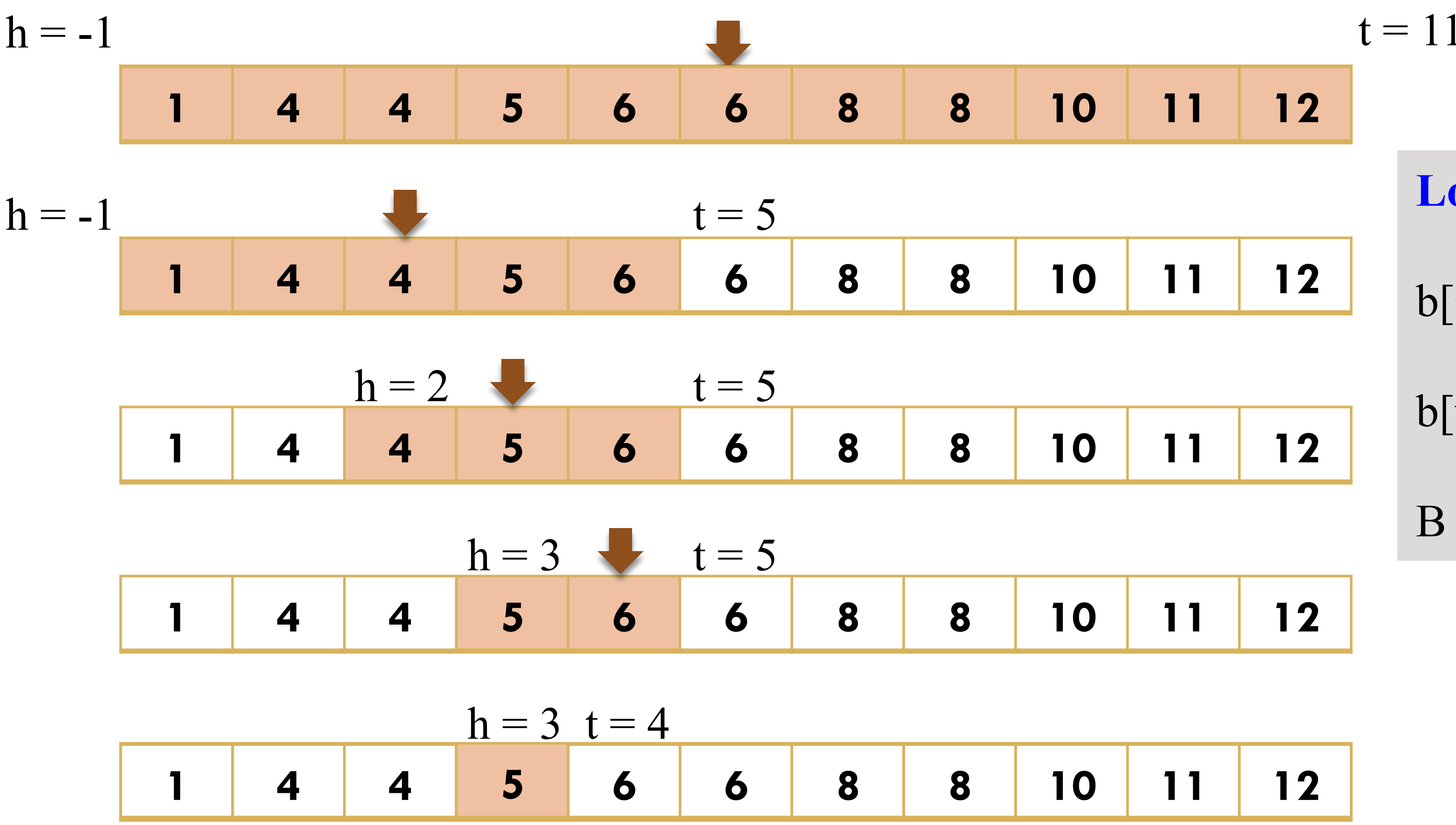
QuickSort with logarithmic space


```
/** 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

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

pre: array is sorted



post: <= v h > v

Loop invariant:

b[0..h] <= v

b[t..] > v

B is sorted