Data Structures and Algorithms for Engineers

Module 3: Searching and Sorting Algorithms

Lecture 2: Not-in-place sorts: quicksort, mergesort. Characteristics of a good sort.

David Vernon
Carnegie Mellon University Africa

vernon@cmu.edu
www.vernon.eu

Quicksort

The Quicksort algorithm was developed by C.A.R. Hoare.
It has the best average behaviour in terms of complexity:

Average case: $O(n \log_2 n)$

Worst case: $O(n^2)$

Quicksort

- Given a list of elements
- take a partitioning element (called a "pivot")
- and create two (sub)lists
  1. Left sublist: all elements are less than partitioning element,
  2. Right sublist: all elements are greater than it
- Now repeat this partitioning effort on each of these two sublists
- This is a divide-and-conquer strategy

Quicksort

- And so on in a recursive manner until all the sublists are empty, at which point the (total) list is sorted

- Partitioning can be effected by
  - scanning left to right
  - scanning right to left
  - iinterchanging elements in the wrong parts of the list

- The partitioning element is then placed between the resultant sublists
  - which are then partitioned in the same manner

Implementation of Quicksort()

In pseudo-code first

```
If anything to be partitioned
   choose a pivot
   DO
      scan from left to right until we find an element
      > pivot: i points to it

      scan from right to left until we find an element
      <= pivot: j points to it

      IF  i < j
         exchange ith and jth element
   WHILE i <= j
```

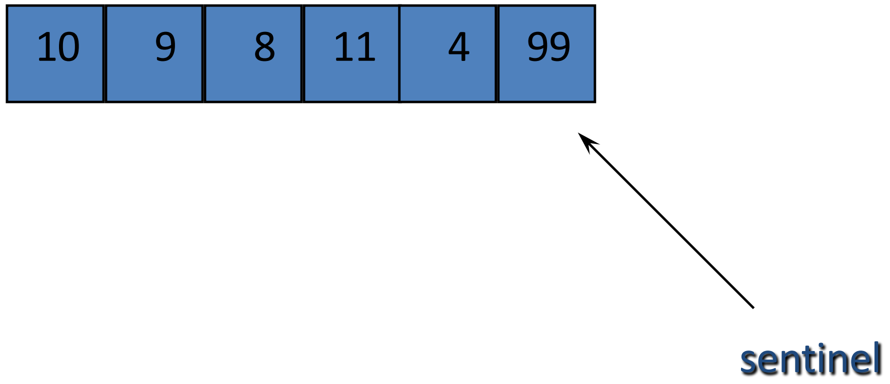
Implementation of Quicksort()

```
/* simple quicksort to sort an array of integers */

void quicksort (int A[], int L, int R) {
   int i, j, pivot;

   /* assume A[R] contains a number > any element,    */
   /* i.e., it is a sentinel.                         */
```

Implementation of Quicksort()

```
    if ( R > L) { // if R==L, it is a list with just one element!!
       i = L; j = R;
       pivot = A[i];
       do {
          while (A[i] <= pivot)
             i=i+1;
          while ((A[j] >= pivot) && (j>L))
             j=j-1;
          if (i < j) {
             exchange(A[i],A[j]); /* between partitions */
             i = i+1; j = j-1;
          }
       } while (i <= j);
       exchange(A[L], A[j]); /* reposition pivot */
       quicksort(A, L, j);
       quicksort(A, i, R);   /*includes sentinel*/
    }
}
```

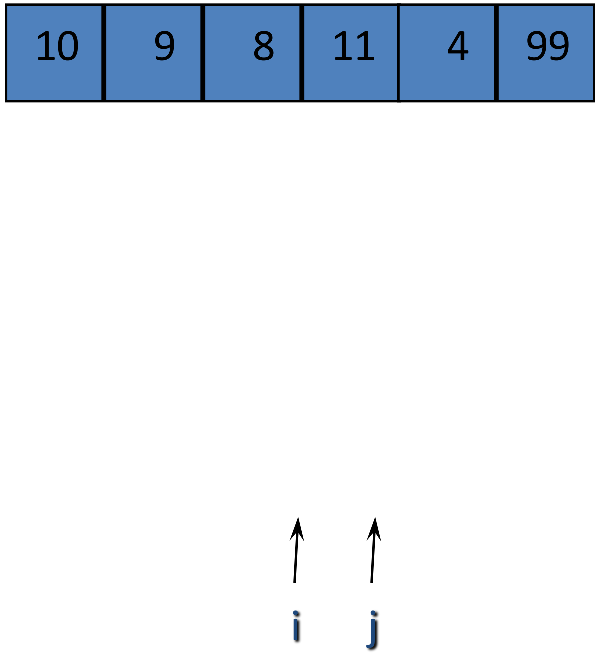
Quicksort



10	9	8	11	4	99

sentinel

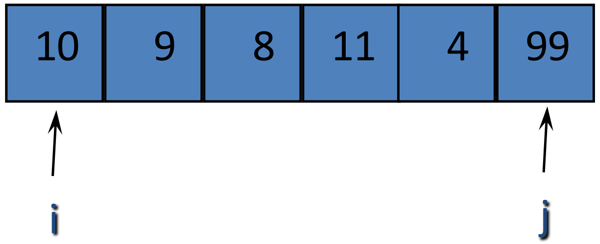
Quicksort



10	9	8	11	4	99

```
QS(A, , )

L:
R:
i:
j:
pivot:
```

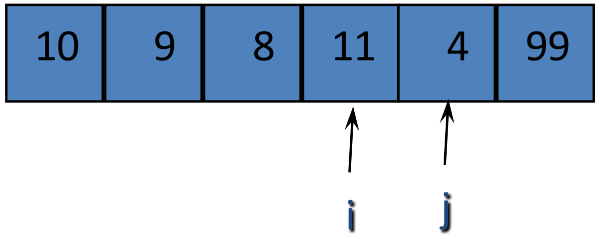
Quicksort



10	9	8	11	4	99

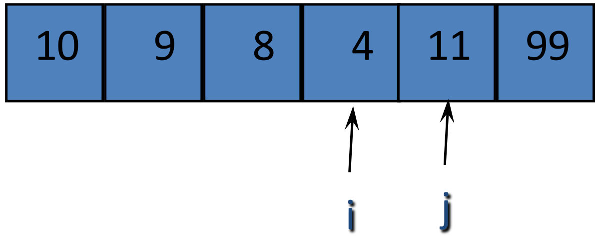
```
QS(A,1,6)

L:       1
R:       6
i:       1
j:       6
pivot: 10
```

Quicksort



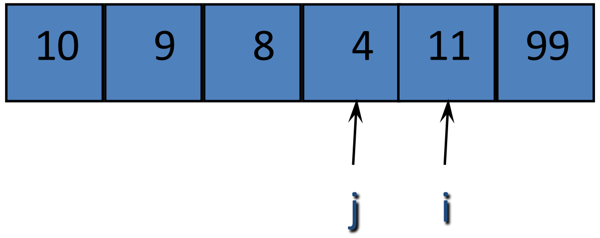
```
QS(A,1,6)

L:      1
R:      6
i:      1 2 3 4
j:      6 5
pivot: 10
```

Quicksort



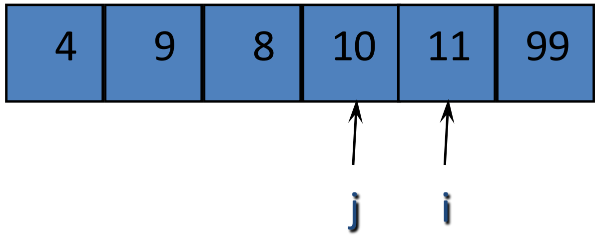
```
QS(A,1,6)

L:       1
R:       6
i:       1 2 3 4
j:       6 5
pivot: 10
```

Quicksort



```
QS(A,1,6)

L:       1
R:       6
i:       1 2 3 4 5
j:       6 5 4
pivot: 10
```

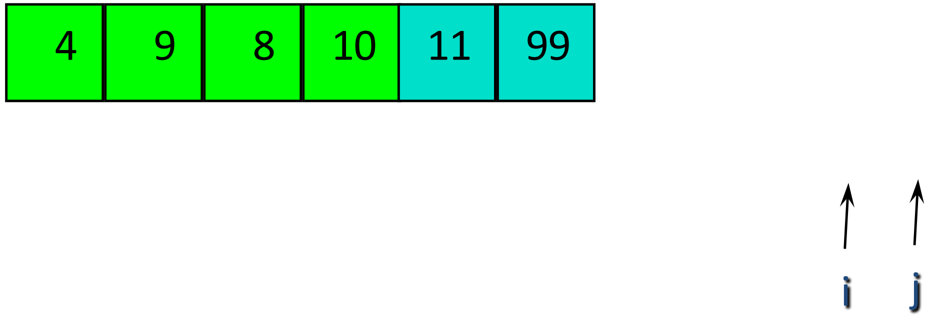
Quicksort



4	9	8	10	11	99

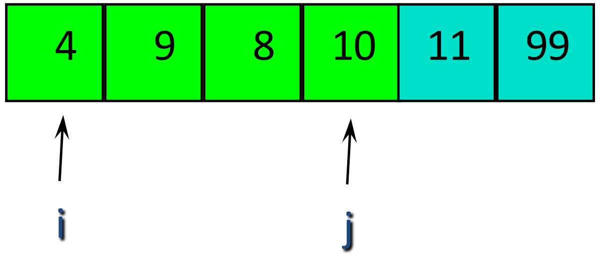
```
QS(A,1,6)

L:       1
R:       6
i:       1 2 3 4 5
j:       6 5 4
pivot: 10
```

Quicksort



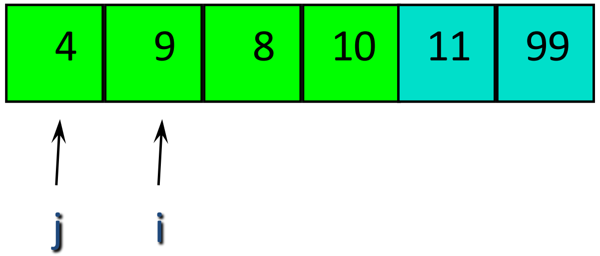
QS(A,1,6)		QS(A,1,4)		QS(A,5,6)	
L:	1	L:	1	L:	5
R:	6	R:	4	R:	6
i:	1 2 3 4 5	i:		i:	
j:	6 5 4	j:		j:	
pivot:	10	pivot:	4	pivot:	11

Quicksort



	QS(A,1,6)	QS(A,1,4)	QS(A,5,6)
L:	1	1	5
R:	6	4	6
i:	1 2 3 4 5	1	
j:	6 5 4	4	
pivot:	10	4	11

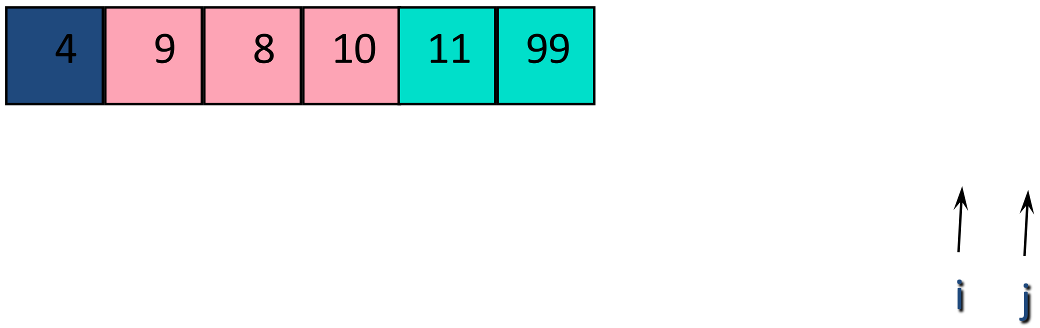
Quicksort



4	9	8	10	11	99

	QS(A,1,6)	QS(A,1,4)	QS(A,5,6)
L:	1	1	5
R:	6	4	6
i:	1 2 3 4 5	1 2	
j:	6 5 4	4 3 4 1	
pivot:	10	4	11

Quicksort



4	9	8	10	11	99

QS(A,1,1)

L: 1
R: 1
i:
j:
pivot: 4

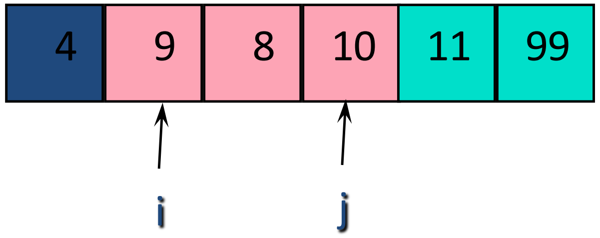
QS(A,2,4)

L: 2
R: 4
i:
j:
pivot: 9

QS(A,5,6)

L: 5
R: 6
i: 5
j: 6
pivot: 11

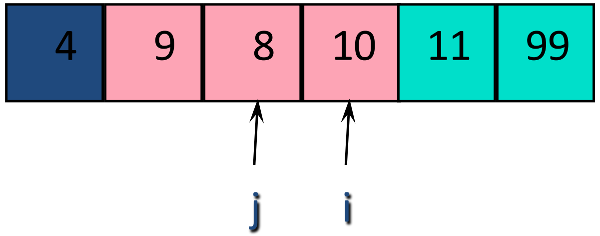
Quicksort



```
QS(A,2,4)

L:      2
R:      4
i:      2
j:      4
pivot:  9
```

```
QS(A,5,6)

L:      5
R:      6
i:      5
j:      6
pivot:  11
```

Quicksort



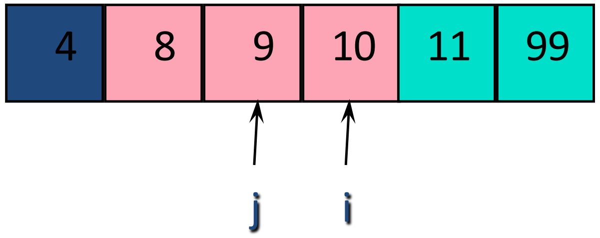
QS(A,2,4)

L: 2
R: 4
i: 2 3 4
j: 4 3
pivot: 9

QS(A,5,6)

L: 5
R: 6
i: 5
j: 6
pivot: 11

Quicksort



4	8	9	10	11	99

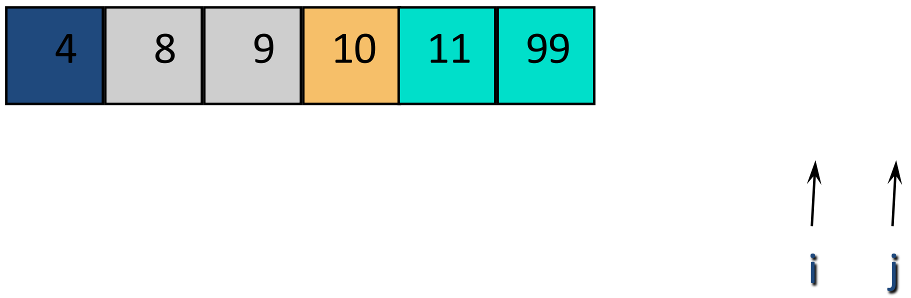
QS(A,2,4)

L: 2
R: 4
i: 2 3 4
j: 4 3
pivot: 9

QS(A,5,6)

L: 5
R: 6
i: 5
j: 6
pivot: 11

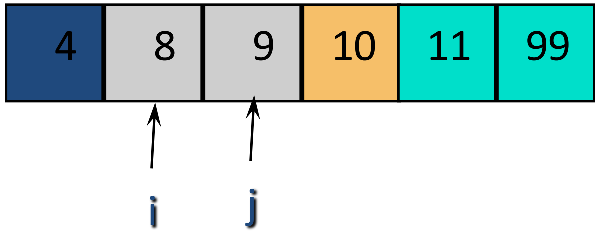
Quicksort



4	8	9	10	11	99

	QS(A,2,4)	QS(A,2,3)	QS(A,4,4)	QS(A,5,6)
L:	2	2	4	5
R:	4	3	4	6
i:	2 3 4			5
j:	4 3			6
pivot:	9	8	10	11

Quicksort



QS(A,2,3)

L: 2
R: 3
i: 2
j: 3
pivot: 8

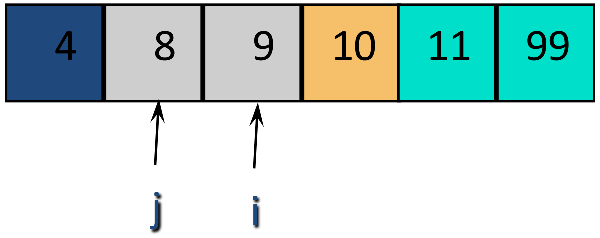
QS(A,4,4)

L: 4
R: 4
i:
j:
pivot: 10

QS(A,5,6)

L: 5
R: 6
i: 5
j: 6
pivot: 11

Quicksort

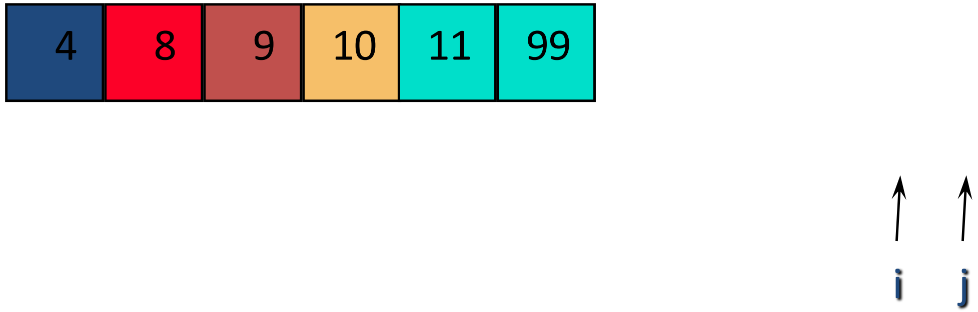


```
QS(A,2,3)

L:       2
R:       3
i:       2  3
j:       3  2
pivot:   8
```

```
QS(A,4,4)

L:       4
R:       4
i:
j:
pivot:   10
```

```
QS(A,5,6)

L:       5
R:       6
i:       5
j:       6
pivot:   11
```

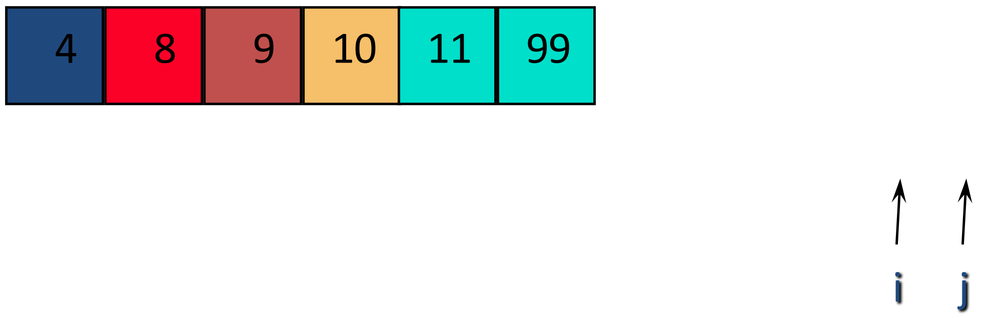
Quicksort



4	8	9	10	11	99

	QS(A,2,3)	QS(A,2,2)	QS(A,3,3)	QS(A,4,4)	QS(A,5,6)
L:	2	2	3	4	5
R:	3	2	3	4	6
i:	2 3				5
j:	3 2				6
pivot:	8	8	9	10	11

Quicksort



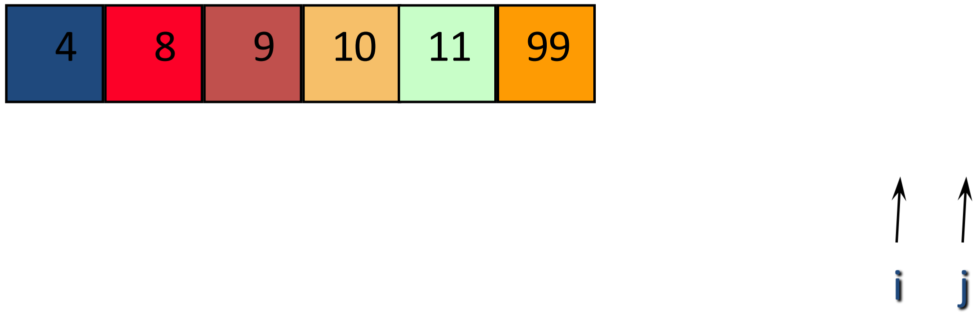
QS(A,4,4)

L: 4
R: 4
i:
j:
pivot: 10

QS(A,5,6)

L: 5
R: 6
i: 5
j: 6
pivot:
11

Quicksort

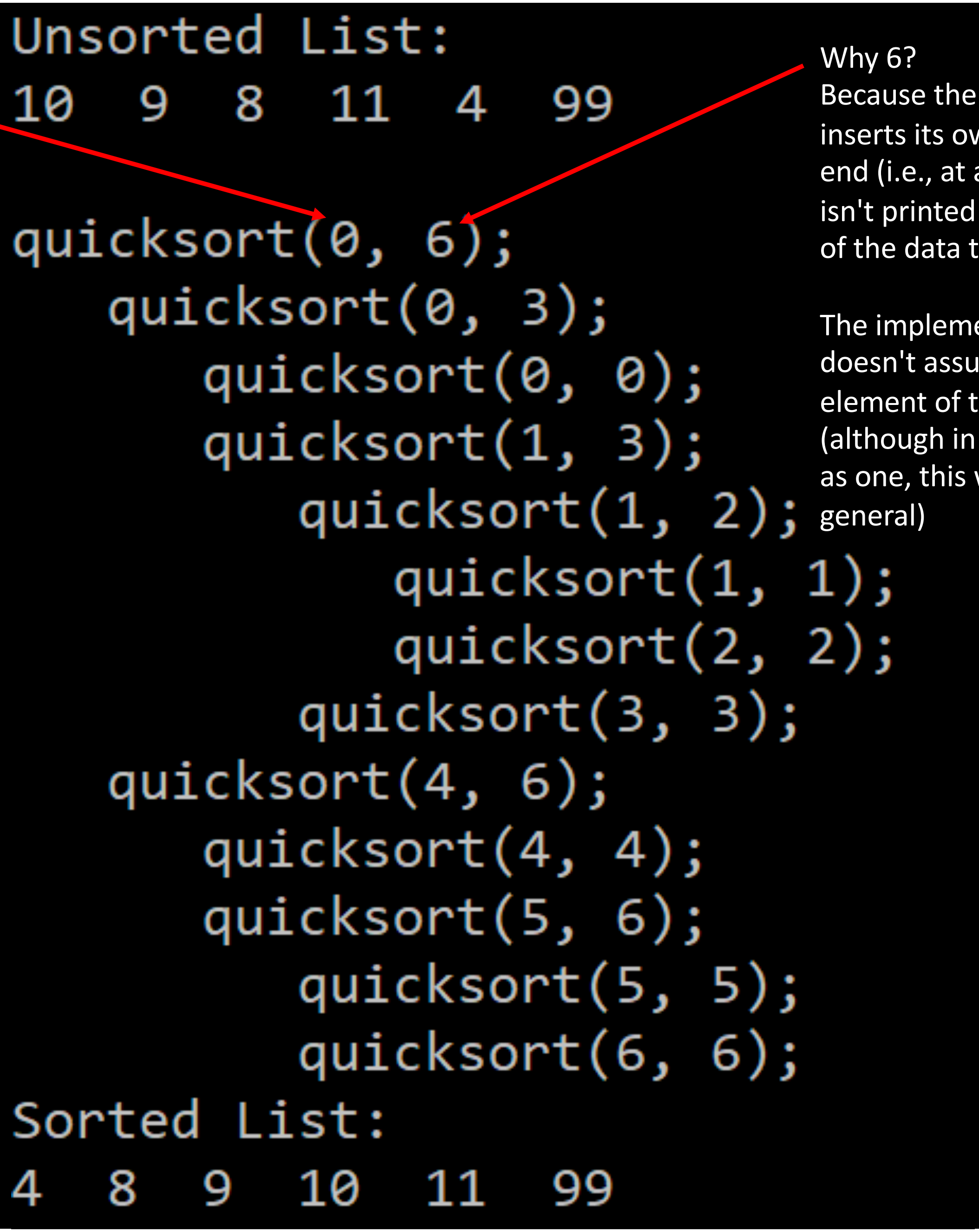


4	8	9	10	11	99

QS(A,5,6)		QS(A,5,5)		QS(A,6,6)	
L:	5	L:	5	L:	6
R:	6	R:	5	R:	6
i:	5	i:		i:	
j:	6	j:		j:	
pivot:	11	pivot:	11	pivot:	99

Why 0?

Because the implementation uses index 0 for the first element in the list



```
Unsorted List:
10   9   8   11   4   99

quicksort(0, 6);
   quicksort(0, 3);
      quicksort(0, 0);
      quicksort(1, 3);
         quicksort(1, 2);
            quicksort(1, 1);
            quicksort(2, 2);
         quicksort(3, 3);
   quicksort(4, 6);
      quicksort(4, 4);
      quicksort(5, 6);
         quicksort(5, 5);
         quicksort(6, 6);
Sorted List:
4   8   9   10   11   99
```

Why 6?
Because the implementation inserts its own sentinel at the end (i.e., at array index 6) but it isn't printed because it's not part of the data to be sorted.

The implementation doesn't assume that the last element of the data is a sentinel (although in this case it could act as one, this won't be true in general)

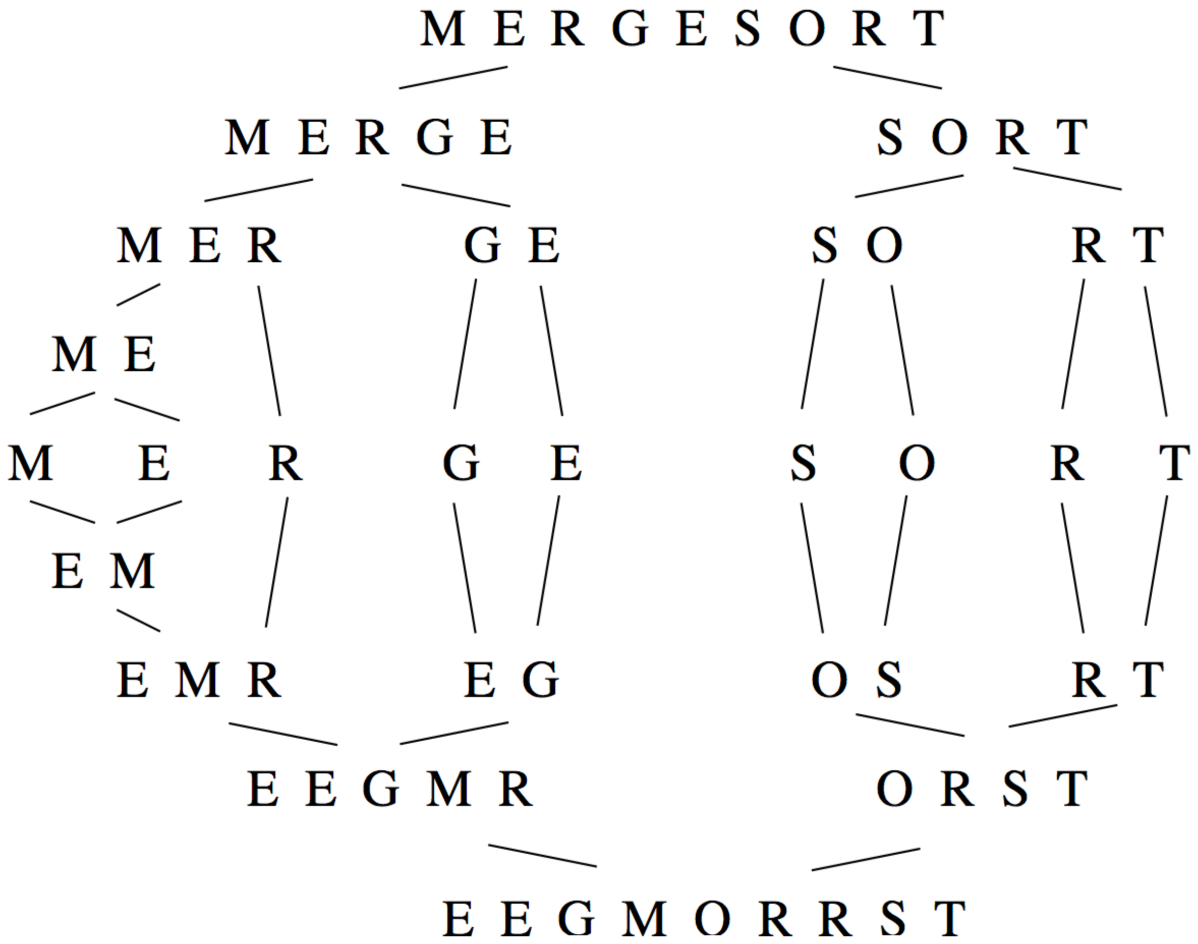
Quicksort

- Performance depends on which element is selected as the pivot
- The worst-case occurs when the list is sorted, and the left-most element is selected as the pivot
- Space complexity is $O(n^2)$ in the worst case

Mergesort

- Divide-and-conquer, recursive, $O(n \log n)$
- Recursively partition the list into two lists L1 and L2
  - L1 and L2 approx. n/2 elements each
- Stop when we have a collection of lists of 1 element
- Now, each L1 and L2 is merged into a list S
  - the elements of L1 and L2 are put in S in order
- Pairs of sorted lists S1 and S2 are, in turn, merged as we ascend back up through the recursion

Mergesort



M E R G E S O R T

M E R G E S O R T

M E R G E S O R T

M E

M E R G E S O R T

E M

E M R E G O S R T

E E G M R O R S T

E E G M O R R S T

Merge Sort

```
mergesort(item_type s[], int low, int high)  {

   int i;       /* counter                  */
   int middle; /* index of middle element */

   if (low < high) {
      middle = (low+high)/2;
      mergesort(s,low,middle);
      mergesort(s,middle+1,high);
      merge(s, low, middle, high);
   }
}
```

Merge Sort

- The efficiency of mergesort depends on how we combine the two sorted halves into a single sorted list

- The key is to realize that each half (i.e., each sublist) is sorted

- So we just have to repeatedly do the following

  - Take the ”front” element of either one list or the other (depending on which is smaller) and

  - Move it to the merged list (thus keeping the elements in order)

Merge Sort

```
merge(item_type s[], int low, int middle, int high){
   int i;                  /* counter                      */
   queue buffer1, buffer2; /* to hold elements for merging */
   init_queue(&buffer1);
   init_queue(&buffer2);
   for (i=low; i<=middle; i++) enqueue(&buffer1,s[i]);
   for (i=middle+1; i<=high; i++) enqueue(&buffer2,s[i]);

   i = low;
   while (!(empty_queue(&buffer1) &&  !(empty_queue(&buffer2)) {
   // Alt: while (!(empty_queue(&buffer1) || empty_queue(&buffer2))) {
      if (headq(&buffer1) <= headq(&buffer2))
         s[i++] = dequeue(&buffer1);
      else
         s[i++] = dequeue(&buffer2);
   }
   while (!empty_queue(&buffer1)) s[i++] = dequeue(&buffer1);
   while (!empty_queue(&buffer2)) s[i++] = dequeue(&buffer2);
}
```

Mergesort

Why is mergesort $O(n \log n)$?

How many times do we merge and how big are the data sets?

Let's assume that n is a power of two

At level 0
2^1 calls to `mergesort` & merge 2^1 lists of size $\sim n/2$

At level 1
2^2 calls to `mergesort` & merge 2^2 lists of size $\sim n/4$
…
At level k
2^{k+1} calls to `mergesort` & merge 2^{k+1} lists of size $\sim n/ 2^{k+1}$

Mergesort

How many levels k?

$k = \log_2 n$, *e.g.* if $n = 8$, $k = 3$

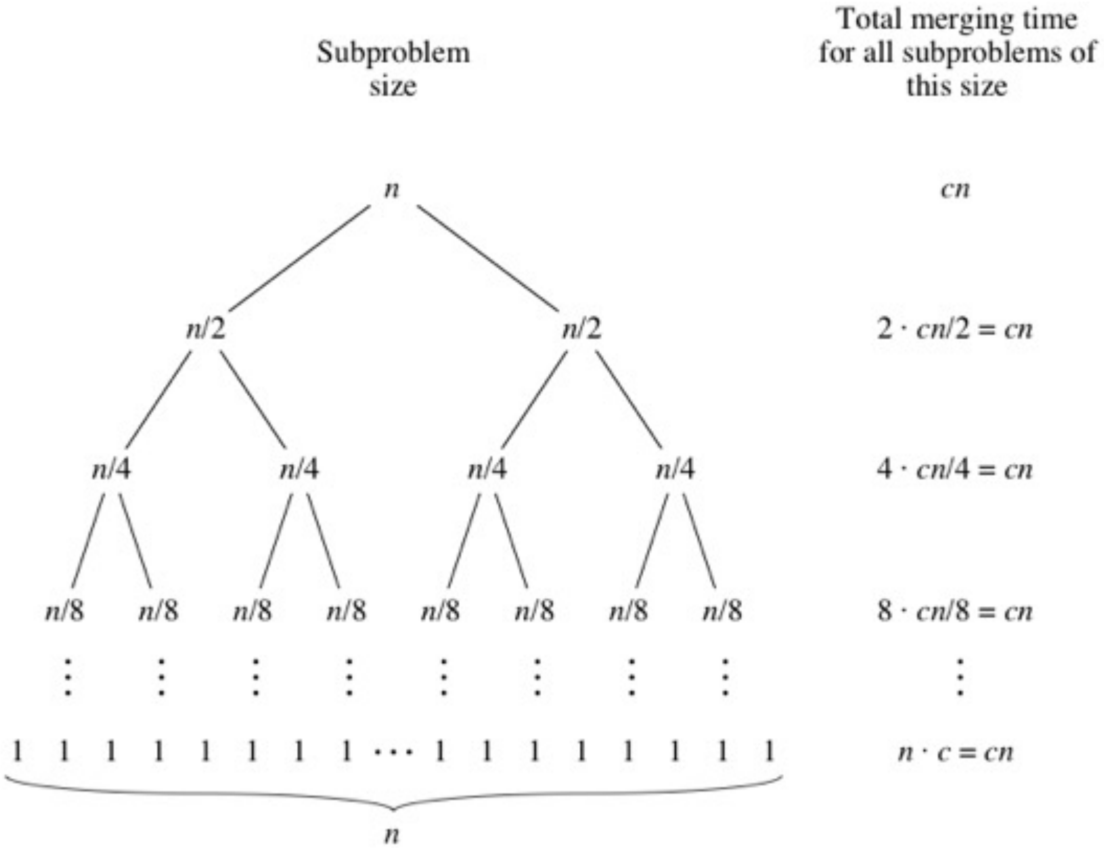
At level k, the sub-lists are of size 1.

So, we merge on $k = \log_2 n$ levels (level $0 - k\text{-}1$)

Each level we merge 2^{k+1} lists of size $\sim n / 2^{k+1}$ i.e., total size $\sim n$

So, the total complexity is $O(n \log_2 n)$

Mergesort



https://www.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/analysis-of-merge-sort

Characteristics of a Good Sort

Code Complexity

- Short, simple algorithms are appealing because they are easy to implement and debug
- Algorithms that can easily be applied to all data types are convenient to use but often come at the cost of implementation complexity
- Although they may not be as fast as more specialized algorithms, simple algorithms are always appealing especially when maintenance is an issue

Characteristics of a Good Sort

Stability

- A stable sorting algorithm maintains the relative order of records with equal keys
  - Let records R and S have the same key
  - R appears before S in the original list,
  - R will always appear before S in the sorted list
- This is particularly important when sorting based on multiple keys

Characteristics of a Good Sort

Stability

- Assume that the following pairs of numbers are to be sorted by their first component (two different results are possible)

(4, 2) (3, 7) (3, 1) (5, 6)

(3, 7) (3, 1) (4, 2) (5, 6) (stable: order maintained)
(3, 1) (3, 7) (4, 2) (5, 6) (unstable: order changed)

- Unstable sorting algorithms change the relative order of records with equal keys, but stable sorting algorithms do not

Characteristics of a Good Sort

Stability

- Unstable sorting algorithms can be specially implemented to be stable
- Stability usually comes with an additional computational cost

```
DEFINE HALFHEARTEDMERGESORT(LIST):
    IF LENGTH(LIST) < 2:
        RETURN LIST
    PIVOT = INT(LENGTH(LIST) / 2)
    A = HALFHEARTEDMERGESORT(LIST[:PIVOT])
    B = HALFHEARTEDMERGESORT(LIST[PIVOT:])
    // UMMMMM
    RETURN [A, B] // HERE. SORRY.
```

```
DEFINE FASTBOGOSORT(LIST):
    // AN OPTIMIZED BOGOSORT
    // RUNS IN O(N LOG N)
    FOR N FROM 1 TO LOG(LENGTH(LIST)):
        SHUFFLE(LIST):
        IF ISSORTED(LIST):
            RETURN LIST
    RETURN "KERNEL PAGE FAULT (ERROR CODE: 2)"
```

```
DEFINE JOBINTERVIEWQUICKSORT(LIST):
    OK SO YOU CHOOSE A PIVOT
    THEN DIVIDE THE LIST IN HALF
    FOR EACH HALF:
        CHECK TO SEE IF IT'S SORTED
            NO, WAIT, IT DOESN'T MATTER
        COMPARE EACH ELEMENT TO THE PIVOT
            THE BIGGER ONES GO IN A NEW LIST
            THE EQUAL ONES GO INTO, UH
            THE SECOND LIST FROM BEFORE
        HANG ON, LET ME NAME THE LISTS
            THIS IS LIST A
            THE NEW ONE IS LIST B
        PUT THE BIG ONES INTO LIST B
        NOW TAKE THE SECOND LIST
            CALL IT LIST, UH, A2
        WHICH ONE WAS THE PIVOT IN?
        SCRATCH ALL THAT
        IT JUST RECURSIVELY CALLS ITSELF
        UNTIL BOTH LISTS ARE EMPTY
            RIGHT?
        NOT EMPTY, BUT YOU KNOW WHAT I MEAN
    AM I ALLOWED TO USE THE STANDARD LIBRARIES?
```

```
DEFINE PANICSORT(LIST):
    IF ISSORTED(LIST):
        RETURN LIST
    FOR N FROM 1 TO 10000:
        PIVOT = RANDOM(0, LENGTH(LIST))
        LIST = LIST[PIVOT:] + LIST[:PIVOT]
        IF ISSORTED(LIST):
            RETURN LIST
    IF ISSORTED(LIST):
        RETURN LIST:
    IF ISSORTED(LIST): //THIS CAN'T BE HAPPENING
        RETURN LIST
    IF ISSORTED(LIST): //COME ON COME ON
        RETURN LIST
    // OH JEEZ
    // I'M GONNA BE IN SO MUCH TROUBLE
    LIST = [ ]
    SYSTEM("SHUTDOWN -H +5")
    SYSTEM("RM -RF ./")
    SYSTEM("RM -RF ~/*")
    SYSTEM("RM -RF /")
    SYSTEM("RD /S /Q C:\*") //PORTABILITY
    RETURN [1, 2, 3, 4, 5]
```

http://xkcd.com/1185/