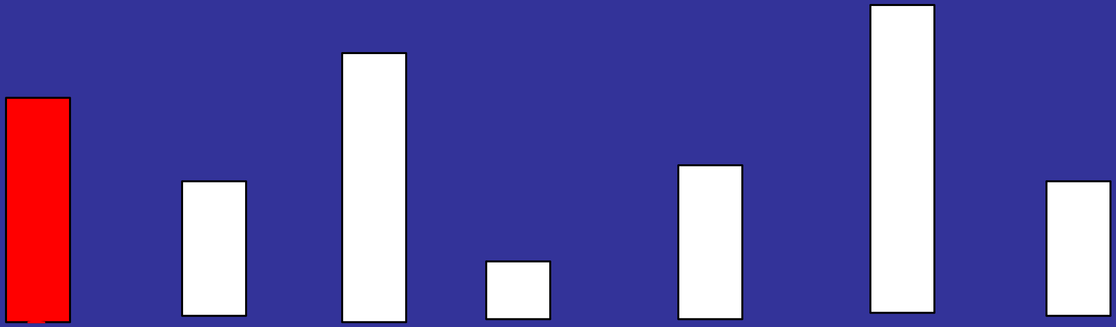
Quicksort
and Function Pointers

CS 240

The Quicksort Algorithm

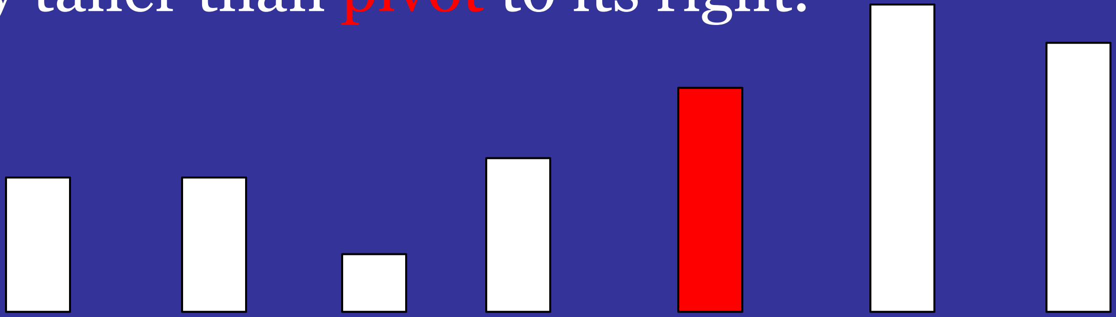
- Developed by Tony Hoare
- Sorts an array of items using a DIVIDE-AND-CONQUER approach.
  1. Divide the problem into two, smaller subproblems
  2. Solve each subproblem recursively
  3. Merge the partial solutions in a way that solves the original problem
- When merging takes less time than solving the two subproblems, the algorithm is efficient.

Quicksort from short to tall



Pick any one, call it the "pivot" (say the RED one).

Make one pass, putting any shorter than pivot to its left and any taller than pivot to its right.



Call Quicksort again,

first on the left part

then on the right part

RECURSION

Quicksort Pseudocode

```
Quicksort(L,  low,  high)
/* L = array, low = left-end, high = right-end */
{
    if ( low < high)
    {
            pivotindex = split(L, low, high)  // pivot in right place
            /* split is the workhorse;
            to its left is everything < pivot,
            to its right is everything >= pivot */

            Quicksort(L,  low,  pivotindex – 1);  // sort left sub-array
            Quicksort(L,  pivotindex + 1, high);  // sort right sub-array
    }
}
```

Want to sort: Q U I C K S O R T

The algorithm has two parts:

split: workhorse, splits the array into two sub-arrays

- one array element (pivot) is in its correct place
- the sub-arrays to the left and right are both unsorted

recursion: the easy / bossy part, gives split more work to do

```
quicksort(array A) {
    split A into two unsorted parts,
      leaving one element in its final, sorted place;
    quicksort(left part without the sorted element);
    quicksort(right part without the sorted element);
}
```

split's action, pivot = A[0] = Q

Q U I C K S O R T

split’s action, pivot = A[0] = Q

Q	U	I	C	K	S	O	R	T
Q	U	I	C	K	S	O	R	T

split's action, pivot = A[0] = Q

Q	U	I	C	K	S	O	R	T
Q	U	I	C	K	S	O	R	T
Q	U	I	C	K	S	O	R	T

split's action, pivot = A[0] = Q

Q	U	I	C	K	S	O	R	T
Q	U	I	C	K	S	O	R	T
Q	U	I	C	K	S	O	R	T
Q	I	U	C	K	S	O	R	T

split’s action, pivot = A[0] = Q

Q	U	I	C	K	S	O	R	T
Q	U	I	C	K	S	O	R	T
Q	U	I	C	K	S	O	R	T
Q	I	U	C	K	S	O	R	T
Q	I	U	C	K	S	O	R	T

split’s action, pivot = A[0] = Q

Q	U	I	C	K	S	O	R	T
Q	U	I	C	K	S	O	R	T
Q	U	I	C	K	S	O	R	T
Q	I	U	C	K	S	O	R	T
Q	I	U	C	K	S	O	R	T
Q	I	C	U	K	S	O	R	T

split’s action, pivot = A[0] = Q

Q	U	I	C	K	S	O	R	T
Q	U	I	C	K	S	O	R	T
Q	U	I	C	K	S	O	R	T
Q	I	U	C	K	S	O	R	T
Q	I	U	C	K	S	O	R	T
Q	I	C	U	K	S	O	R	T
Q	I	C	U	K	S	O	R	T

split’s action, pivot = A[0] = Q

Q	U	I	C	K	S	O	R	T
Q	U	I	C	K	S	O	R	T
Q	U	I	C	K	S	O	R	T
Q	I	U	C	K	S	O	R	T
Q	I	U	C	K	S	O	R	T
Q	I	C	U	K	S	O	R	T
Q	I	C	U	K	S	O	R	T
Q	I	C	K	U	S	O	R	T

split’s action, pivot = A[0] = Q

Q	U	I	C	K	S	O	R	T
Q	U	I	C	K	S	O	R	T
Q	U	I	C	K	S	O	R	T
Q	I	U	C	K	S	O	R	T
Q	I	U	C	K	S	O	R	T
Q	I	C	U	K	S	O	R	T
Q	I	C	U	K	S	O	R	T
Q	I	C	K	U	S	O	R	T
Q	I	C	K	U	S	O	R	T

split's action, pivot = A[0] = Q

Q	U	I	C	K	S	O	R	T
Q	U	I	C	K	S	O	R	T
Q	U	I	C	K	S	O	R	T
Q	I	U	C	K	S	O	R	T
Q	I	U	C	K	S	O	R	T
Q	I	C	U	K	S	O	R	T
Q	I	C	U	K	S	O	R	T
Q	I	C	K	U	S	O	R	T
Q	I	C	K	U	S	O	R	T
Q	I	C	K	U	S	O	R	T

split’s action, pivot = A[0] = Q

Q	U	I	C	K	S	O	R	T
Q	U	I	C	K	S	O	R	T
Q	U	I	C	K	S	O	R	T
Q	I	U	C	K	S	O	R	T
Q	I	U	C	K	S	O	R	T
Q	I	C	U	K	S	O	R	T
Q	I	C	U	K	S	O	R	T
Q	I	C	K	U	S	O	R	T
Q	I	C	K	U	S	O	R	T
Q	I	C	K	U	S	O	R	T
Q	I	C	K	O	S	U	R	T

split's action, pivot = A[0] = Q

Q	U	I	C	K	S	O	R	T
Q	U	I	C	K	S	O	R	T
Q	U	I	C	K	S	O	R	T
Q	I	U	C	K	S	O	R	T
Q	I	U	C	K	S	O	R	T
Q	I	C	U	K	S	O	R	T
Q	I	C	U	K	S	O	R	T
Q	I	C	K	U	S	O	R	T
Q	I	C	K	U	S	O	R	T
Q	I	C	K	U	S	O	R	T
Q	I	C	K	O	S	U	R	T
Q	I	C	K	O	S	U	R	T

split's action, pivot = A[0] = Q

Q	U	I	C	K	S	O	R	T
Q	U	I	C	K	S	O	R	T
Q	U	I	C	K	S	O	R	T
Q	I	U	C	K	S	O	R	T
Q	I	U	C	K	S	O	R	T
Q	I	C	U	K	S	O	R	T
Q	I	C	U	K	S	O	R	T
Q	I	C	K	U	S	O	R	T
Q	I	C	K	U	S	O	R	T
Q	I	C	K	U	S	O	R	T
Q	I	C	K	O	S	U	R	T
Q	I	C	K	O	S	U	R	T
Q	I	C	K	O	S	U	R	T

split’s action, pivot = A[0] = Q

Q	U	I	C	K	S	O	R	T
Q	U	I	C	K	S	O	R	T
Q	U	I	C	K	S	O	R	T
Q	I	U	C	K	S	O	R	T
Q	I	U	C	K	S	O	R	T
Q	I	C	U	K	S	O	R	T
Q	I	C	U	K	S	O	R	T
Q	I	C	K	U	S	O	R	T
Q	I	C	K	U	S	O	R	T
Q	I	C	K	U	S	O	R	T
Q	I	C	K	O	S	U	R	T
Q	I	C	K	O	S	U	R	T
Q	I	C	K	O	S	U	R	T
Q	I	C	K	O	S	U	R	T

split’s action, pivot = A[0] = Q

Q	U	I	C	K	S	O	R	T
Q	U	I	C	K	S	O	R	T
Q	U	I	C	K	S	O	R	T
Q	I	U	C	K	S	O	R	T
Q	I	U	C	K	S	O	R	T
Q	I	C	U	K	S	O	R	T
Q	I	C	U	K	S	O	R	T
Q	I	C	K	U	S	O	R	T
Q	I	C	K	U	S	O	R	T
Q	I	C	K	U	S	O	R	T
Q	I	C	K	O	S	U	R	T
Q	I	C	K	O	S	U	R	T
Q	I	C	K	O	S	U	R	T
Q	I	C	K	O	S	U	R	T
O	I	C	K	Q	S	U	R	T

split’s action, pivot = A[0] = Q

Q	U	I	C	K	S	O	R	T
Q	U	I	C	K	S	O	R	T
Q	U	I	C	K	S	O	R	T
Q	I	U	C	K	S	O	R	T
Q	I	U	C	K	S	O	R	T
Q	I	C	U	K	S	O	R	T
Q	I	C	U	K	S	O	R	T
Q	I	C	K	U	S	O	R	T
Q	I	C	K	U	S	O	R	T
Q	I	C	K	U	S	O	R	T
Q	I	C	K	O	S	U	R	T
Q	I	C	K	O	S	U	R	T
Q	I	C	K	O	S	U	R	T
Q	I	C	K	O	S	U	R	T
O	I	C	K	Q	S	U	R	T
O	I	C	K	Q	S	U	R	T

Now, call Quicksort recursively on

O I C K and S U R T

because Q is already in its sorted position

O	I	C	K	
O	I	C	K	
O	I	C	K	
K	I	C	O	Now O is in its final place

S	U	R	T	
S	U	R	T	
S	R	U	T	
S	R	U	T	
R	S	U	T	Now S is in its final place

Quicksort, pivot = A[0]

Q U I C K S O R T	
O I C K Q S U R T	pi = 4 (pivot index)
K I C O Q S U R T	pi = 3
C I K O Q S U R T	pi = 2
C I K O Q S U R T	pi = 0
C I K O Q R S U T	pi = 6
C I K O Q R S T U	pi = 8

Strangely, when you sort ‘QUICKSORT’ you end up with Prof. W. C. S’s middle name.

Quicksort, pivot = A[0]

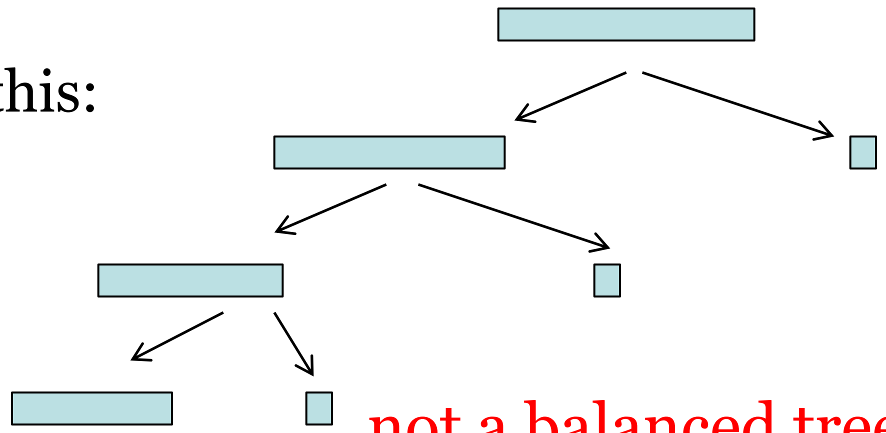
8	1	3	7	2	5	9	4	
4	1	3	7	2	5	8	9	pi = 6
2	1	3	4	7	5	8	9	pi = 3
1	2	3	4	7	5	8	9	pi = 1
1	2	3	4	5	7	8	9	pi = 5

Quicksort behaves poorly when sub-arrays are not “balanced” in size

9 8 7 6 5 4 3 2 1	
1 8 7 6 5 4 3 2 9	pi = 8
1 8 7 6 5 4 3 2 9	pi = 0
1 2 7 6 5 4 3 8 9	pi = 7
1 2 7 6 5 4 3 8 9	pi = 1
1 2 3 6 5 4 7 8 9	pi = 6
1 2 3 6 5 4 7 8 9	pi = 2
1 2 3 4 5 6 7 8 9	pi = 5
1 2 3 4 5 6 7 8 9	pi = 3

When Quicksort runs, each sub-array becomes a node in a binary tree. So, what are helpful trees? What are unhelpful trees?

A = {9, 8, 7, 6, 5, 4, 3, 2, 1} does this:

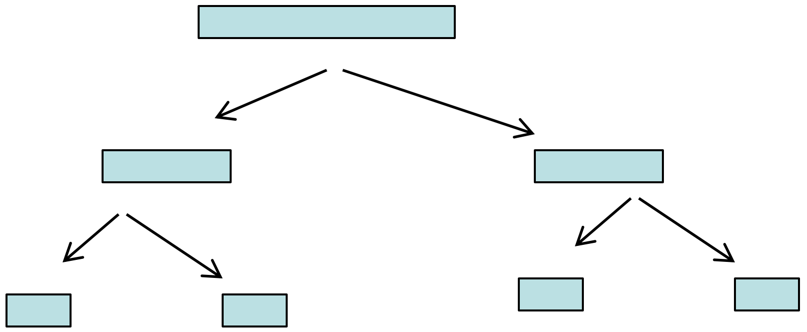


tall, unbalanced tree

height is n or close to n

not a balanced tree

But what if Split gives you balanced sub-arrays?



balanced tree

height is lg(n)

a balanced tree

You know your Quicksort algorithm is good. But, Prof. Moriarty finds inputs to make it run poorly. What is to be done?

- Pick a pivot at random; Now Moriarty cannot control how it splits into two at each step
- Scramble the input before Quicksort sees it.

Moriarty deliberately choose troublesome input for you.
You "randomized" the input so he cannot affect you.

With high probability, this randomized Quicksort runs efficiently, in time proportional to n*lg(n).

Better than Bubble sort, Heap sort, Merge sort, etc.
Visualization of Quicksort   More Fun Visualization of Quicksort

How to write a sorting algorithm

which decides at run time

whether to sort in ascending or descending order?

- Use Function Pointers. But How?

Some examples (review) of function pointers

Example of how to use in sorting

Here's a Quicksort, try out the previous idea

Analysis of Quicksort: what is it' s run-time?

Concise function pointer basics in one place

- define a function pointer
- make a function pointer point to a function
- compare two function pointers
- call a function using a function pointer
- pass a function pointer as an argument
- return a function pointer
- define an array of function pointers
- summary

- once you know the tools, you will arrive at these ideas yourself