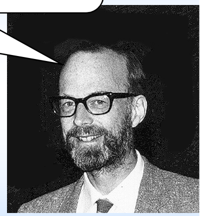
Data Structures and Algorithms

Sorting 3 quicksort

"We should forget about small efficiencies, say about 97% of the time. Premature optimization is the root of all evil"

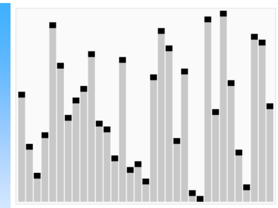


Tony Hoare (Sir Charles Anthony Hoare) invented quicksort in 1960. He was a British computer scientist who worked in Russia before returning to the UK to become a professor at Oxford. In addition to inventing quicksort, he developed an early programming language (Algo), Hoare logic and the concept of the monitor lock in concurrent programming.
Speaking at a conference in 2009, Hoare apologized for inventing the null reference. He called it his billion dollar mistake

Reading: Goodrich Chapter 11
Donald Knuth: The Art of Computer Programming, Volume 3: Sorting and Searching

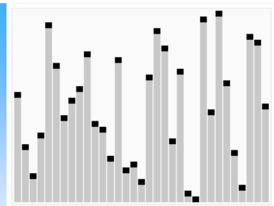
http://www.sorting-algorithms.com/quick-sort

quicksort



- quicksort is a **divide and conquer** algorithm
- It *partitions* the existing array by rearranging the elements
- It partitions by choosing a *pivot* value, then swapping elements so that all the elements to the left of the pivot are smaller and all those to the right are bigger

quicksort Overview



16	97	32	11	44	23	55

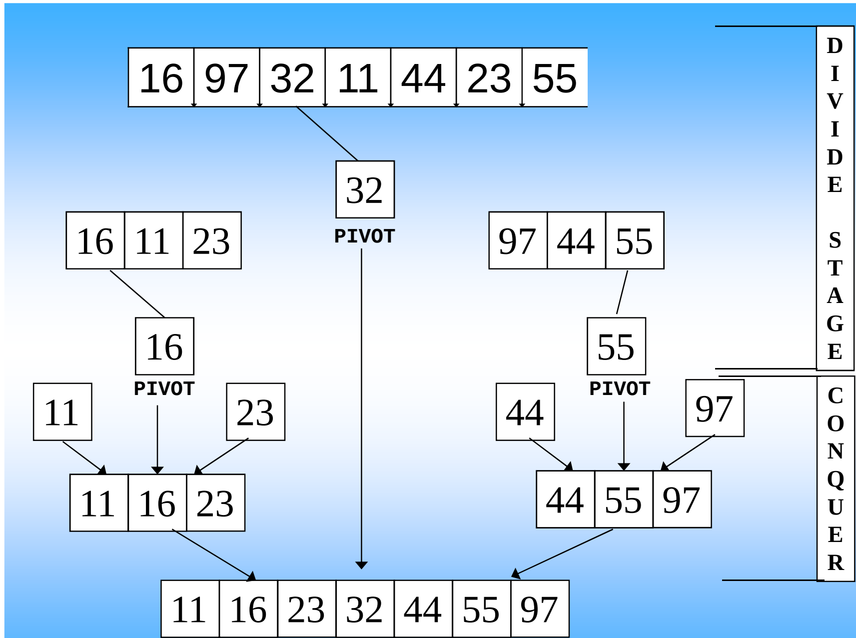
Pick Pivot 32 **How? More later**

Partition

16	11	23

97	44	55

QuickSort([16,11,23]) **QuickSort([97,44,55])**



quicksort informal algorithm

Base step

1. if the number of elements in T is size 0 or 1 then return

Recursive divide step

2. Pick an element *v* in T as the *pivot* element
3. Partition T without *v* (T-{*v*}) into two groups, the left group, L, consisting of elements smaller than or equal to *v* and the right group R consisting of elements greater than *v*.

$$L = \{x \in T - \{v\} \mid x \leq v\} \quad R = \{x \in T - \{v\} \mid x > v\}$$

4. L=quicksort(L) and R=quicksort (R)
5. Return result of L+pivot+R

Task: quicksort: Sorts Array T into ascending order

```
begin quicksort(T[low…high ]
    //1. base case, single instance is sorted
    if (low==high)
            return T[low]
    //1. base case 2, empty array, return null
    if (high>low)
            return null
    //2. Choose pivot
    pivot:=choosePivot()
    //3. Partition
    left:=partitionLeft(T,pivot)
    //4. Recursive calls
    right:= partitionRight(T,pivot)
    left:=quickSort(left)
    right:=quickSort(right)
    //5. Combine and return
    full:=left+pivot+right
    return full
```

Base Cases. *We need the empty case because a partition may be empty*

Divide Stage: *split into non overlapping sub problems (**partition**) and recursively solve*

Conquer. *Combine the solution of the recursive calls to get the combined solution*

quicksort Issues

The algorithm does not describe how to efficiently *implement* quicksort. Questions not clarified yet are

1. How to choose a pivot.
2. How do we actually perform a partition? Can we do it in place?
3. What do we do with duplicate pivot elements?

Choosing a Pivot

- The ideal pivot would split the array exactly in two (see analysis later)
- The middle value of a set of numbers is called the median
- However, finding the median takes time (it can be done in O(n), but has high overhead)
- The usual strategy is to select k elements randomly, then take the median of these
  - Median of three
  - Median of seven/nine
- A commonly used extension involves taking the median of medians

In Place Partitioning

We want to get from this

16	97	32	11	44	23	55

PIVOT

Note we don't care what order these elements are in

To this

16	11	23	32	97	44	55

PIVOT

without using any extra memory

In Place Partitioning

Swap pivot into position 1

32	97	16	11	44	23	55

LEFT RIGHT

Keep pointers to the left and right partition

In Place Partitioning

While T[left]<pivot, advance pointer (element already in the right partition)

97>32 So stop

32	97	16	11	44	23	55

LEFT RIGHT

While T[right]>pivot, decrease pointer (element already in the right partition)

55>32 So decrease

23<32 So stop

In Place Partitioning

32	23	16	11	44	**97**	55

LEFT RIGHT

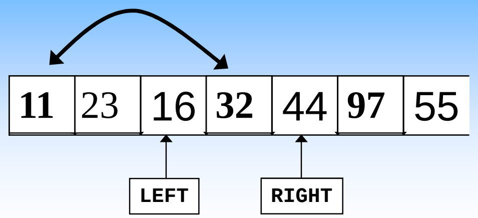
The element in T[left] needs to be in the right partition

The element in T[right] needs to be in the left partition

SO SWAP THEM

we can then advance both pointers and repeat

In Place Partitioning



Increase Left

Decrease Right

Since the pointers have crossed (left>right) we can stop

Last thing to do is swap the pivot into the middle (into position right)

In-Place Partitioning informal for array T[start]…T[end]

1. Swap the pivot out of the way and set left and right to start+1 and end
2. Repeat until left>right
   2.1 Advance the left pointer until the next element that should be in the right partition or end of array
   2.2. Decrease the right pointer until the next element that should be in the left partition
   2.3. Swap T[left] and T[right]
   2.4. Add one to left, subtract one from right
3. Swap pivot with the last element in left partition: T[right]

Partition(Array T, int start, int end, int pivotPos) return int

```
//1. Swap the pivot out of the way and set pointers
temp:=T[start]        T[start]:=T[pivotPos]     T[pivotPos]:=temp
left:=start+1, right:=end
//2. Repeat until left>right
while left<=right
    //2.1, 2.2 Scan through those already in the correct partition
    while T[left]<=T[start]
            left++
    while T[right]>T[start]
            right--
//2.3. Swap T[left] and T[right]
    if(left<right)
            temp:=T[left]     T[left]:=T[right]          T[right]:=temp
    left++, right--
//3. Restore pivot:
temp:=T[start]        T[start]:=T[right],         T[right]:=temp
//4. return pivot position
return right
```


Partition: More elegant version

partition(T, p, q, pivot)

exchange $T[p] \leftrightarrow T[pivot]$

$i \leftarrow p$

for $j \leftarrow p+1$ **to** q

  if $A[j] \leq A[p]$

    $i \leftarrow i+1$

    exchange $T[i] \leftrightarrow T[j]$

exchange $T[p] \leftrightarrow T[i]$

return i

http://math.hws.edu/TMCM/java/xSortLab/

quicksort formal algorithm

In place quicksort(Array[ ]T, integer start, end)

//1. Base Case

```
if start>=end
    return
```

//2. Divide: Select pivot r such that start≤r≤end

```
r:=choosePivot(T, start,end)
```

//3. Partition and return new pivot position

```
r:=partition(T,start,end,r)
```

//4. recursively quicksort left and right

```
quicksort(T,start,r-1)
quicksort(T,r+1,end)
```


quicksort Analysis

- Assume selecting the pivot is less work than partitioning
- Partitioning is an order *n* operation. Lets say it takes *cn* fundamental operations in all cases

1. Determine the fundamental operation

Comparison and return in partition: T[left]<=T[start],T[right]>T[start] and return

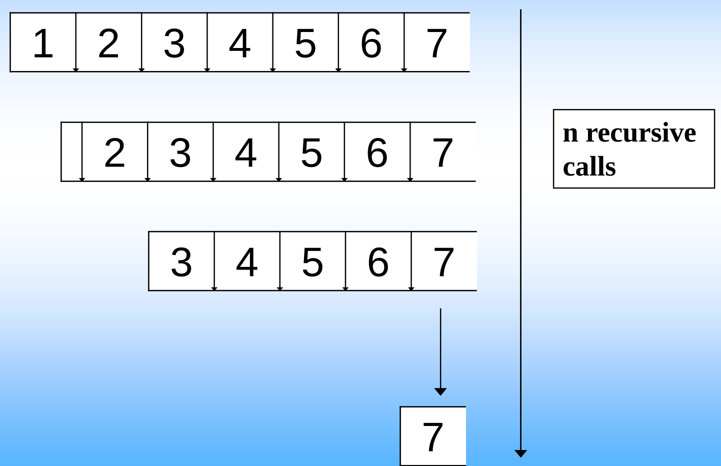
2. Decide on the case

worst case: pivot chosen each time is the largest (or smallest) element

best case: pivot chosen each time is the median element of the array

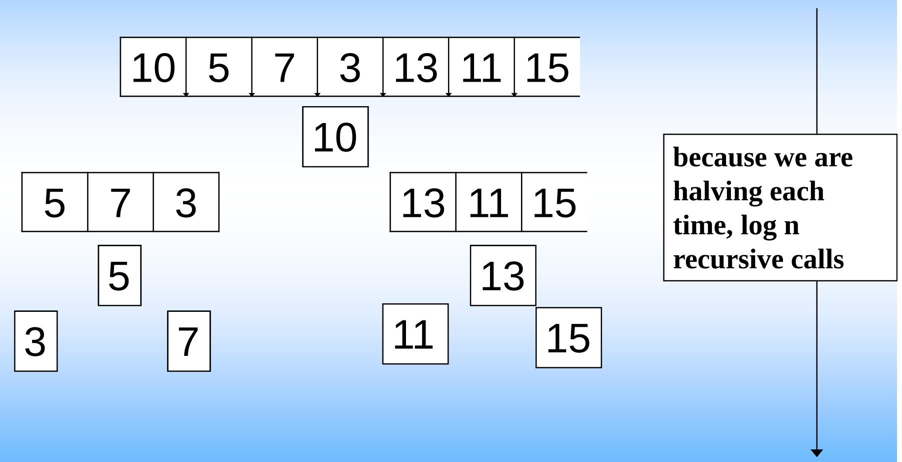
worst case: pivot chosen each time is the largest (or smallest) element

e.g. pick first element of the following array



best case: pivot chosen each time is the median element of the array

e.g. pick first element of the following array



Quick Sort 3: Worst Case Analysis

3. Form the run time complexity function t(n)

```
begin quickSort(Array[ ]T, integer start, end)
    if start>=end
        return
    int r:=choosePivot(T, start,end)
    r:=partition(T,start,end,r)
    QuickSort(T,start,r-1)
    QuickSort(T,r+1,end)
```

Base Case

$t(0) = 1$ $\quad$ $t(1) = 1$

Recursive Case

Left side is empty

right side has all the rest

$$t(n) = c \cdot n + t(0) + t(n-1)$$

partition an unsorted array of length n. Let c=1 for simplicity

Quick Sort 3: Worst Case Analysis

Eq. 1 $\quad t(n) = t(n-1) + n + 1$

Hence $\quad t(n-1) = t(n-2) + (n-1) + 1$

we can write $\quad t(n-1) = t(n-2) + n$

Sub into Eq. 1 $\quad$ Eq. 2 $\quad t(n) = t(n-2) + n + (n+1)$

Unwind again $\quad t(n-2) = t(n-3) + (n-2) + 1 = t(n-3) + (n-1)$

Sub into Eq 2. $\quad$ Eq. 3 $\quad t(n) = t(n-3) + (n-1) + n + (n+1)$

Repeat back substitution to get to t(1)

$$t(n) = 1 + 2 + \ldots (n-2) + (n-1) + n + (n+1)$$

Quick Sort 3: Worst Case Analysis

$$t(n) = 1 + 2 + \ldots (n-2) + (n-1) + n + (n+1)$$

$$t(n) = \sum_{i=1}^{n+1} i = \frac{(n+1)\cdot(n+2)}{2}$$

4. Characterise t(n)

$t(n)$ is $O(n^2)$

In the worst case, quick sort is quadratic (no better than bubble sort)!

Quick Sort 3: Best Case Analysis

3. Form the run time complexity function t(n)

```
begin quickSort(Array[ ]T, integer start, end)
    if start>=end
        return
    int r:=choosePivot(T, start,end)
    r:=partition(T,start,end,r)
    QuickSort(T,start,r-1)
    QuickSort(T,r+1,end)
```

Base Case

$t(0) = 1$ $\quad$ $t(1) = 1$

Recursive Case

Left side is half

right side has half

$$t(n) = c \cdot n + t\left(\frac{n}{2}\right) + t\left(\frac{n}{2}\right)$$

partition an unsorted array of length n

Quick Sort 3: Best Case Analysis

$$t(n) = 2 \cdot t\left(\frac{n}{2}\right) + c \cdot n$$

This now the same analysis as mergesort. We have two half size recursive calls plus the linear overhead of forming the partition.

$$t(n) = n \cdot \log(n) + c \cdot n$$

4. Characterise t(n)

$t(n)$ **is best case** $O(n \log(n))$

In the best case, quick sort is O(nlog(n)),

but then insertion sort is O(n) in the best case, so surely quicksort is rubbish?

Quick Sort 3: Average Case Analysis

3. Form the run time complexity function t(n)

Suppose that, at a call to quick sort, our pivot selection procedure means that each partition size is equally likely

- If the partition sizes are *(n-1)* and *0*

$$t(n) = t(n-1) + t(0) + n$$

Worst

- If the partition sizes are both n/2

$$t(n) = t\left(\frac{n}{2}\right) + t\left(\frac{n}{2}\right) + n$$

Best

- Generally, if the partition sizes are *(n-i)* and *i* then

$$t(n) = t(n-i) + t(i) + n$$

Quick Sort 3: Average Case Analysis

3. Form the run time complexity function t(n)

- If we average over all partition sizes, we get

$$t(n) = \frac{1}{n}\big(t(0) + t(1) + \cdots t(n-1) + t(n-1) + \cdots + t(0)\big) + n$$

Equation (1)

$$t(n) = \frac{2}{n}\big(t(0) + t(1) + \cdots t(n-1)\big) + n$$

We wish to solve this recurrence to get the form

$$t(n) = f(n)$$

to do this, one approach is to first rewrite (1) in just terms of *t(n)* and *t(n-1)*, then use a similar expansion technique as used for the worst case

Quick Sort 3: Average Case Analysis

3. Form the run time complexity function t(n)

(2) $nt(n) = 2\big(t(0) + t(1) + \cdots t(n-1)\big) + n^2$

Write the equivalent expression for *n-1*

(3) $(n-1)t(n-1) = 2\big(t(0) + t(1) + \cdots t(n-2)\big) + (n-1)^2$

Subtract (3) from (2) $nt(n) - (n-1)t(n-1) = 2\big(t(n-1)\big) + n^2 - (n-1)^2$

Rearrange and simplify $nt(n) = 2\big(t(n-1)\big) + (n-1)t(n-1) + 2n - 1$

$$nt(n) = (n-1)t(n-1) + 2n - 1$$

Ignoring the constant gives (4)

$$nt(n) = (n+1)t(n-1) + 2n$$

Quick Sort 3: Average Case Analysis

$$nt(n) = (n+1)t(n-1) + 2n$$

To solve this we need to rearrange by dividing through by $n(n+1)$

(5) $$\frac{t(n)}{n+1} = \frac{t(n-1)}{n} + \frac{2}{n+1}$$

Now, since,

$$\frac{t(n-1)}{n} = \frac{t(n-2)}{n-1} + \frac{2}{n}$$

$$\frac{t(n-2)}{n-1} = \frac{t(n-3)}{n-2} + \frac{2}{n-1}$$

$$\frac{t(n-3)}{n-2} = \frac{t(n-4)}{n-3} + \frac{2}{n-2} \quad \ldots\ldots \quad \frac{t(3)}{4} = \frac{t(2)}{3} + \frac{2}{4} \quad \frac{t(2)}{3} = \frac{t(1)}{2} + \frac{2}{3}$$

Quick Sort 3: Average Case Analysis

$$\frac{t(n)}{n+1} = \frac{t(n-1)}{n} + \frac{2}{n+1}$$

Repeatedly substituting into (5) $$\frac{t(n)}{n+1} = \frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1} + \cdots + \frac{2}{4} + \frac{2}{3} + \frac{t(1)}{2}$$

$$\frac{t(n)}{n+1} = 2\left(1 + \frac{1}{2} + \frac{1}{3} + \cdots \frac{1}{n+1}\right) - \frac{5}{2}$$

$$= a\sum_{i=1}^{n+1}\frac{1}{i} + b$$

Quick Sort 3: Average Case Analysis

since $\sum \frac{1}{x}$ is equivalent to $\int \frac{1}{x}dx$ as n tends to infinity

and $\int \frac{1}{x}dx = \ln(x)$

Hence $\sum_{i=1}^{n}\frac{1}{i}$ is bounded above and below by c.log(n).

Hence $\frac{t(n)}{n+1} = O(\log(n))$

and $t(n) = O(n\log(n))$

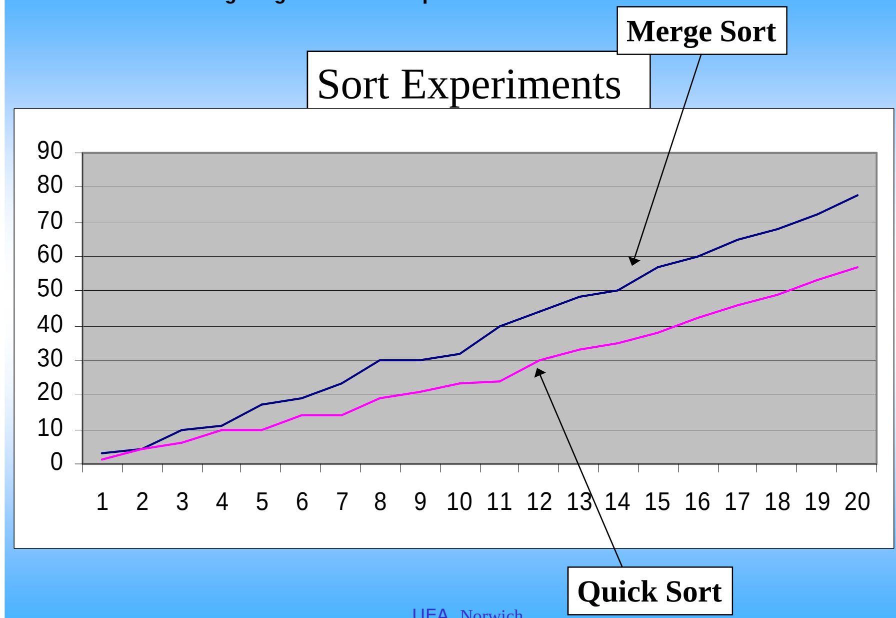
The average case complexity of quick sort is *O(n log(n))*

this proof is not examinable

Quick Sort summary

- Quick Sort is an **average case** Θ(*n.log(n)*) sorting algorithm based on comparison.
- Quick Sort is **worst case** Θ(*n^2*) with standard pivoting methods
- Quick Sort can be adapted to be **worst case** Θ(*n.log(n)*) by using the O(n) Quick Select method to choose the pivot.
- In practice this slows it down on most cases!

http://cs.stackexchange.com/questions/3/why-is-quicksort-better-than-other-sorting-algorithms-in-practice



Quicksort Optimizations

- Pivot selection: median of medians
- Use two pivots and split into three recursive calls (Dual Pivot Quick sort)
- Move duplicates of the pivot next to each other and don't recursively call them
- Perform the recursive call on the smallest segment first
- Use insertion sort for small segment sizes

After Sorting Lecture 3 you should be able to …

1. **Describe quicksort both informally and in formal pseudo code**
2. **Know the pivot selection methods and describe the partition operation in pseudo code.**
3. **quicksort an example array**
4. **Analyse the worst case time complexity**
5. **Know what the average case complexity is**