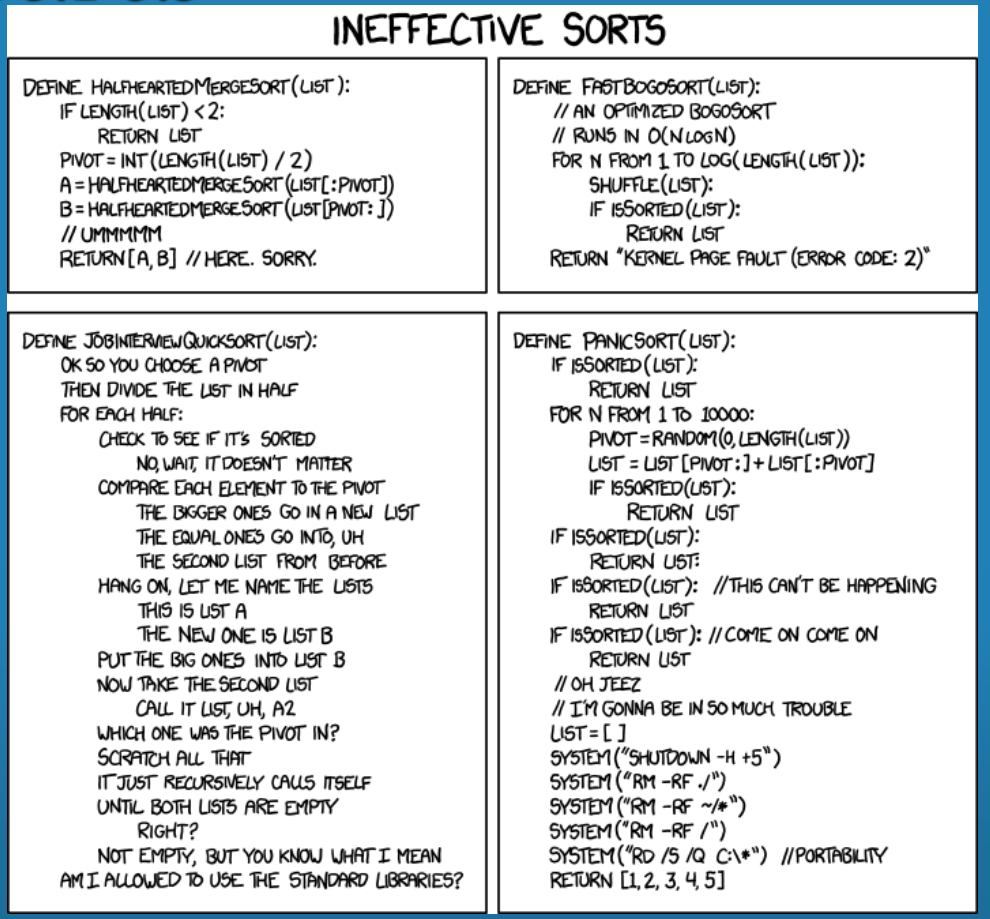
CS 395 – Analysis of Algorithms

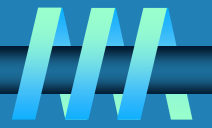
Chapter 5 – Divide-and-Conquer

Read 5.1-5.5

- Basic Idea
- Master Theorem
- Mergesort
- Quicksort – w10

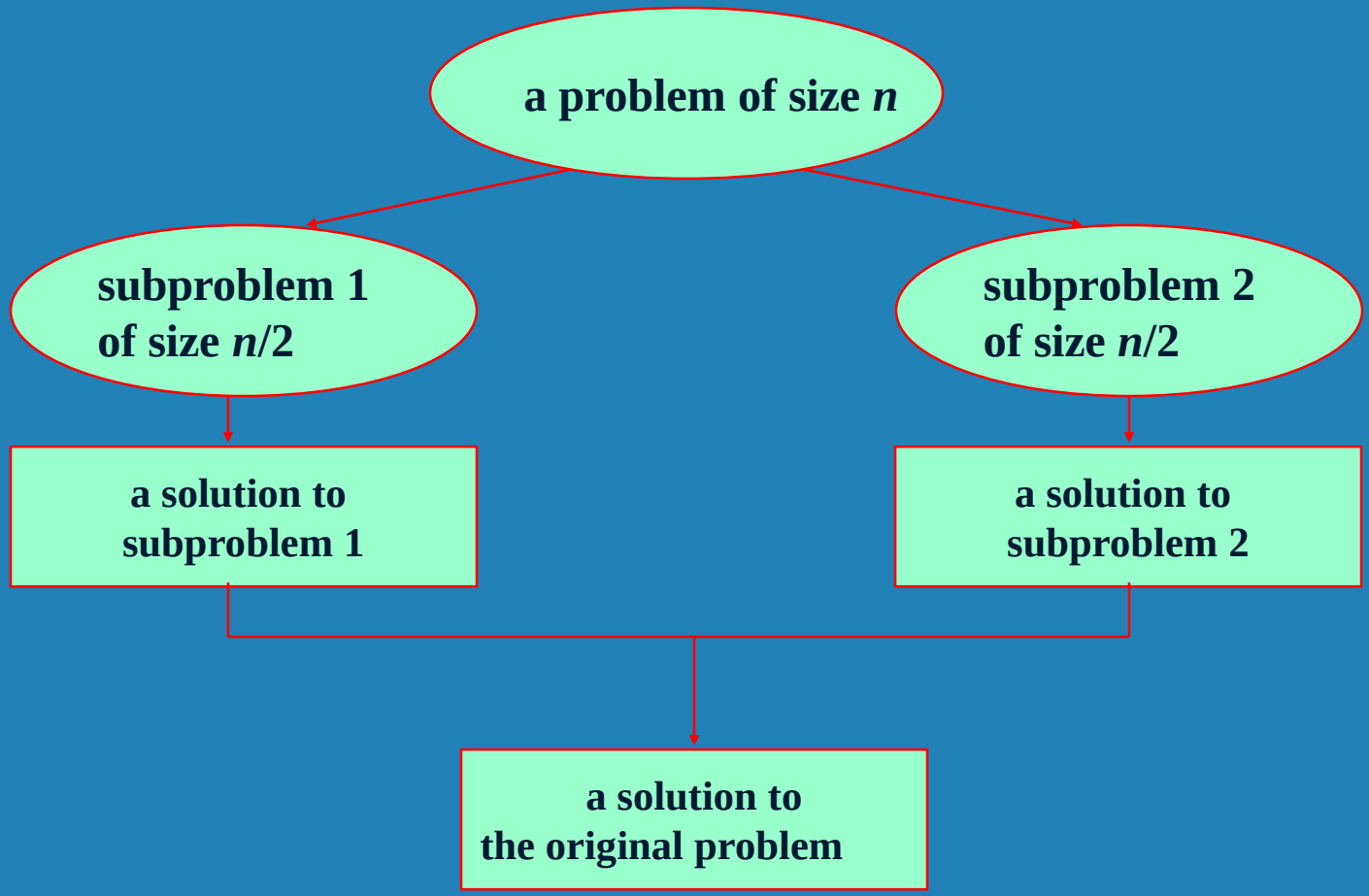


Divide-and-Conquer



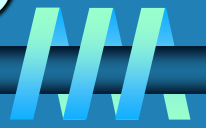
- The most-well known algorithm design strategy:
- Divide instance of problem into two or more similar, smaller instances
- Solve smaller instances recursively
- Obtain solution to original (larger) instance by combining these solutions

Divide-and-Conquer Technique (cont.)

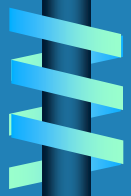


Dr. BC Note: According to this definition, Merge Sort and Quick Sort comes under divide and conquer (because there are 2 sub-problems) and Binary Search comes under decrease and conquer (because there is one sub-problem).

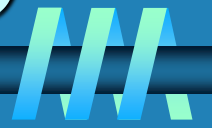
Divide-and-Conquer Technique (cont.)



- In each subdivision step, the smaller instances should have approx. the same size!
  - This might not happen, for some particular instances
- All smaller problem instances have to be solved !!
  - Usually two new smaller instances, at each step
- When do we stop the subdivision process?
  - Base cases? Just one or more?
  - Smaller instances might be solved by another algorithm

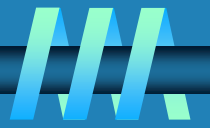


Divide-and-Conquer Technique (cont.)

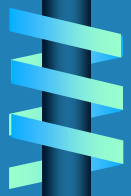


- This recursive strategy can be implemented
  - Using recursive functions / procedures (obvious solution)
  - Iteratively, using a stack, queue, etc.
    - Choose which sub-problem to solve next
- Problems?
  - Recursion is slow!
    - Identify all possible base cases
    - Solve small instances using other algorithms
  - Not the best approach for simple problems
    - e.g., adding N numbers
  - Sub-problems might overlap!
    - Reuse previous results / solutions

Divide-and-Conquer Examples



- Sorting: Mergesort and Quicksort
- Binary tree traversals
- Multiplication of large integers
- Matrix multiplication: Strassen's algorithm
- Closest-pair and convex-hull algorithms



General Divide-and-Conquer Recurrence

- Assume, $n = b^k$, where $k \geq 1$ and b is typically 2

- Let T(n) be an eventually non-decreasing function that satisfies the recurrence for the running time

$$T(n) = aT(n/b) + f(n)$$

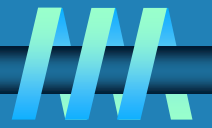
An instance of size n can be divided into b instances of size n/b, with a of them needing to be solved.

a : number of smaller instances ($a \geq 2$) needing to be solved

b : size factor (integer, $b \geq 2$)

f(n) : number of ops (or time) for defining smaller instances and/or combining their results

Master Theorem



If $f(n) \in \Theta(n^d)$, $d \geq 0$

If $a < b^d$, $T(n) \in \Theta(n^d)$

If $a = b^d$, $T(n) \in \Theta(n^d \log n)$

If $a > b^d$, $T(n) \in \Theta(n^{\log_b a})$

Can we simplify this?

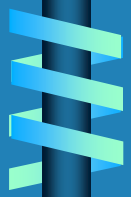
Note: The same results hold with O and Ω too.

a=4 b=2 n^1 means d=1

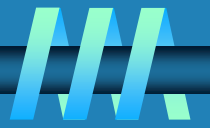
Examples: **$T(n) = 4T(n/2) + n \Rightarrow T(n) \in$** $4>2^1$ $T(n) \in \Theta(n^{\log_2 4})$

$T(n) = 4T(n/2) + n^2 \Rightarrow T(n) \in$ $4=2^2$ $T(n) \in \Theta(n^2 \log n)$

$T(n) = 4T(n/2) + n^3 \Rightarrow T(n) \in$ $4<2^3$ $T(n) \in \Theta(n^3)$



Master Theorem



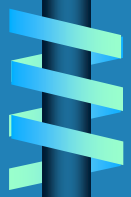
If $f(n) \in \Theta(n^d)$, $d \geq 0$

If $a < b^d$, $T(n) \in \Theta(n^d)$

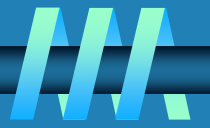
If $a = b^d$, $T(n) \in \Theta(n^d \log n)$

If $a > b^d$, $T(n) \in \Theta(n^{\log_b a})$

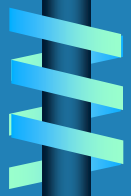
- You cannot use the Master Theorem if
  - T(n) is not monotone, ex: T(n) = sin n
  - f(n) is not a polynomial, ex: $T(n) = 2T(n/2) + 2^n$
  - b cannot be expressed as a constant, ex: T(n) = T(sqrt(n))



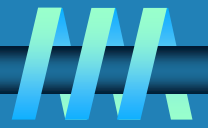
Mergesort



- Split array A[0..n-1] in two about equal halves and make copies of each half in arrays B and C
- Sort arrays B and C recursively
- Merge sorted arrays B and C into array A as follows:
  - Repeat the following until no elements remain in one of the arrays:
    - compare the first elements in the remaining unprocessed portions of the arrays
    - copy the smaller of the two into A, while incrementing the index indicating the unprocessed portion of that array
  - Once all elements in one of the arrays are processed, copy the remaining unprocessed elements from the other array into A.

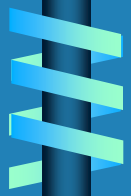


Pseudocode of Mergesort

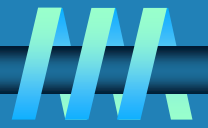


ALGORITHM *Mergesort*$(A[0..n-1])$

//Sorts array $A[0..n-1]$ by recursive mergesort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
if $n > 1$
    copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
    copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil n/2 \rceil - 1]$
    Mergesort$(B[0..\lfloor n/2 \rfloor - 1])$
    Mergesort$(C[0..\lceil n/2 \rceil - 1])$
    Merge(B, C, A)

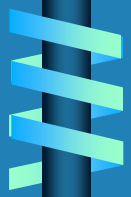


Pseudocode of Merge



ALGORITHM *Merge*$(B[0..p-1], C[0..q-1], A[0..p+q-1])$

//Merges two sorted arrays into one sorted array
//Input: Arrays $B[0..p-1]$ and $C[0..q-1]$ both sorted
//Output: Sorted array $A[0..p+q-1]$ of the elements of B and C
$i \leftarrow 0;\ j \leftarrow 0;\ k \leftarrow 0$
while $i < p$ **and** $j < q$ **do**
    if $B[i] \leq C[j]$
        $A[k] \leftarrow B[i];\ i \leftarrow i + 1$
    else $A[k] \leftarrow C[j];\ j \leftarrow j + 1$
    $k \leftarrow k + 1$
if $i = p$
    copy $C[j..q-1]$ to $A[k..p+q-1]$
else copy $B[i..p-1]$ to $A[k..p+q-1]$

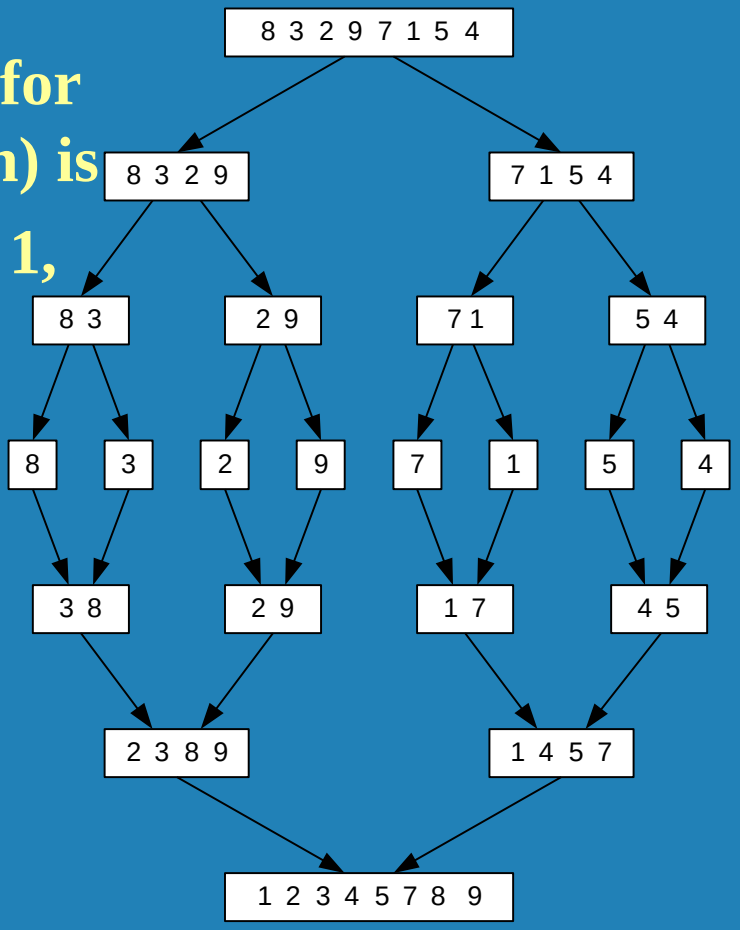


Mergesort Example

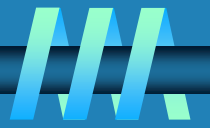
Assuming for simplicity that n is a power of 2, the recurrence relation for the number of key comparisons C(n) is

$$C(n) = 2C(n/2) + C_{merge}(n) \quad \text{for } n > 1,$$

$$C(1) = 0.$$

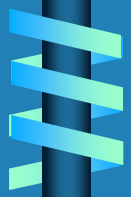


Analysis of MergeSort



- Using Master Theorem:
- -- splitting array into two halves/approx. equal-sized subarrays at each recursive step (hence, a = 2 and b = 2)
- -- the overhead f(n) is the cost of Merge subroutine
- -- merging two sorted sub-arrays into one larger array (whose size equals the sum of the two subarray sizes) is O(n)

$$C_{worst}(n) = 2C_{worst}(n/2) + n - 1$$



Analysis of MergeSort

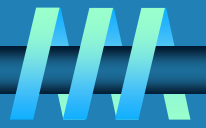
$$C_{worst}(n) = 2C_{worst}(n/2) + n - 1$$

- **Hence, by Master Theorem: a = 2, b = 2, d = 1**
- **- where d is the power in f(n) = O(n^d)**

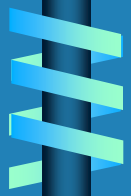
Master Theorem:
If $f(n) \in \Theta(n^d)$, $d \geq 0$
If $a < b^d$, $T(n) \in \Theta(n^d)$
If $a = b^d$, $T(n) \in \Theta(n^d \log n)$
If $a > b^d$, $T(n) \in \Theta(n^{\log_b a})$

- **- it's the middle case, therefore T(n) = Θ(n log n)**

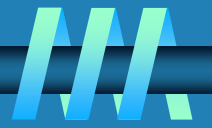
Analysis of Mergesort



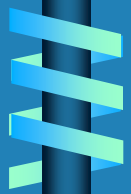
- All cases have same efficiency: $\Theta(n \log n)$
- Number of comparisons in the worst case is close to theoretical minimum for comparison-based sorting:
  - $\lceil \log_2 n! \rceil \approx n \log_2 n - 1.44n$
  - NOTE: above is based on Information Theory, not the Master Theorem
  - For large n, we can safely ignore the -1.44n term
- Space requirement: $\Theta(n)$ (not in-place)
- Stable algorithm



Variations of Mergesort (bonus topic)

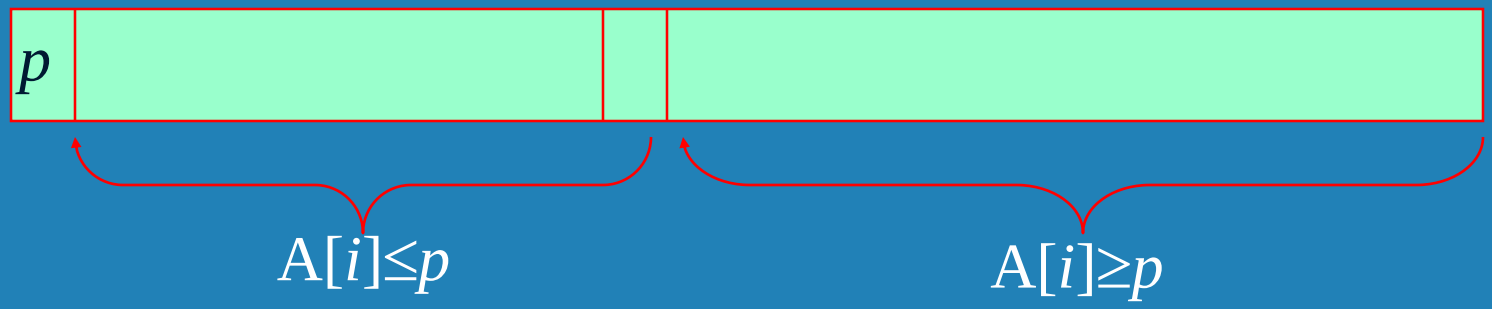


- Can be implemented without recursion (bottom-up)
  - Steps:
    - Merge pairs of array's elements
    - Merge the sorted pairs
    - Continue this way
  - Avoids time and space overhead of using stack to handle recursive calls
- Multiway Mergesort
  - Steps:
    - Divide in more than 2 parts
    - Sort each recursively
    - Merge them together
  - Particularly useful for disk-based sorting of files



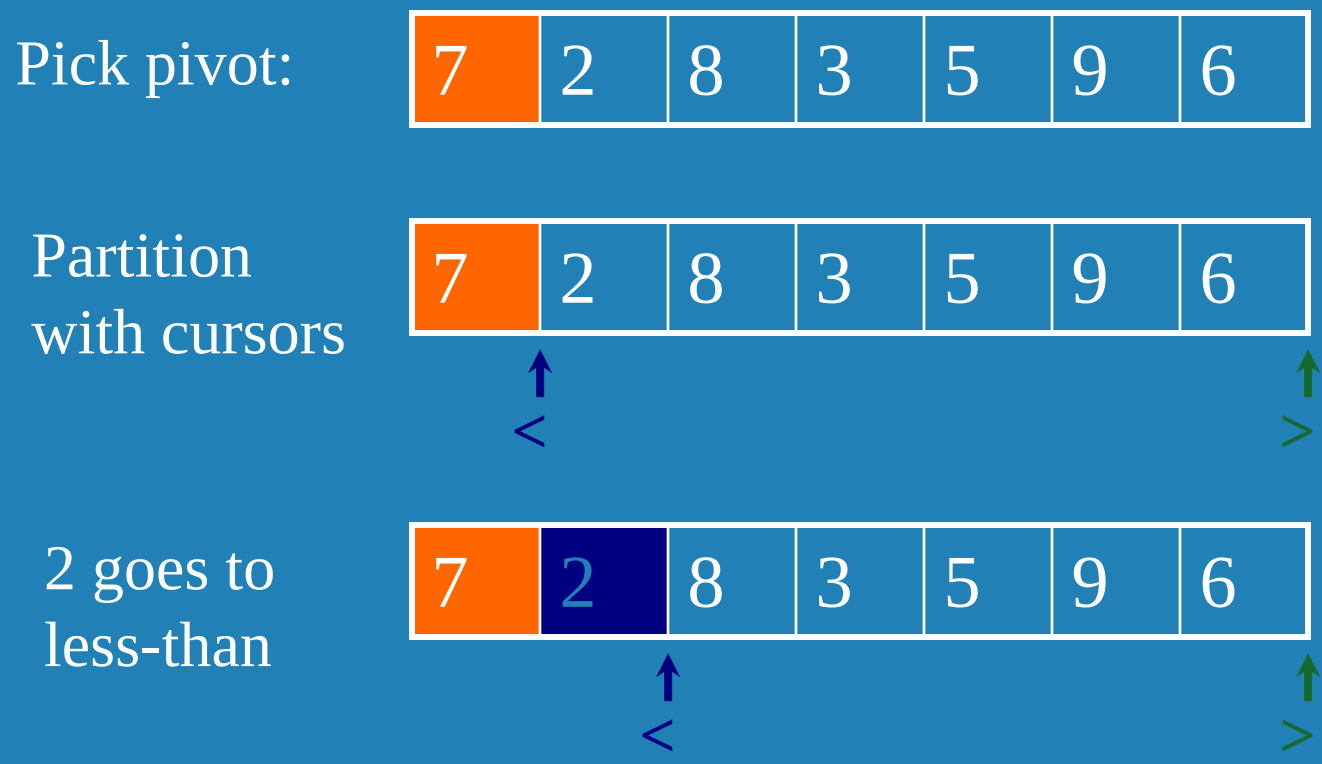
Quicksort

- Select a pivot (partitioning element) – here, the first element
- Rearrange the list so that all the elements in the first s positions are smaller than or equal to the pivot and all the elements in the remaining n-s positions are larger than or equal to the pivot



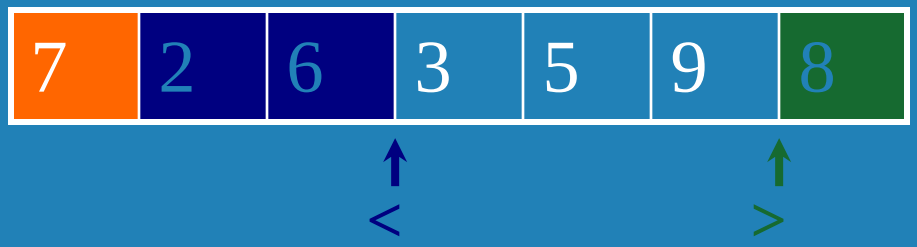
- Exchange the pivot with the last element in the first (i.e., ≤) subarray — the pivot is now in its final position
- Sort the two subarrays recursively

QuickSort Partition



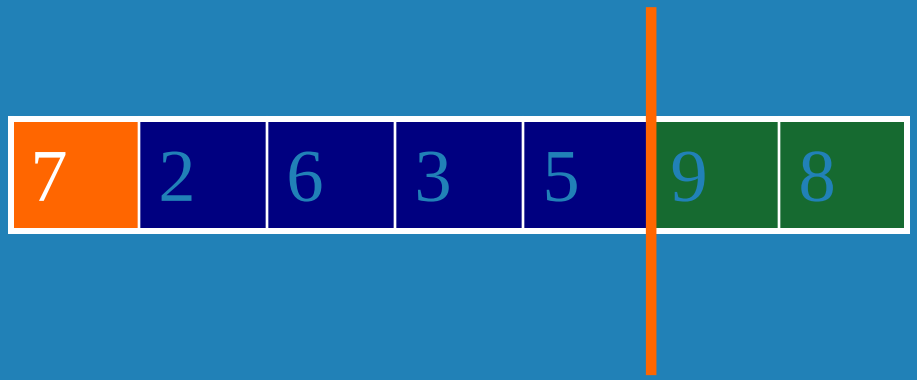
QuickSort Partition (cont'd)

6, 8 swap
less/greater-than

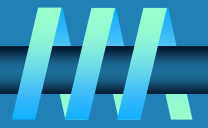


3,5 less-than
9 greater-than

Partition done.
Recursively
sort each side.

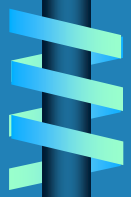


Quicksort Algorithm



ALGORITHM *Quicksort*$(A[l..r])$

//Sorts a subarray by quicksort
//Input: Subarray of array $A[0..n-1]$, defined by its left and right
// indices l and r
//Output: Subarray $A[l..r]$ sorted in nondecreasing order
if $l < r$
  $s \leftarrow$ *Partition*$(A[l..r])$ //s is a split position
  Quicksort$(A[l..s-1])$
  Quicksort$(A[s+1..r])$



Hoare's Partitioning Algorithm

ALGORITHM *HoarePartition*$(A[l..r])$

//Partitions a subarray by Hoare's algorithm, using the first element
// as a pivot
//Input: Subarray of array $A[0..n-1]$, defined by its left and right
// indices l and r $(l < r)$
//Output: Partition of $A[l..r]$, with the split position returned as
// this function's value

$p \leftarrow A[l]$
$i \leftarrow l;\ j \leftarrow r+1$
repeat
  repeat $i \leftarrow i+1$ **until** $A[i] \geq p$
  repeat $j \leftarrow j-1$ **until** $A[j] \leq p$
  swap$(A[i], A[j])$
until $i \geq j$
swap$(A[i], A[j])$ //undo last swap when $i \geq j$
swap$(A[l], A[j])$
return j

quicksort

koicqsurt

ickoqsurt

cikoqsurt

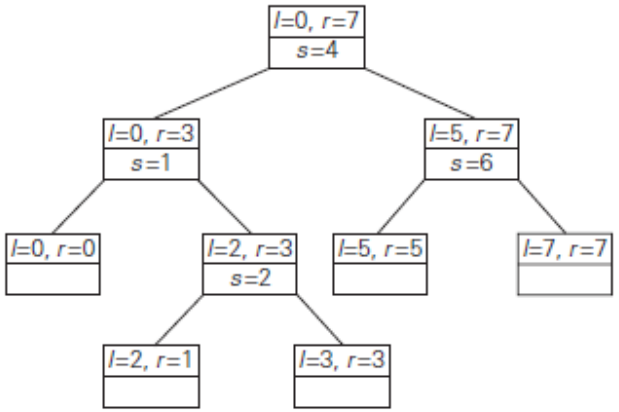
cikoqrsut

cikoqrstu

Wait. What happened to sorting this part (between the k and the q)?

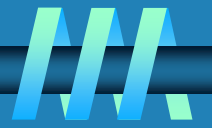
Quicksort Example

0	1	2	3	4	5	6	7
	i						j
5	3	1	9	8	2	4	7
			i			j	
5	3	1	9	8	2	4	7
			i			j	
5	3	1	4	8	2	9	7
				i	j		
5	3	1	4	8	2	9	7
				i	j		
5	3	1	4	2	8	9	7
				j	i		
5	3	1	4	2	8	9	7
2	3	1	4	**5**	8	9	7
	i		j				
2	3	1	4				
	i	j					
2	3	1	4				
	i	j					
2	1	3	4				
	j	i					
2	1	3	4				
1	**2**	3	4				
1							
			ij				
		3	4				
		j	i				
		3	4				
			4				
						i	j
					8	9	7
						i	j
					8	7	9
						j	i
					8	7	9
					7	**8**	9
					7		
							9



(b)

Analysis of Quicksort



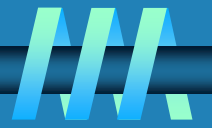
- **Best case: split in the middle — $\Theta(n \log n)$**
  - $C_{best}(n) = 2C_{best}(n/2) + n$ for $n > 1$,
    $C_{best}(1) = 0$.
  - **Master Theorem**
    - **a=2, b=2, d=1**
    - **a ? b^d**
    - **$2 = 2^1$**
    - **$\Theta(n \log n)$**

Master Theorem:
If $f(n) \in \Theta(n^d)$, $d \geq 0$
If $a < b^d$, $T(n) \in \Theta(n^d)$
If $a = b^d$, $T(n) \in \Theta(n^d \log n)$
If $a > b^d$, $T(n) \in \Theta(n^{\log_b a})$

- **Worst case: sorted array! — $\Theta(n^2)$**

$$C_{worst}(n) = (n+1) + n + \cdots + 3 = \frac{(n+1)(n+2)}{2} - 3 \in \Theta(n^2).$$

Analysis of Quicksort

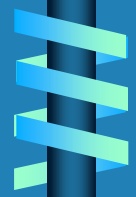


- Average case: random arrays — Θ(n log n)

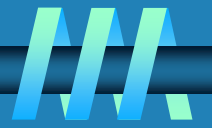
$$C_{avg}(n) = \frac{1}{n}\sum_{s=0}^{n-1}[(n+1) + C_{avg}(s) + C_{avg}(n-1-s)] \quad \text{for } n > 1,$$

$$C_{avg}(0) = 0, \qquad C_{avg}(1) = 0.$$

- Its solution, which is much trickier than the worst- and best-case analyses, turns out to be
  - $C_{avg}(n) \approx 2n \ln n \approx 1.39n \log_2 n$.
- Thus, on the average, quicksort makes only 39% more comparisons than in the best case.
  - Moreover, its inner most loop is so efficient that it usually runs faster than mergesort on randomly ordered arrays of nontrivial sizes.



Analysis of Quicksort



- **Improvements:**
  - **better pivot selection: median-of-three partitioning**
  - **switch to insertion sort on small subfiles/subarrays**
  - **elimination of recursion**
  - **Three-way partition**
  - **These combine to 20-25% improvement**

- **Considered the method of choice for internal sorting of large files ($n \geq 10000$)**