Quicksort

Introduction

- Fastest known sorting algorithm in practice
- Average case: O(N log N) (we don't prove it)
- Worst case: $O(N^2)$
  - But, the worst case seldom happens.
- Another divide-and-conquer recursive algorithm, like mergesort

Quicksort

- Divide step:
  - Pick any element (***pivot***) v in S
  - Partition S – {v} into two disjoint groups

    S1 = {x ∈ S – {v} | x <= v}

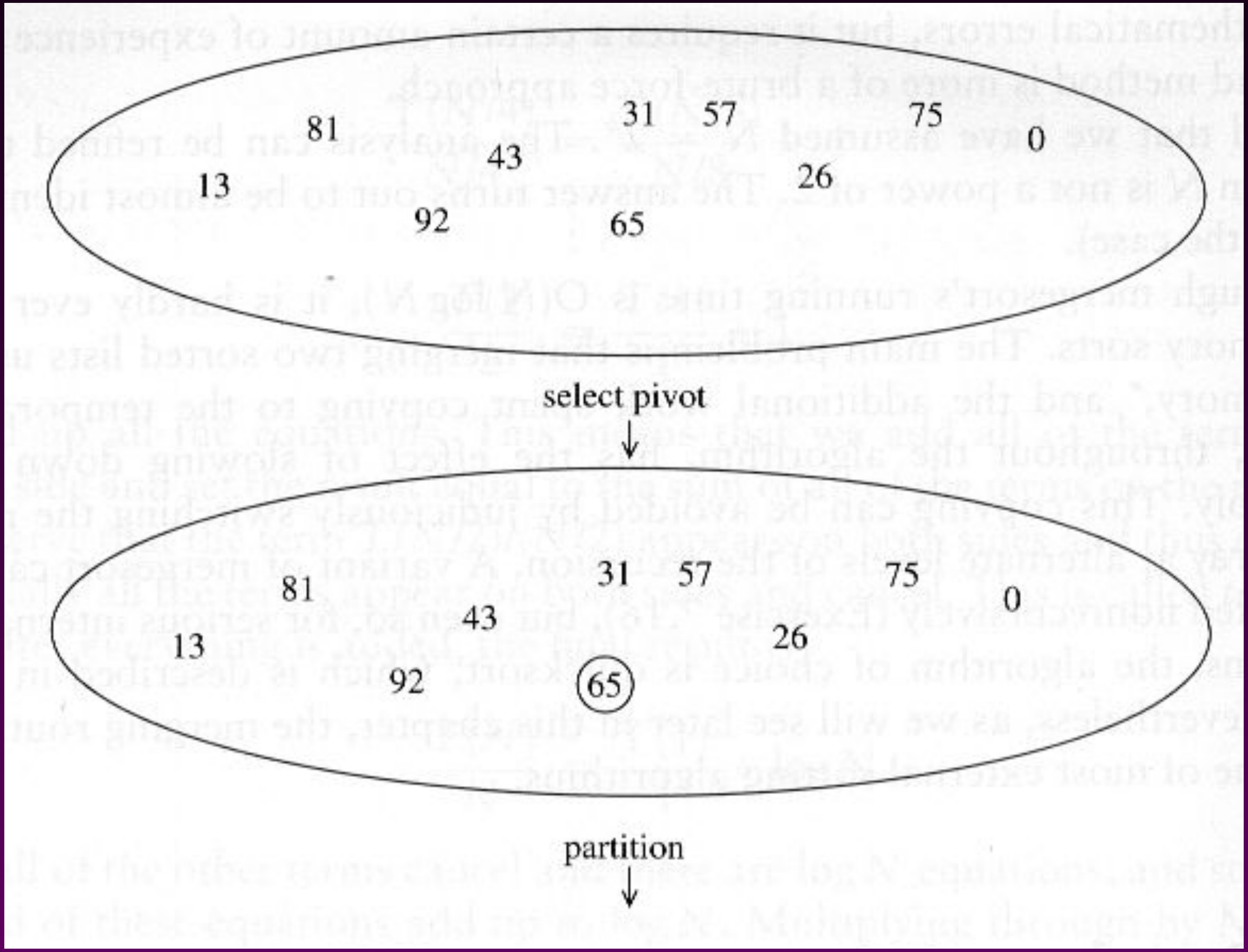
    S2 = {x ∈ S – {v} | x **≥** v}

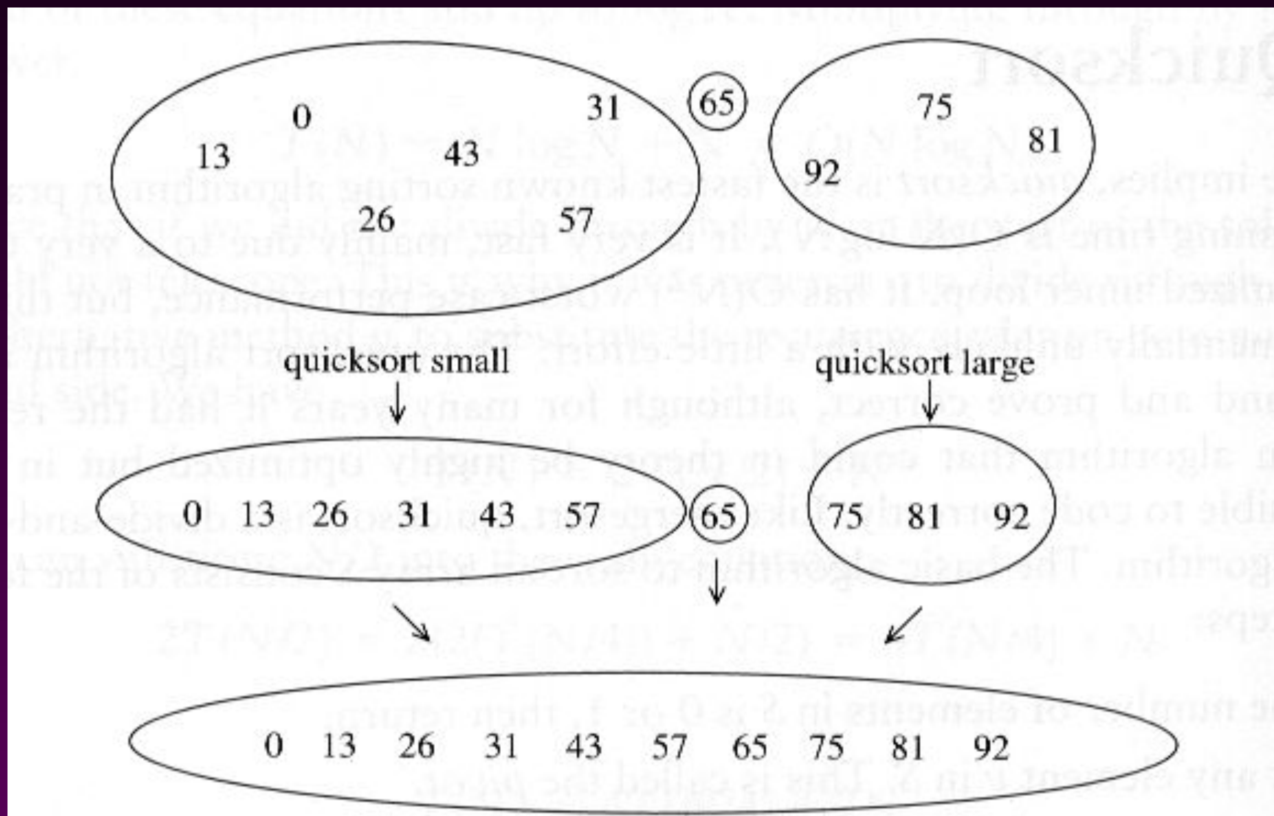
- Conquer step: recursively sort S1 and S2

- Combine step: the sorted S1 (by the time returned from recursion), followed by v, followed by the sorted S2 (i.e., nothing extra needs to be done)

To simplify, we may assume that we don't have repetitive elements,
So to ignore the 'equality' case!

Example


81
31 57
75
0
43
13
26
92
65
select pivot
81
31 57
75
0
43
13
26
92
65
partition


13 0 43 31 26 57
65
75 81 92
quicksort small
quicksort large
0 13 26 31 43 57
65
75 81 92
0 13 26 31 43 57 65 75 81 92

Pseudo-code

```
Input: an array a[left, right]

QuickSort (a, left, right) {
     if (left < right) {
        pivot = Partition (a, left, right)
        Quicksort (a, left, pivot-1)
        Quicksort (a, pivot+1, right)
     }
}
```

Compare with MergeSort:

```
MergeSort (a, left, right) {
      if (left < right) {
          mid = divide (a, left, right)
          MergeSort (a, left, mid-1)
          MergeSort (a, mid+1, right)
          merge(a, left, mid+1, right)
      }
}
```

Two key steps

- How to pick a pivot?
- How to partition?

Pick a pivot

- Use the first element as pivot
  - if the input is random, ok
  - if the input is presorted (or in reverse order)
    - all the elements go into S2 (or S1)
    - this happens consistently throughout the recursive calls
    - Results in $O(n^2)$ behavior (Analyze this case later)
- Choose the pivot randomly
  - generally safe
  - random number generation can be expensive

In-place Partition

- If use additional array (not in-place) like MergeSort
  - Straightforward to code like MergeSort (write it down!)
  - Inefficient!

↓

- Many ways to implement
- Even the slightest deviations may cause surprisingly bad results.
  - Not stable as it does not preserve the ordering of the identical keys.
  - Hard to write correctly ☹

An easy version of in-place partition to understand, but not the original form

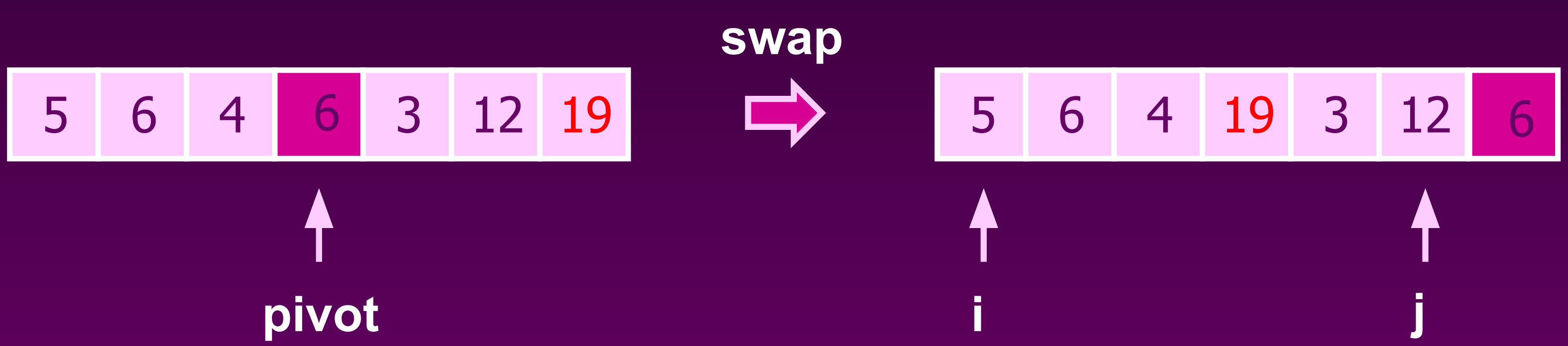
```
int partition(a, left, right, pivotIndex) {
    pivotValue = a[pivotIndex];
    swap(a[pivotIndex], a[right]); // Move pivot to end

    // move all smaller (than pivotValue) to the begining
    storeIndex = left;
    for (i from left to right) {
        if a[i] < pivotValue
            swap(a[storeIndex], a[i]);
            storeIndex = storeIndex + 1 ;
    }

    swap(a[right], a[storeIndex]); // Move pivot to its final place
    return storeIndex;
}
```

Look at Wikipedia

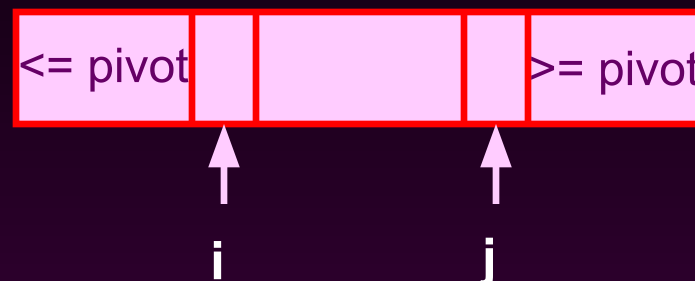
```
quicksort(a,left,right) {

    if (right>left) {
    pivotIndex = left;
    select a pivot value a[pivotIndex];

    pivotNewIndex=partition(a,left,right,pivotIndex);

    quicksort(a,left,pivotNewIndex-1);
    quicksort(a,pivotNewIndex+1,right);
    }
}
```

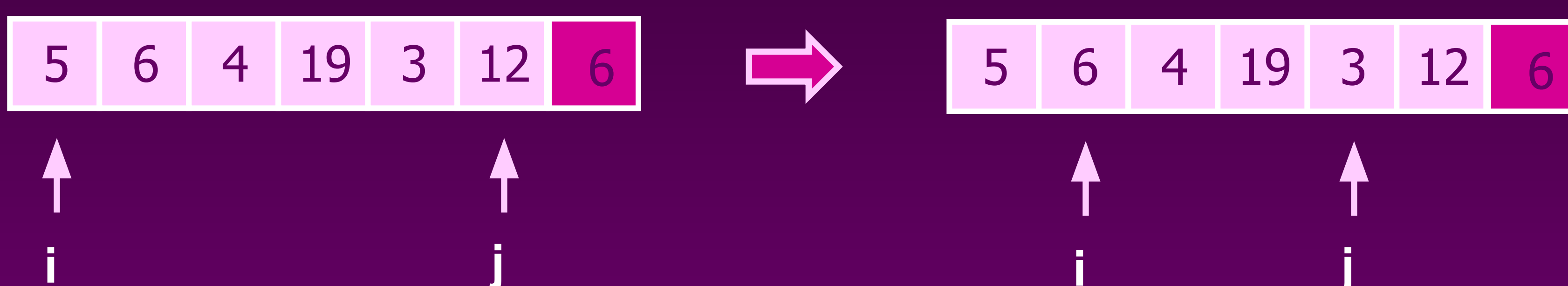
A better partition

- Want to partition an array A[left .. right]
- First, get the pivot element out of the way by swapping it with the last element. (Swap pivot and A[right])
- Let i start at the first element and j start at the next-to-last element (i = left, j = right – 1)

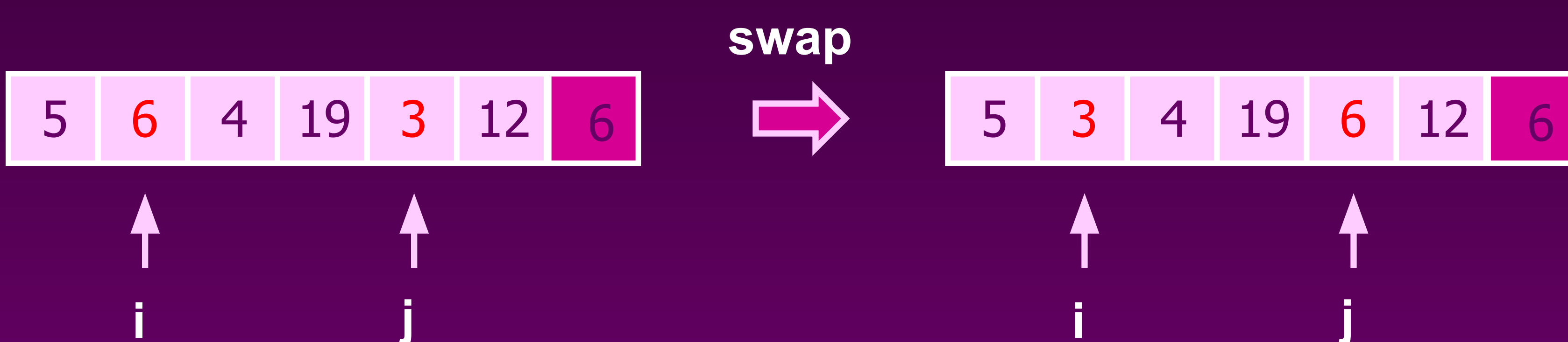


- Want to have
  - A[x] <= pivot, for x < i
  - A[x] >= pivot, for x > j
- When i < j
  - Move i right, skipping over elements smaller than the pivot
  - Move j left, skipping over elements greater than the pivot
  - When both i and j have stopped
    - A[i] >= pivot
    - A[j] <= pivot

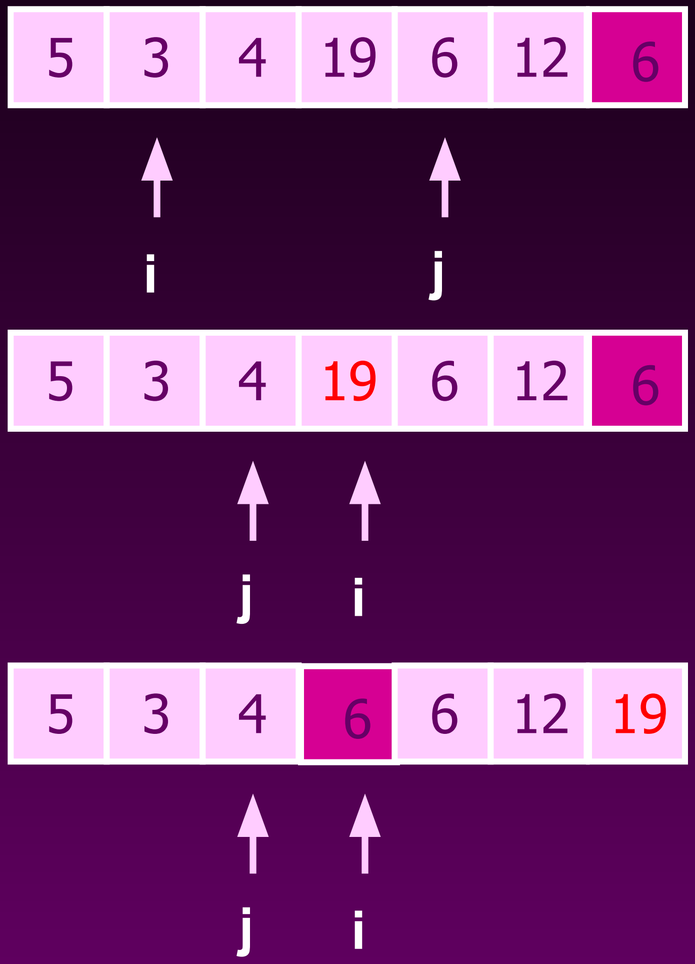




- When i and j have stopped and i is to the left of j
  - Swap A[i] and A[j]
    - The large element is pushed to the right and the small element is pushed to the left
  - After swapping
    - A[i] <= pivot
    - A[j] >= pivot
  - Repeat the process until i and j cross



- When i and j have crossed
  - Swap A[i] and pivot
- Result:
  - A[x] <= pivot, for x < i
  - A[x] >= pivot, for x > i



Implementation (put the pivot on the leftmost instead of rightmost)

```
void quickSort(int array[], int start, int end)
{
     int i = start; // index of left-to-right scan
     int k = end; // index of right-to-left scan

     if (end - start >= 1) // check that there are at least two elements to sort
     {
          int pivot = array[start]; // set the pivot as the first element in the partition
          while (k > i) // while the scan indices from left and right have not met,
          {
               while (array[i] <= pivot && i <= end && k > i)    // from the left, look for the fi
                    i++;                     // element greater than the pivot
               while (array[k] > pivot && k >= start && k >= i) // from the right, look for the fi
                    k--;                     // element not greater than the pivot
               if (k > i)                    // if the left seekindex is still smaller than
                    swap(array, i, k);            // the right index,
                                        // swap the corresponding elements
          }
          swap(array, start, k);                  // after the indices have crossed,
                                        // swap the last element in
                                        // the left partition with the pivot
          quickSort(array, start, k - 1);              // quicksort the left partition
          quickSort(array, k + 1, end);               // quicksort the right partition
     }
     else // if there is only one element in the partition, do not do any sorting
     {
      return; // the array is sorted, so exit
     }
}
```

Adapted from
http://www.mycsresource.net/articles/programming/sorting_algos/quicksort/

```
void quickSort(int array[])
// pre: array is full, all elements are non-null integers
// post: the array is sorted in ascending order
{
    quickSort(array, 0, array.length - 1); // quicksort all the elements in the ar
}
void quickSort(int array[], int start, int end)
{
…
}
void swap(int array[], int index1, int index2) {…}
// pre: array is full and index1, index2 < array.length
// post: the values at indices 1 and 2 have been swapped
```

With duplicate elements …

- Partitioning so far defined is ambiguous for duplicate elements (the equality is included for both sets)
- Its ‘randomness’ makes a ‘balanced’ distribution of duplicate elements
- When all elements are identical:
    - both i and j stop many swaps
    - but cross in the middle, partition is balanced (so it’s n log n)

A better Pivot

Use the median of the array

- Partitioning always cuts the array into roughly half
- An optimal quicksort (O(N log N))
- However, hard to find the exact median (chicken-egg?)
  - e.g., sort an array to pick the value in the middle
- Approximation to the exact median: …

Median of three

- We will use median of three
  - Compare just three elements: the leftmost, rightmost and center
  - Swap these elements if necessary so that
    - A[left] = Smallest
    - A[right] = Largest
    - A[center] = Median of three
  - Pick A[center] as the pivot
  - Swap A[center] and A[right – 1] so that pivot is at second last position (why?)

median3

```
int center = ( left + right ) / 2;
if( a[ center ] < a[ left ] )
    swap( a[ left ], a[ center ] );
if( a[ right ] < a[ left ] )
    swap( a[ left ], a[ right ] );
if( a[ right ] < a[ center ] )
    swap( a[ center ], a[ right ] );

    // Place pivot at position right - 1
swap( a[ center ], a[ right - 1 ] );
```

2	5	6	4	13	3	12	19	6

A[left] = 2, A[center] = 13,
A[right] = 6

2	5	6	4	6	3	12	19	13

Swap A[center] and A[right]

2	5	6	4	6	3	12	19	13

Choose A[center] as pivot

pivot

2	5	6	4	19	3	12	6	13

Swap pivot and A[right – 1]

pivot

Note we only need to partition A[left + 1, …, right – 2]. Why?

- Works only if pivot is picked as median-of-three.
  - A[left] <= pivot and A[right] >= pivot
  - Thus, only need to partition A[left + 1, …, right – 2]
- j will not run past the beginning
  - because a[left] <= pivot
- i will not run past the end
  - because a[right-1] = pivot

```
int i = left, j = right - 1;
for( ; ; )
{
    while( a[ ++i ] < pivot ) { }
    while( pivot < a[ --j ] ) { }
    if( i < j )
        swap( a[ i ], a[ j ] );
    else
        break;
}
```

The coding style is efficient, but hard to read ☹

```
i=left;
j=right-1;

while (1) {
    do i=i+1;
   while (a[i] < pivot);

    do j=j-1;
    while (pivot < a[j]);

    if (i<j) swap(a[i],a[j]);
    else break;
}
```

```
int i = left, j = right - 1;
for( ; ; )
{
    while( a[ ++i ] < pivot ) { }
    while( pivot < a[ --j ] ) { }
    if( i < j )
        swap( a[ i ], a[ j ] );
    else
        break;
}
```

Small arrays

- For very small arrays, quicksort does not perform as well as insertion sort
  - how small depends on many factors, such as the time spent making a recursive call, the compiler, etc
- Do not use quicksort recursively for small arrays
  - Instead, use a sorting algorithm that is efficient for small arrays, such as insertion sort

A practical implementation

```
if( left + 10 <= right )
{
    Comparable pivot = median3( a, left, right );

        // Begin partitioning
    int i = left, j = right - 1;
    for( ; ; )
    {
        while( a[ ++i ] < pivot ) { }
        while( pivot < a[ --j ] ) { }
        if( i < j )
            swap( a[ i ], a[ j ] );
        else
            break;
    }

    swap( a[ i ], a[ right - 1 ] );  // Restore pivot

    quicksort( a, left, i - 1 );     // Sort small elements
    quicksort( a, i + 1, right );    // Sort large elements
}
else  // Do an insertion sort on the subarray
    insertionSort( a, left, right );
```

Choose pivot

Partitioning

Recursion

For small arrays

Quicksort Analysis

- Assumptions:
  - A random pivot (no median-of-three partitioning)
  - No cutoff for small arrays
- Running time
  - pivot selection: constant time, i.e. $O(1)$
  - partitioning: linear time, i.e. $O(N)$
  - running time of the two recursive calls
- $T(N)=T(i)+T(N-i-1)+cN$ where c is a constant
  - i: number of elements in S1

Worst-Case Analysis

- What will be the worst case?
  - The pivot is the smallest element, all the time
  - Partition is always unbalanced

$$
\begin{aligned}
T(N) &= T(N-1) + cN \\
T(N-1) &= T(N-2) + c(N-1) \\
T(N-2) &= T(N-3) + c(N-2) \\
&\vdots \\
T(2) &= T(1) + c(2) \\
T(N) &= T(1) + c\sum_{i=2}^{N} i = O(N^2)
\end{aligned}
$$

Best-case Analysis

- What will be the best case?
  - Partition is perfectly balanced.
  - Pivot is always in the middle (median of the array)

$$
\begin{aligned}
T(N) &= 2T(N/2) + cN \\
\frac{T(N)}{N} &= \frac{T(N/2)}{N/2} + c \\
\frac{T(N/2)}{N/2} &= \frac{T(N/4)}{N/4} + c \\
\frac{T(N/4)}{N/4} &= \frac{T(N/8)}{N/8} + c \\
&\vdots \\
\frac{T(2)}{2} &= \frac{T(1)}{1} + c \\
\frac{T(N)}{N} &= \frac{T(1)}{1} + c\log N \\
T(N) &= cN\log N + N = O(N\log N)
\end{aligned}
$$

Average-Case Analysis

- Assume
  - Each of the sizes for S1 is equally likely
- This assumption is valid for our pivoting (median-of-three) strategy
- On average, the running time is O(N log N) (covered in comp271)

Quicksort is ‘faster’ than Mergesort

- Both quicksort and mergesort take O(N log N) in the average case.
- Why is quicksort faster than mergesort?
  - The inner loop consists of an increment/decrement (by 1, which is fast), a test and a jump.
  - There is no extra juggling as in mergesort.

```
int i = left, j = right - 1;
for( ; ; )
{
    while( a[ ++i ] < pivot ) { }
    while( pivot < a[ --j ] ) { }
    if( i < j )
        swap( a[ i ], a[ j ] );
    else
        break;
}
```

inner loop