# More Sorting

# Quicksort

- Quicksort  is very commonly used, efficient, comparison based sorting algorithm.
- It is a **divide and conquer** algorithm.
- Not stable.

- Its average case time complexity is O ( N log N),
  - which is the same as Mergesort
  - but its constants are typically better than Mergesort
  - Hence it usually performs better than Mergsort.

# Quicksort

```
QuickSort ( A )
        QSort ( A, 0, A.length – 1 );
```

---

```
QSort ( A, start, end )
        if ( start == end )
                return;

        pivotIndex = PickAPivotIndex ( start, end )

        pivotFinalIndex =  Partition ( A, start, end, pivotIndex )      // Now the element at pivotFinalIndex is at its final sorted position.

        QSort( A, start, pivotFinalIndex – 1 );                         // sort sub array to left of pivotFinalIndex

        QSort( A, pivotFinalIndex + 1,  end );                          // sort sub array to right of pivotFinalIndex
```

# Quicksort

1st row shows state after partition has completed

| start | | | | leftMostBigger | | | | | end |
|---|---|---|---|---|---|---|---|---|---|
| 16 | 4 | 7 | 3 | 17 | 34 | 25 | 52 | 88 | 43 |
| | | | | | | | | | |

***Elements*** *less than pivot*

***Elements*** *greater than or equal to pivot*

# Quicksort

2nd row below shows state **after swap** happens ( Note: swap happens after partition completes )

| start | | | | leftMostBigger | | | | | end |
|---|---|---|---|---|---|---|---|---|---|
| 16 | 4 | 7 | 3 | 17 | 34 | 25 | 52 | 88 | 43 |
| 3 | 4 | 7 | 16 | 17 | 34 | 25 | 52 | 88 | 43 |

***Elements** less than pivot*
***Elements** greater than or equal to pivot*

Note that pivot element 16 is now in its final place in the sorted array

# Quicksort

| start | | | | leftMostBigger | | | | | end |
|---|---|---|---|---|---|---|---|---|---|
| 3 | 4 | 7 | 16 | 17 | 34 | 25 | 52 | 88 | 43 |

Now, we will call QSort on

- sub array to left of partition element
- sub array to right of partition element
- and we won't touch the partition element

# Quicksort

**PickAPivotIndex** (start, end )

return random number between start and end ( inclusive ).

---

// move all elements less than pivot value to left of pivot, and all elements greater than pivot value to its right.

**Partition** ( A, start, end, pivotIndex )

```
Swap ( A[ start ], A [ pivotIndex ] );

newStart = start + 1;
leftMostBiggerIndex = newStart;

for ( ii = newStart; ii <= end;  ++ ii )
        if   A [ ii ] < A [ pivotIndex ]
                Swap ( A [ ii ], A [ leftMostBiggerIndex ] );
                ++ leftMostBiggerIndex;

pivotFinalIndex = leftMostBiggerIndex – 1;          // final position of pivot is  immediate left of leftMostBiggerIndex
Swap ( A[ start ], A[ pivotFinalIndex ] );          // put pivot in its final position

return pivotFinalIndex;
```

# Quicksort

- In place
- Not stable

- Time complexity
  - Best case and average : O ( N log N )
  - Worst case: O ( $N^2$ )
    - Worst case can happen:
      1. if pivots are consistently smallest or largest between *start* and *end*.
         However, this is very hard to achieve ☺ and probability of that happening is extremely low.
         See next slide for details.

      2. If there are a lot of duplicates.
         In this case, use 3 way partitioning. (Look up the Dutch national flag problem).

- Space complexity: O ( log N ) due to recursion.

# Quicksort

- One way you would get the worst case time:
  - Take a sorted or almost sorted array (or reverse sorted).
  - Always pick the first or last element between *start* and *end* as the pivot.
  - This means u will partition the array with one element on one side and rest of the array on other side.
  - To prevent against this, pick a random pivot.
  - Note: You could also pick the middle element and be OK, but input can be crafted against this methodology, so better to pick random.

# Quicksort

- Cache locality
  - Quicksort has good cache locality.

  - Its better than Merge sort (which also accesses consecutive elements in an array) because Mergesort uses an auxiliary array.

# LAB 1 of 2

- Write a function that sorts the input array using the quick sort algorithm.
  - For picking the pivot:
    - If u know a random number generation function, can use it.
    - If not, just pick middle element  (from start and end ) for now.

# Quicksort

Time complexity recurrence relation

**Best case:**

$T( N ) = 2\ T ( N / 2 ) + cN$ ← Assumes we partition the array into **half** each time ( unreasonable assumption ☺ )

$T( N/2 ) = 2\ T ( N / 4 ) + cN/2$

$T( N/4 ) = 2\ T ( N / 8 ) + cN/4$

and so on…

Above comes to $T( N ) = N \log N$

# Quicksort

Time complexity recurrence relation

**Worst case:** ← Every time, we reduce the array by only 1 element (pick the biggest or smallest element each time).

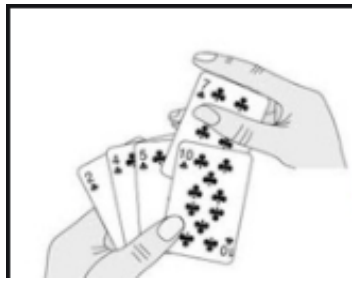
$$
\begin{aligned}
T(N) &= T(N-1) + cN \\
&= T(N-2) + c(N-1) + cN \\
&= T(1) + c2 + c3 + \ldots + c(N-1) + cN \\
&= c1 + c2 + c3 + \ldots + c(N-1) + cN \\
&= c\,(1 + 2 + 3 + \ldots + (N-1) + N) \\
&= c\,(N(N+1)/2) \\
&= O(N^2)
\end{aligned}
$$

# Insertion sort

- Its not very efficient in general.
- It’s a great choice if, say, the input is already sorted and u want to add a new element to it… O ( N )
- In place
- Stable

- Time complexity
  - Worst and average case: O ($N^2$ )
  - Best case: O ( N )

# Insertion sort



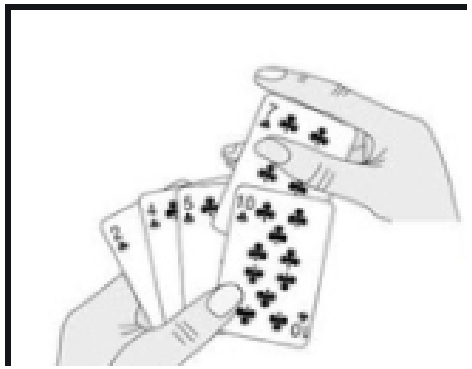
- Lets think of this algo in the following way:
  - You have to sort N numbers
  - Assume the first N-1 numbers are somehow already sorted.
  - Now, the problem is just to put the $N^{th}$ number in the right place inside sorted array with N-1 elements.
  - Extend that thinking to:
    - assume first N-2 numbers already sorted.
    - Now u need to just put the ( N-1 )$^{th}$ and $N^{th}$ number in the right place inside this sorted array.
    - And so on…
  - Here we are thinking of a top down approach (recursive).
  - Another approach would be bottom up (iterative)

# Insertion sort - pseudocode for recursive

```
InsertionSort ( A )
        InSort( A, A.length – 1 );


InSort( A, N )
        if ( N <= 1 )
                return;
        InSort( A, N – 1 );                    // first lets sort N-1 numbers

        // If we r here, it means we sorted N – 1 numbers already ( and the first time we will come here is when N has a value of 2, so N-1 is just 1 )
        // remember, N here is NOT ALWAYS A.length-1, we are recursing, so N is whatever it is on call stack here

        i = N;                                 // We look for a place to insert A [ N ]
        while ( i > 0  and  A[ i ] < A [ i – 1 ] )    // While the element is less than its immediate left neighbor
                Swap ( A[ i ], A [ i – 1 ] );
                --i;
```

# Insertion sort

- Iterative approach
  - I can solve the problem for one element.
  - Then solve for two elements.
  - …
  - Then solve for N  elements



# Insertion sort – pseudocode for Iterative

```
InsertionSort ( A )
        InSort( A, A.length – 1 );

InSort( A, N )
        if ( N <= 1 )
                return;

        for i = 1 to N          // start from index 1, think of element at 0th index as a sorted sub array of size 1.
                j = i – 1;
                while ( j >= 0   and  A [ j+1 ] < A [ j ] )          // While the element is less than its immediate left neighbor
                        Swap ( A[ j + 1 ],  A [ j ] );
                        --j;
```

QQ: What implications if u change the comparison to be <=

```
                while ( j >= 0   and  A [ j+1 ] <= A [ j ] )
                        Swap ( A[ j + 1 ],  A [ j ] );
                        --j;
```

# Insertion sort

Example:

- Input: 16 4 $7_a$ 3 17 34 25 $7_b$

- 16 4 $7_a$ 3 17 34 25 $7_b$
- 4 16 $7_a$ 3 17 34 25 $7_b$
- 4 $7_a$ 16 3 17 34 25 $7_b$
- 3 4 $7_a$ 16 17 34 25 $7_b$
- 3 4 $7_a$ 16 17 34 25 $7_b$
- 3 4 $7_a$ 16 17 34 25 $7_b$
- 3 4 $7_a$ 16 17 25 34 $7_b$
- 3 4 $7_a$ $7_b$ 16 17 25 34 ← Sorted, and stable

# Insertion sort

- Lets look at the **best case** … input is already sorted (or almost)

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| • | 4 | $7_a$ | $7_b$ | 16 | 17 | 25 | 34 ← Starting input |
| • | 4 | $7_a$ | $7_b$ | 16 | 17 | 25 | 34 |
| • | 4 | $7_a$ | $7_b$ | 16 | 17 | 25 | 34 ← No movement here or in any row below |
| • | 4 | $7_a$ | $7_b$ | 16 | 17 | 25 | 34 |
| • | 4 | $7_a$ | $7_b$ | 16 | 17 | 25 | 34 |
| • | 4 | $7_a$ | $7_b$ | 16 | 17 | 25 | 34 |
| • | 4 | $7_a$ | $7_b$ | 16 | 17 | 25 | 34 |
| • | 4 | $7_a$ | $7_b$ | 16 | 17 | 25 | 34 |

- First element is compared with its immediate left neighbor, and we decide nothing needs to be done.
- Move to next element, again compare with its immediate left neighbor, nothing needs to be done.
- So, in each iteration, there is one comparison and no movement…. O ( 1 )
- Doing this for N-1 elements … O ( N ) is the complexity for this best case scenario

# Insertion sort

- Now lets look at the **worst case** scenario … input is reverse sorted.
  - Every element in the starting input is less than ALL the elements to its left.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| • | 34 | 25 | 17 | 16 | $7_a$ | $7_b$ | 4 |
| • | 34 | 25 | 17 | 16 | $7_a$ | $7_b$ | 4 | ← 25: *1 comparison* |
| • | 25 | 34 | 17 | 16 | $7_a$ | $7_b$ | 4 | ← 17: < all elements to its left. *2 comparisons* |
| • | 17 | 25 | 34 | 16 | $7_a$ | $7_b$ | 4 | ← 16: 3 comparisons for moving 16 |
| • | 16 | 17 | 25 | 34 | $7_a$ | $7_b$ | 4 |
| • | $7_a$ | 16 | 17 | 25 | 34 | $7_b$ | 4 |
| • | $7_a$ | $7_b$ | 16 | 17 | 25 | 34 | 4 | ← 4: 6 comparisons → *N-1 comparisons* |
| • | 4 | $7_a$ | $7_b$ | 16 | 17 | 25 | 34 | ← 4 had to move all the way to the beginning |

- Total comparisons: 1 + 2 + … + (N-1) = O ($N^2$ )

# Counting sort

- Counting sort is not a comparison based algorithm.
- It is not a general purpose sorting algorithm… values have to be integers within a known “small” range.

- Not in place
- Usually linear time and space complexity.

# Counting sort

- Input: 2 4 7 3 1 2 0 7 7

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| **Counts / frequency** | 1 | 1 | 2 | 1 | 1 | 0 | 0 | 3 |

- We use a counting array, whose size is K where K is the maximum value in the input ← **Space is O ( K )**

- For each element x in input ← **O ( N )**
  - ++ countingArray[ x ] ← **O ( 1 )**

- For each index ii in the counting array ← **O ( K )**
  - If ( countingArray [ ii ] > 0 )
    - print ii // number of times we print is countingArray [ ii ]

# Counting sort

- Input: 2 4 7 3 1 2 0 7 7

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| **Counts / frequency** | 1 | 1 | 2 | 1 | 1 | 0 | 0 | 3 |

- We use a counting array, whose size is K where K is the maximum value in the input ← **Space is O ( K )**

- Initialize countingArray to all 0s ← **O ( K )**

- For each element x in input ← **O ( N )**
  ++ countingArray[ x ] ← **O ( 1 )**

- For each index ii in the counting array ← **O ( K )**
  If ( countingArray [ ii ] > 0 )
  print ii // number of times we print is countingArray [ ii ]

- Complexity
  - Time: O ( N + K ).
  - Space: O ( K ). ← we will see how this might change a bit due to counting sort on objects instead of integers.
  - If K is "small enough", then this is linear complexity (this is when this algo would be / could be used).
  - If K is, say, $N^2$, then this is not linear time complexity, or linear space complexity.

- **Next slide**: What if we have objects as input, instead of just numbers.

# Counting sort -- Objects

- If we are dealing with objects (common scenario) instead of numbers:
  - We would need to store the objects in the counting array.
  - And Count sort would still need integers to sort
    - Some integer attribute of the objects (or something that can be mapped to an integer value)
  - And to keep the sort stable, we would keep the “order of insertion” in mind.
    - “order of insertion” refers to storing more than one object having the same sorting criteria.

| Index | 0 | 1 | 2 | … | K-1 |
|---|---|---|---|---|---|
| | Object A | Object B1 | | Object C | Object D1 |
| | | Object B2 | | | Object D2 |
| | | | | | Object D3 |

- Space complexity now is:
  - O ( K + N ) since we have a counting array of size K and we store N objects in it.

# Choosing a sort algo

- It depends on what is needed in ur solution to the problem u r trying to solve.
- If u care about stability: Merge Sort.
  - Trade off ?
- If u cannot use extra space: Quicksort
  - Trade off?
- If u have a “small” sized input or almost sorted data: Insertion Sort
- If input is in a restricted range: Count sort
- When would u use Heapsort?
  - Pros: Guaranteed O( N log N ) time. O(1) space
  - Cons: Not stable. Bad cache locality

# Sorting cheat sheet

| Algo | Worst | Avg | Best | Stable | InPlace | Space | Other |
|---|---|---|---|---|---|---|---|
| Bubble | $N^2$ | $N^2$ | $N^2$ | Yes | Yes | O( 1 ) | Comparison sort.<br>Stable bcos same element wont jump over. E.g.: 3 9 7a 11 17 7b Here, 7b will not go to the left of 7a, so relative ordering maintained. |
| Insertion | $N^2$ | $N^2$ | N | Yes | Yes | O( 1 ) | Comparison sort.<br>Best is N bcos no shifting or almost no shifting is required when array is sorted or almost sorted.<br>Worst case is when reverse sorted.<br>Running time can change depending on values and position of values (unlike Bubble or Selection sort) |
| Selection | $N^2$ | $N^2$ | $N^2$ | No (see comments) | Yes | O( 1 ) | Comparison sort.<br>Not stable because this can happen: $7_A$ 3 9 $7_B$ 1 Find min, which is 1, and swapping with first elem gives 1 3 9 $7_B$ $7_A$. So order of $7_A$ and $7_B$ is switched, making it unstable. To make it stable, can insert 1 in its place, instead of swapping (but that is expensive due to shifting).<br>1 $7_A$ 3 9 $7_B$ |
| Quicksort | $N^2$ | N log N | N log N | No | Yes | O (log N) | Comparison sort. QS is in place, but does use stack space for recursion.<br>Standard QS not stable, but can use extra mem to get a stable version, but then its not in place.<br>Standard implementation is in place (not stable). Its constants are usually better than MS (merge sort), hence it is usually faster than MS.<br>Worst case can happen if array is already sorted (or reverse sorted) or almost sorted, and u pick first or last element as pivot. If u always pick the middle element, input can be crafted to cause worst case behavior. Just pick random element.<br>Note, if array is randomly ordered (not sorted), then picking first or last element as pivot would be ok, but its better to pick a random element (or median) as pivot to guard against the case where array is already sorted or almost sorted.<br>QS can be implemented iteratively if u use ur own stack.<br>Cache locality of QS is good. Often it is better than cache locality for MS, because MS uses a 2nd array in addition to input array. |
| Mergesort | N log N | N log N | N log N | Yes | No | O( N ) | Comparison sort. Run time is predictable, always O (N log N) regardless of input values/position.<br>Cache locality is good, but QS generally has slightly better cache locality since MS uses an additional array (its not in place) |
| Heapsort | N log N | N log N | N log N | No | Yes | O( N ) | Comparison sort. Cache locality not great due to all the long range swapping within the heap. Guaranteed O( N log N) time, O(1) space |
| Radixsort | O( nK ) | O( nK ) | O( nK ) | Yes | No | O( N+ K ) | |
| Counting sort | | | | Yes | No | O( K ) | Not in place as it uses a frequency array |
| Bucket sort | | | | | | O( N ) | |
| Tim sort | N log N | N log N | N | yes | | O( N ) | Hybrid of merge and insertion. Used in Python. Used in Java to sort objects (because stability could be important when sorting objects due to objects potentially having metadata/satellite data. But in Java if u r sorting primitive types, stability is not important and it uses Quicksort since that is typically faster than Mergesort. |
| Introsort | | | | No | Yes | | Comparison sort. Hybrid of QS, heap and insertion. Starts with QS, if recursion depth exceeds some limit, switches to heap sort, and if number of elements falls below a threshold, switches to insertion sort.<br>C++ sort *typically* uses Introsort (could vary depending on implementation of C++ STL). C++ stable_sort typically uses a version of mergesort. |

# LAB 2 of 2

- Take the same input you used in the quicksort lab (just so u can compare the two algorithm run times)
- Implement insertion sort
- Do u see a difference in the run times of quicksort and insertion sort when running on the same input?