

Quicksort

Sorting algorithms

- Insertion, selection and bubble sort have quadratic worst-case performance
- The faster comparison based algorithm ?
 $O(n \log n)$
- Mergesort and Quicksort

Quicksort Algorithm

Given an array of n elements (e.g., integers):

- If array only contains one element, return
- Else
 - pick one element to use as *pivot*.
 - Partition elements into two sub-arrays:
 - Elements less than or equal to pivot
 - Elements greater than pivot
 - Quicksort two sub-arrays
 - Return results

Example

We are given array of n integers to sort:

40	20	10	80	60	50	7	30	100
----	----	----	----	----	----	---	----	-----

Pick Pivot Element

There are a number of ways to pick the pivot element. We will use the first element in the array:

40	20	10	80	60	50	7	30	100
----	----	----	----	----	----	---	----	-----

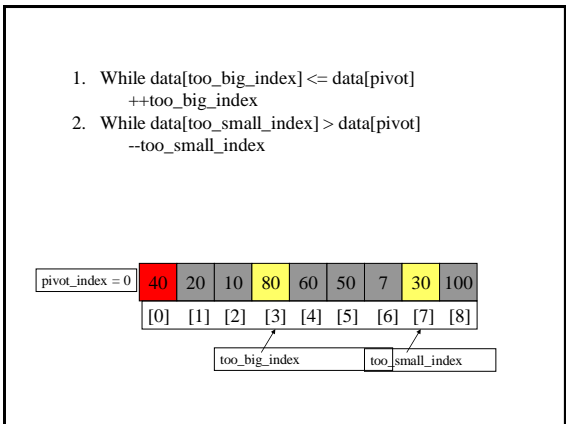
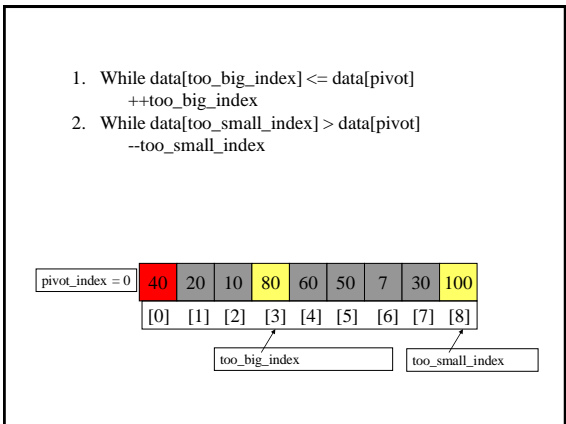
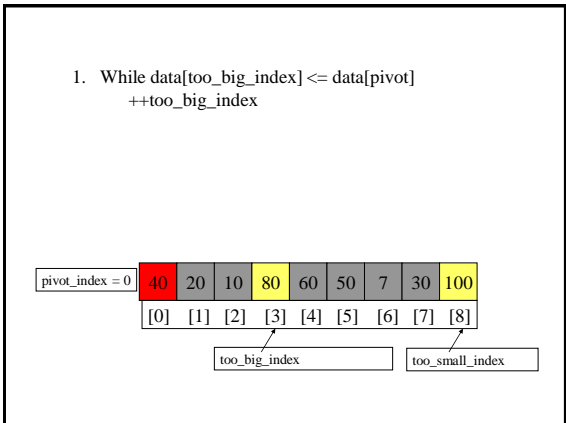
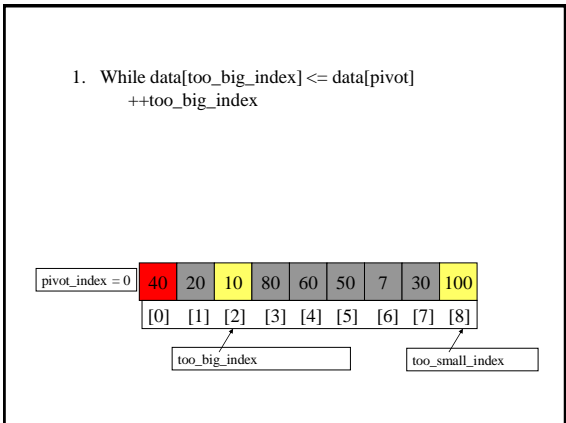
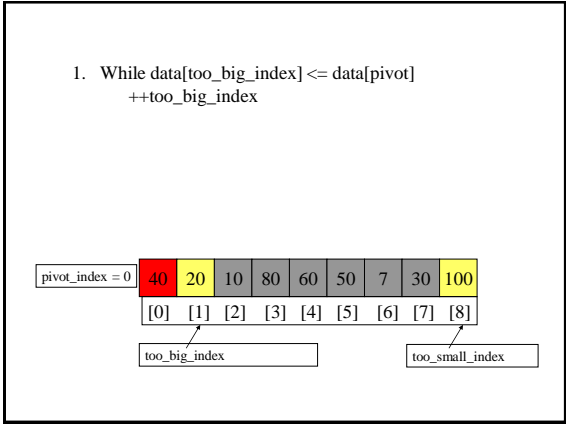
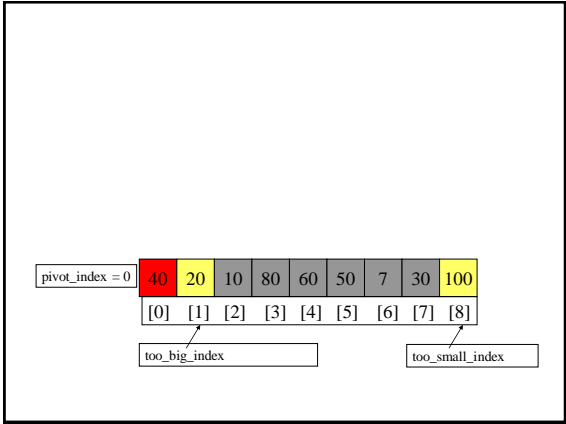
Partitioning Array

Given a pivot, partition the elements of the array such that the resulting array consists of:

1. One sub-array that contains elements \geq pivot
2. Another sub-array that contains elements $<$ pivot

The sub-arrays are stored in the original data array.

Partitioning loops through, swapping elements below/above pivot.



-
- Diagram illustrating the partitioning step in quicksort. The array is [40, 20, 10, 80, 60, 50, 7, 30, 100]. The pivot is 40 (at index 0). Elements 80 and 30 are highlighted in yellow. Red arrows indicate the swap of 80 and 30. Below the array, 'too_big_index' points to index 3 and 'too_small_index' points to index 7.

-
- Diagram illustrating the partitioning step of the quicksort algorithm. A pivot element (40) is chosen from the array [40, 20, 10, 30, 60, 50, 7, 80, 100]. Elements less than the pivot are moved to the left, and elements greater than the pivot are moved to the right. The pivot is then placed in its final sorted position. The diagram shows the array being partitioned around the pivot 40, with elements less than 40 (20, 10, 30) on the left and elements greater than 40 (60, 50, 7, 80, 100) on the right. The pivot 40 is placed in its final position at index 3. The diagram also shows the pivot index (0) and the indices of the elements being compared (too_big_index and too_small_index).

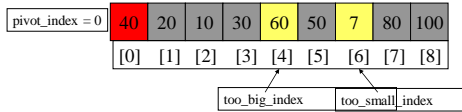
-
- Diagram illustrating a partitioning step in an array. The array is `[40, 20, 10, 30, 60, 50, 7, 80, 100]`. The pivot is `40` at index `0`. Elements less than the pivot are in indices `1-3`, and elements greater than the pivot are in indices `4-8`. The partitioning process is shown with arrows indicating the movement of elements.

-
- Diagram illustrating the partitioning step in Quicksort. The array is [40, 20, 10, 30, 60, 50, 7, 80, 100]. The pivot is 30 (at index 3). Elements less than the pivot (40, 20, 10, 7) are in red. Elements greater than the pivot (60, 50, 80, 100) are in yellow. Elements equal to the pivot (30) are in grey. The pivot_index is 0. The too_big_index is 3 and the too_small_index is 7.

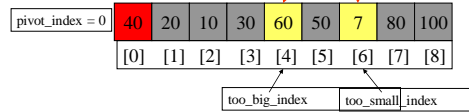
-
- Diagram illustrating a partitioning step in an array. The array is `[40, 20, 10, 30, 60, 50, 7, 80, 100]`. The pivot is `40` at index `0`. Elements less than the pivot are in red (20, 10, 30) and elements greater than the pivot are in yellow (60, 80). Arrows indicate the partitioning process: `too_big_index` points to index 4 (value 50) and `too_small_index` points to index 7 (value 80).

-
- Diagram illustrating a partitioning step in an array. The array is `[40, 20, 10, 30, 60, 50, 7, 80, 100]`. The pivot is `40` at index `0`. Elements less than the pivot are in the "too_small" region, and elements greater than the pivot are in the "too_big" region. The pivot is at index `4`.

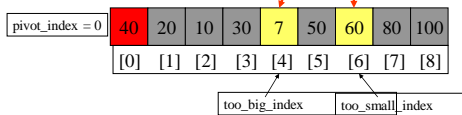
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $++\text{too_big_index}$
- 2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $--\text{too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.



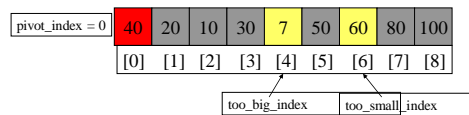
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $++\text{too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $--\text{too_small_index}$
- 3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.



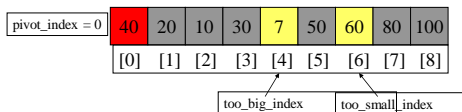
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $++\text{too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $--\text{too_small_index}$
- 3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.



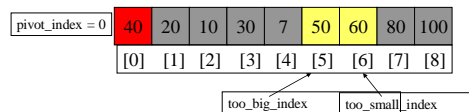
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $++\text{too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $--\text{too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
- 4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.



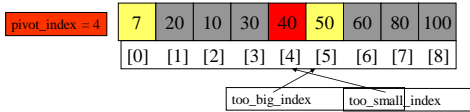
- 1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $++\text{too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $--\text{too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.



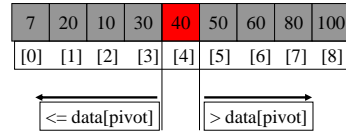
- 1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $++\text{too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $--\text{too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.



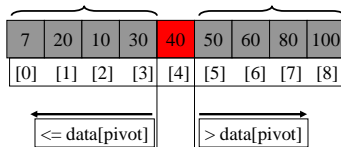
1. While `data[too_big_index] <= data[pivot]`
 `++too_big_index`
2. While `data[too_small_index] > data[pivot]`
 `--too_small_index`
3. If `too_big_index < too_small_index`
 swap `data[too_big_index]` and `data[too_small_index]`
4. While `too_small_index > too_big_index`, go to 1.
- 5. Swap `data[too_small_index]` and `data[pivot_index]`



Partition Result



Recursion: Quicksort Sub-arrays



A practical implementation

```

if( left + 10 <= right )
{
    Comparable pivot = median3( a, left, right ); // Choose pivot
    // 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 ); // For small arrays

```

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

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
 - Recursion:
 1. Partition splits array in two sub-arrays of size $n/2$
 2. Quicksort each sub-array

Quicksort Analysis

- Assumption:
 - A random pivot
- Running time
 - pivot selection: constant time, i.e. $O(1)$
 - partitioning: linear time, i.e. $O(N)$
 - running time of the two recursive calls
- $T(N) = O(1) + O(N) + T(i) + T(N-i-1)$
- $T(N) = cN + T(i) + T(N-i-1)$
 - where c is a constant
 - i is number of elements in array 1

Best-case Analysis

- What will be the best case?
 - Partition is perfectly balanced.
 - Pivot is always in the middle (median of the array)
 - $T(N) = cN + 2T(N/2)$
 - $T(N) = O(N \log N)$

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

Average-Case Analysis

- Assume
 - Each of the sizes for S_1 is equally likely
- This assumption is valid for our pivoting (median-of-three) strategy
- On average, the running time is $O(N \log N)$

Quicksort Analysis

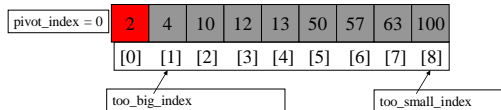
- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?

Quicksort Analysis

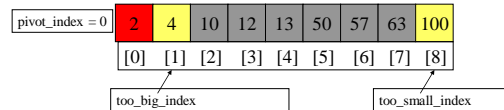
- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?
 - Recursion:
 1. Partition splits array in two sub-arrays:
 - one sub-array of size 0
 - the other sub-array of size $n-1$
 2. Quicksort each sub-array

Quicksort: Worst Case

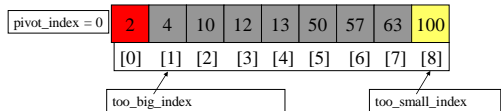
- Assume first element is chosen as pivot.
- Assume we get array that is already in order:



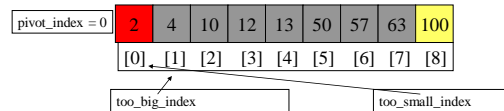
- While `data[too_big_index] <= data[pivot]`
 `++too_big_index`
- While `data[too_small_index] > data[pivot]`
 `--too_small_index`
- If `too_big_index < too_small_index`
 swap `data[too_big_index]` and `data[too_small_index]`
- While `too_small_index > too_big_index`, go to 1.
- Swap `data[too_small_index]` and `data[pivot_index]`



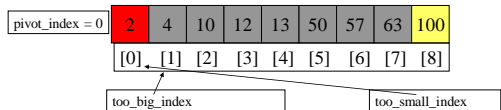
- While `data[too_big_index] <= data[pivot]`
 `++too_big_index`
- While `data[too_small_index] > data[pivot]`
 `--too_small_index`
- If `too_big_index < too_small_index`
 swap `data[too_big_index]` and `data[too_small_index]`
- While `too_small_index > too_big_index`, go to 1.
- Swap `data[too_small_index]` and `data[pivot_index]`



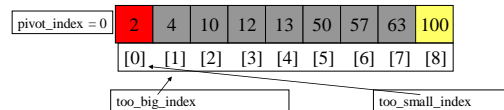
- While `data[too_big_index] <= data[pivot]`
 `++too_big_index`
- While `data[too_small_index] > data[pivot]`
 `--too_small_index`
- If `too_big_index < too_small_index`
 swap `data[too_big_index]` and `data[too_small_index]`
- While `too_small_index > too_big_index`, go to 1.
- Swap `data[too_small_index]` and `data[pivot_index]`



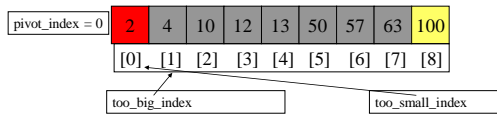
- While `data[too_big_index] <= data[pivot]`
 `++too_big_index`
- While `data[too_small_index] > data[pivot]`
 `--too_small_index`
- If `too_big_index < too_small_index`
 swap `data[too_big_index]` and `data[too_small_index]`
- While `too_small_index > too_big_index`, go to 1.
- Swap `data[too_small_index]` and `data[pivot_index]`



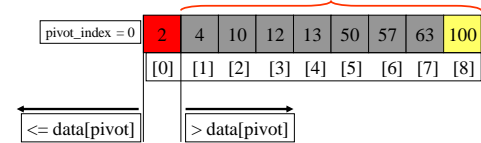
- While `data[too_big_index] <= data[pivot]`
 `++too_big_index`
- While `data[too_small_index] > data[pivot]`
 `--too_small_index`
- If `too_big_index < too_small_index`
 swap `data[too_big_index]` and `data[too_small_index]`
- While `too_small_index > too_big_index`, go to 1.
- Swap `data[too_small_index]` and `data[pivot_index]`



1. While `data[too_big_index] <= data[pivot]`
 `++too_big_index`
2. While `data[too_small_index] > data[pivot]`
 `--too_small_index`
3. If `too_big_index < too_small_index`
 swap `data[too_big_index]` and `data[too_small_index]`
4. While `too_small_index > too_big_index`, go to 1.
- 5. Swap `data[too_small_index]` and `data[pivot_index]`



1. While `data[too_big_index] <= data[pivot]`
 `++too_big_index`
2. While `data[too_small_index] > data[pivot]`
 `--too_small_index`
3. If `too_big_index < too_small_index`
 swap `data[too_big_index]` and `data[too_small_index]`
4. While `too_small_index > too_big_index`, go to 1.
- 5. Swap `data[too_small_index]` and `data[pivot_index]`



Worst-Case Analysis

- What will be the worst case?
 - The pivot is the smallest element, all the time
 - Partition is always unbalanced
 - `i=0`

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

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 organising 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
}

```

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time: $O(n^2)$

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time: $O(n^2)$
- What can we do to avoid worst case?

Improved Pivot Selection

Pick median value of three elements from data array:
`data[0]`, `data[n/2]`, and `data[n-1]`.

Use this median value as pivot.