Algorithms and Data Structures

A12. Sorting: Quicksort (& Heapsort)

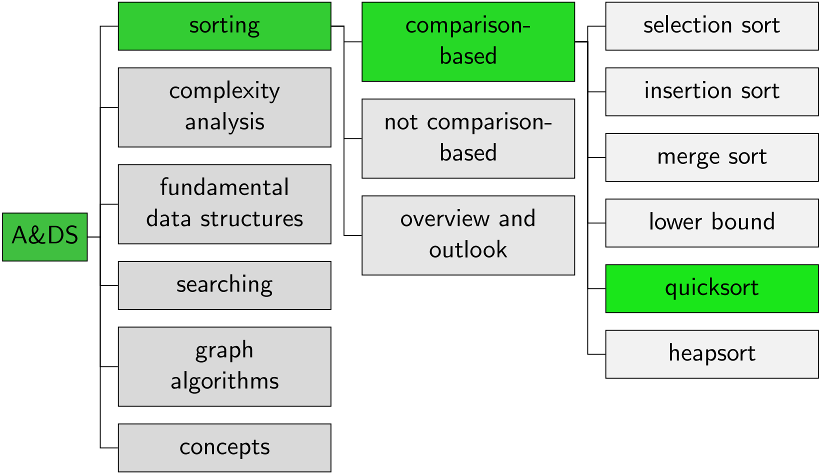
Gabriele Röger

University of Basel

March 20, 2024

Quicksort

Content of the Course



Quicksort: Idea

- Like merge sort a divide-and-conquer algorithm.
- In contrast to merge sort, the sequence is not divided by position but by values.
- For this purpose, select one element P (the so-called pivot).
- Divide (rearrange) the array, such that P is at its final position, left of P there are only elements $\leq P$, and right of P only elements $\geq P$.

$\leq P$	P	$\geq P$

- Conquer by calling quicksort recursively for the ranges left of P and right of P.
- Combine by doing nothing (recursive calls already lead to fully sorted array).

Quicksort: Algorithm

```
def sort(array):
    sort_aux(array, 0, len(array)-1)

def sort_aux(array, lo, hi):
    if hi <= lo:
        return
    choose_pivot_and_swap_it_to_lo(array, lo, hi)
    pivot_pos = partition(array, lo, hi)
    sort_aux(array, lo, pivot_pos - 1)
    sort_aux(array, pivot_pos + 1, hi)
```

How do we Choose the Pivot?

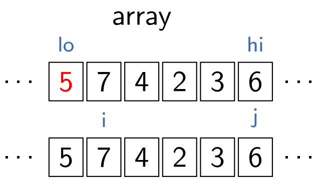
For the correctness of the algorithm, any choice is fine. (Why?)

Example strategies:

- Naïve: Always use the first element.
- Median of Three: Use median of first, middle and last element.
- Randomized: Use a random element.

Good pivots separate the range into roughly equally-sized ranges.

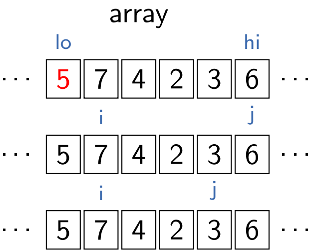
How do we Partition the Range?



Pivot is at position lo.

Initialize $i = \text{lo} + 1, j = \text{hi}$

How do we Partition the Range?



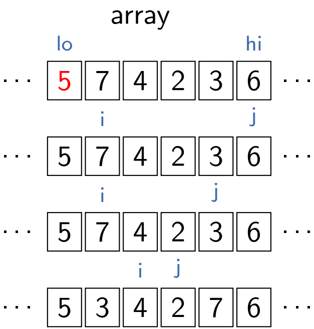
Pivot is at position lo.

Initialize $i = \text{lo} + 1, j = \text{hi}$

i to the right until element $\geq$ pivot

j to the left until element $\leq$ pivot

How do we Partition the Range?



Pivot is at position lo.

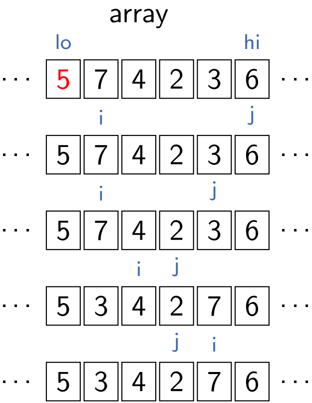
Initialize $i = \text{lo} + 1, j = \text{hi}$

i to the right until element $\geq$ pivot

j to the left until element $\leq$ pivot

If $i < j$: swap elements, $i{+}{+}$, $j{-}{-}$

How do we Partition the Range?



Pivot is at position lo.

Initialize $i = \text{lo} + 1, j = \text{hi}$

i to the right until element $\geq$ pivot

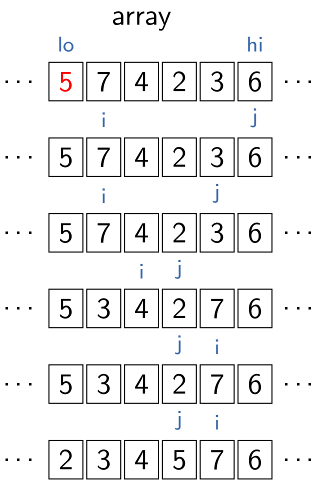
j to the left until element $\leq$ pivot

If $i < j$: swap elements, $i{+}{+}$, $j{-}{-}$

i to the right until element $\geq$ pivot

j to the left until element $\leq$ pivot

How do we Partition the Range?



Pivot is at position lo.

Initialize $i = \text{lo} + 1, j = \text{hi}$

i to the right until element $\geq$ pivot

j to the left until element $\leq$ pivot

If $i < j$: swap elements, $i{+}{+}$, $j{-}{-}$

i to the right until element $\geq$ pivot

j to the left until element $\leq$ pivot

$i \geq j$: swap pivot to position j

Done!

Quicksort: Partitioning

```python
def partition(array, lo, hi):
    pivot = array[lo]
    i = lo + 1
    j = hi
    while (True):
        while i < hi and array[i] < pivot:
            i += 1
        while array[j] > pivot:
            j -= 1
        if i >= j:
            break

        array[i], array[j] = array[j], array[i]
        i, j = i + 1, j - 1
    array[lo], array[j] = array[j], array[lo]
    return j
```

Exercise

What is the content of array [6, 5, 7, 8, 3] after a call of `partition` for the entire range (from position 0 to 4)?

Partitioning: Variants

- `partitioning` performs Hoare's partitioning method.
  - Tony Hoare: British computer scientist, inventor of quicksort
- There is also Lomuto's partitioning:
  - Inferior to Hoare's method.
  - Three times more swaps on average.
  - Leads to bad running time if all elements are equal.
  - Since it is easier to explain and analyze, used in some teaching resources (e.g. Cormen et al. textbook).
- We only consider Hoare's method.

Quicksort: Running Time I

Best case: Pivot separates into equally-sized ranges.

- $O(\log_2 n)$ recursive calls
- Each has hi-lo key comparisons during partitioning.
- On a single recursion depth overall $O(n)$ comparisons in partitioning.

$\rightarrow$ $O(n \log n)$

Quicksort: Running Time I

Best case: Pivot separates into equally-sized ranges.

- $O(\log_2 n)$ recursive calls
- Each has hi-lo key comparisons during partitioning.
- On a single recursion depth overall $O(n)$ comparisons in partitioning.

$\rightarrow$ $O(n \log n)$

Worst case: Pivot always smallest or largest element.

- overall n-1 (nontrivial) recursive calls for length $n, n-1, \ldots, 2$.
- Each has hi-lo key comparisons during partitioning.

$\rightarrow$ $\Theta(n^2)$

Quicksort: Running Time II

Average case:

- Assumption: n different elements,
  each of the $n!$ permutations has equal probability,
  random choice of pivot
- $O(\log n)$ recursive calls
- overall $O(n \log n)$
- $\approx 39\%$ slower than best case

Quicksort: Running Time II

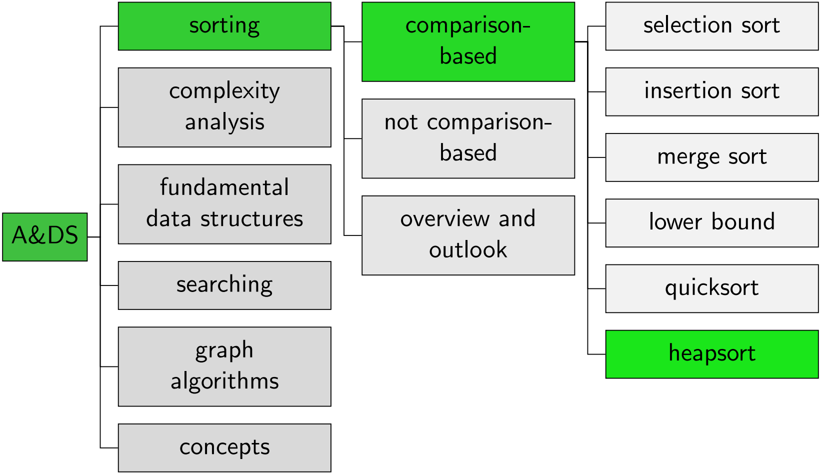
Average case:

- Assumption: n different elements,
  each of the $n!$ permutations has equal probability,
  random choice of pivot
- $O(\log n)$ recursive calls
- overall $O(n \log n)$
- $\approx 39\%$ slower than best case

With a random choice of the pivot, the worst case is extremely unlikely. Therefore, quicksort is often considered an $O(n \log n)$ algorithm.

Heapsort

Content of the Course



Heapsort

- Heap: data structure that allows to find and remove the largest element quickly:
  find: $\Theta(1)$, remove: $\Theta(\log n)$
- Basic idea as in selection sort but from right to left: Successively swap the largest element to the end of the non-sorted range.
- We can represent the heap directly in the input sequence, so that heap sort only needs constant additional storage.
- The running time is linearithmic.
- We cover the details once we have introduced heaps.

Summary

Summary

- Quicksort is a divide-and-conquer approach that divides the range relative to a pivot element.
- In the worst case, quicksort has quadratic running time.
- In the average case, the running time is linearithmic.
- With a random choice of the pivot, the worst case is extremely unlikely.