Quicksort

November 22, 2019 7:16 PM

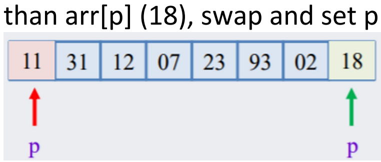
- **Recurrence relations**

```
double arrMax(double arr[], int size, int start) {
  if (start == size - 1)
    return arr[start];
  else
    return max( arr[start], arrMax(arr, size, start + 1) );
}
```

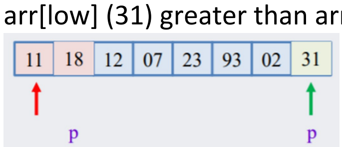
- **Merge sort analysis**
  - arlgorithm: split in half, sort first half, sort second half, merge together
  - T(n) E O(nlogn)
- **Binary search**
  - Inspect midpoint, recursively search left or right half of array
  - T(n) E O(logn)

Quicksort

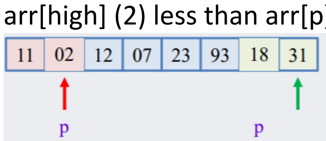
- efficient sorting algorithm than selection or insertion sort
  - sorts array by repeatedly partitioning it
- partitioning is the process of dividing an array into sections (partitions) based on some criteria
  - Big and small values
  - Negative and positive numbers
  - Names that begin with a-m names that begins with n-z
  - Darker and lighter pixels
- partitions roughly equal in size
  - partition array into small and big values using a partitioning algorithm
  - use 3 indices, place 1 at each end of array, low and high. third index starts at low
  - scan high from right to left until arr[high] is less than arr[p], arr[high] (11) is already less than arr[p] (18), swap and set p to high



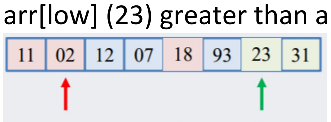
  - scan low from left to right until arr[low] greater than arr[p]
  - arr[low] (31) greater than arr[p] (18) so swap and set p to low



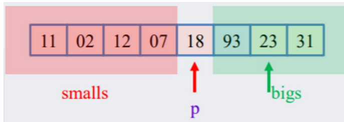
  - scan hgh from right to left until arr[high] less than arr[p]
  - arr[high] (2) less than arr[p] (18), swap and set p to high



  - scan low left-right until arr[low] greater than arr[p]
  - arr[low] (23) greater than arr[p] (18), swap, set p to low

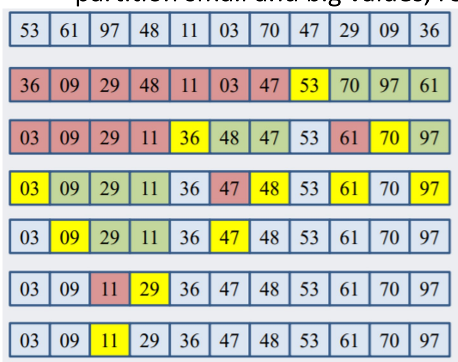


  - scan high right-left until arr[high] less than arr[p] or high = p
  - stop! index p contains pivot value, all elements of left of pivot = smaller values vs right of pivot = larger values, not ordered



Quicksort overview

- quicksort algorithm works by repeatedly partitioning array
- each time subarray partitioned, has sequence of small values, sequence of big values, and pivot value in correct position
- partition small and big values, repeat until each subarray consists of just 1 element



Quicksort Algorithm

- partition is where all comparisons are done, according to the Quicksort process

```
void qsort(int arr[], int low, int high) {
  int p;
  if (low < high) {
    p = partition(arr, low, high);
    qsort(arr, low, p-1);
    qsort(arr, p+1, end);
  }
}
```

```
void quicksort(int arr[], int size) {
  qsort(arr, 0, size-1);
}
```

Quicksort Analysis

- best and worst case
- Best case: each time a sub-array partitioned, the pivot is midpoint of slice, divided in half
  - each sub array divided in half in each partition, compared
  - process ends once sub arrays left to be partitioned are size 1
  - How many times does n have to be divided in half before result is 1?
    - $\log_2 n$ times, Quicksort performs $n*\log_2 n$ operations
- Worst case: array partitioned n times
  - n comparisions in first partition step, n-1 in second...
  - $\sum^{n-1}_{i=1} = n*(n-1)/2$, = around n^2
  - when nearly sorted in either direction
- Average case: randimize positions of array elements to fix partially sorted case
  - random scramble complexity: for n permutations: $O(n)+ O(n*\log_2 n)$

Merge vs Quicksort

- Quicksort worst case rare, easily avoided, faster than Merge sort
- can be sorted in place using O(logn) stack space
  - quicksort

Name	Best	Average	Worst	Stable	Memory
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	challenging	$O(1)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$	Yes	$O(1)$
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Yes	$O(n)$
Quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	challenging	$O(\log n)$