ECE36800 Data structures

Quicksort

Chapter 7 (pp. 303-335)

Lu Su

School of Electrical and Computer Engineering
Purdue University

Overview

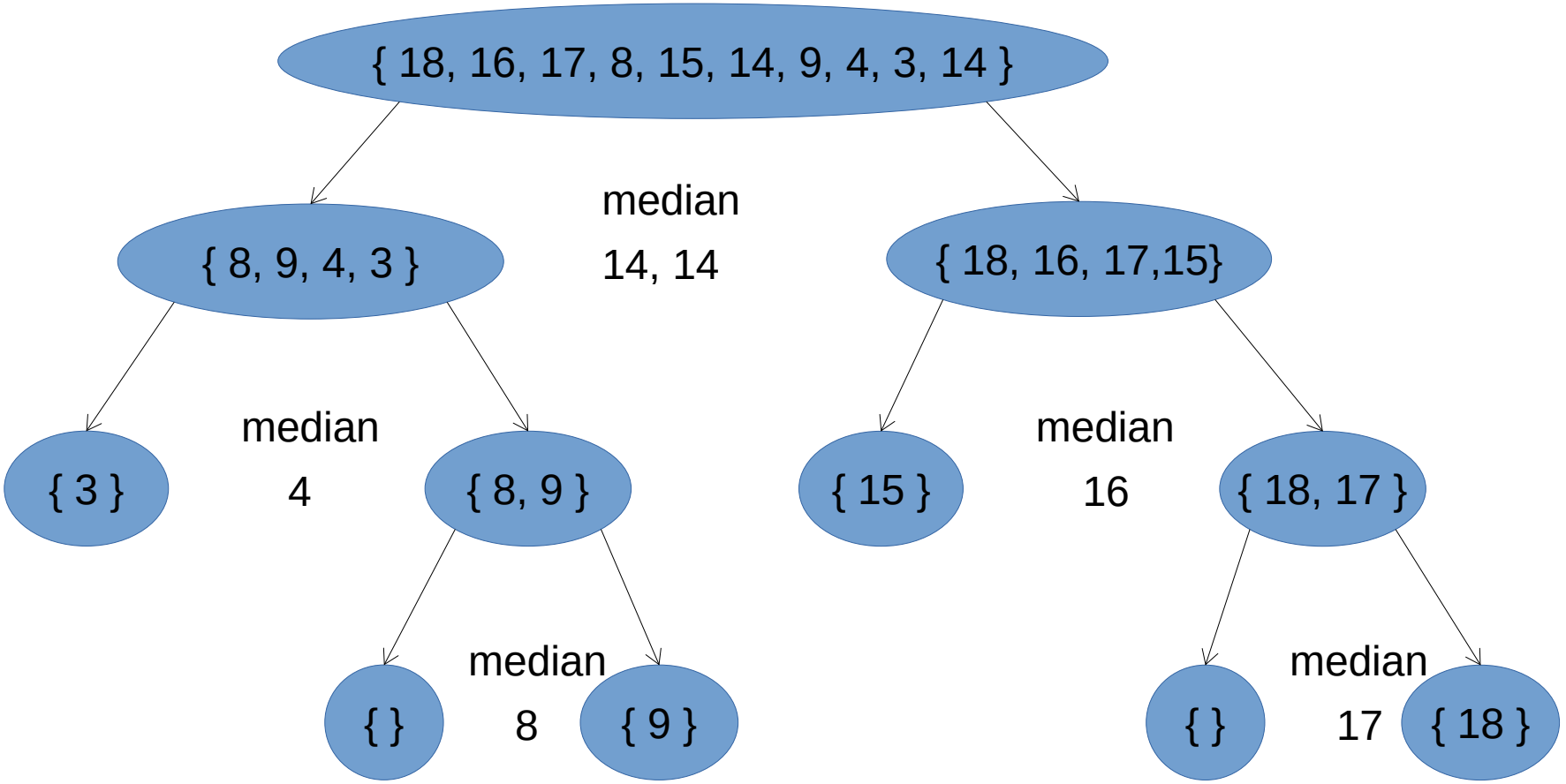
- Divide-and-conquer
- Algorithm
- Example
- Improvements
- Complexity

- Reference pages pp. 303-335

An “ideal” selection sort

- Simple selection sort always find the largest entry in a collection
  - In the next iteration, problem size reduces by 1
  - Sorting takes n – 1 iterations
- What if the algorithm selects the median?
  - If we use the median to partition (divide) the collection of n items into two sub-collections, each of size n/2
- The first sub-collection is less than the median and the second sub-collection is greater than the median
- We then can apply recursively the method (conquer) on the two sub-collections
  - After log n passes, we will be left with a collection with only 1 entry, which is by definition, sorted

An example


{ 18, 16, 17, 8, 15, 14, 9, 4, 3, 14 }
median
14, 14
{ 8, 9, 4, 3 }
{ 18, 16, 17,15}
median
4
median
16
{ 3 }
{ 8, 9 }
{ 15 }
{ 18, 17 }
median
8
median
17
{ }
{ 9 }
{ }
{ 18 }

Complexity analysis

- Assume that selection of median O(n) time complexity
  - Indeed, such an algorithm exists but it is beyond the scope of this course
  - The coefficient of the linear term is quite high, making it impractical
- Every level requires O(n) operations to find all the medians at that level
- There are log n levels
- Overall time complexity is O(n log n)

Quicksort

- Basic idea
  - Choose a pivot
  - Partition array into two subarrays with items less than or equal to pivot, and items greater than (and possibly equal to) pivot (divide)
  - Sort two subarrays recursively (conquer)
- Recursion: require call stack or manipulate a stack, not an in-place sorting algorithm
- Important for performance: good choice of pivot

Quicksort: algorithm

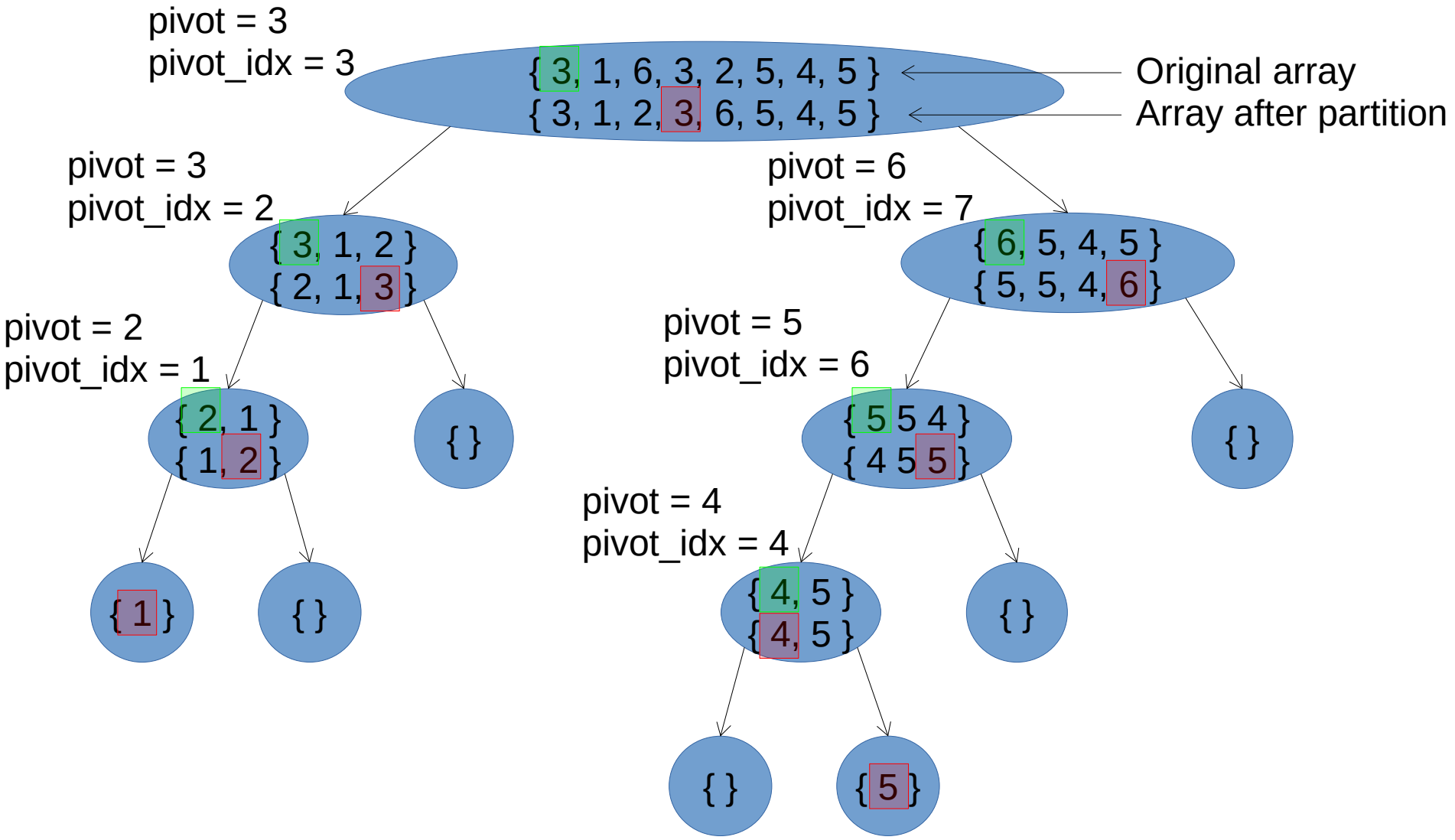
- Sort n integers r[0] to r[n-1] in ascending order
- Call qsort(r, 0, n – 1)

```
qsort (r[], lidx, ridx):
  if lidx ≥ ridx
    return;
  // divide
  pivot_idx = partition(r, lidx, ridx);
  // conquer left subarray
  qsort(r, lidx, pivot_idx-1);
  // conquer right subarray
  qsort(r, pivot_idx+1, ridx);
```

Partition(r[], lidx, ridx):

```
pivot ← r[lidx]
lo ← lidx
hi ← ridx
while lo < hi
  // find an item larger than pivot
  while r[lo] ≤ pivot and lo < ridx
    lo++
  // find an item not larger than pivot
  while r[hi] > pivot
    hi--;
  // swap out-of-order items
  if lo < hi
    r[lo] ↔ r[hi]
r[lidx] ← r[hi] // set pivot at the right place
r[hi] ← pivot // to partition array
return hi
```

Example


pivot = 3
pivot_idx = 3
{ 3, 1, 6, 3, 2, 5, 4, 5 }
{ 3, 1, 2, 3, 6, 5, 4, 5 }
Original array
Array after partition
pivot = 3
pivot_idx = 2
{ 3, 1, 2 }
{ 2, 1, 3 }
pivot = 6
pivot_idx = 7
{ 6, 5, 4, 5 }
{ 5, 5, 4, 6 }
pivot = 2
pivot_idx = 1
{ 2, 1 }
{ 1, 2 }
{ }
pivot = 5
pivot_idx = 6
{ 5 5 4 }
{ 4 5 5 }
{ }
pivot = 4
pivot_idx = 4
{ 1 }
{ }
{ 4, 5 }
{ 4, 5 }
{ }
{ }
{ 5 }

Notes on quicksort

- Not a stable sorting algorithm because of the partition function
- Recursion requires stack space
  - It is important the stack space is minimized
- Important that the recursive calls are on subarrays that have fewer elements than the original
  - The partition function puts the pivot at the correct position, the subarrays are therefore guaranteed to have fewer elements than the original
  - Other partition functions may not put the pivot at the correct position, must call qsort recursively with correct left and right indices to avoid infinite recursion

Complexity analysis

- Best case: Similar to the ideal case, O(n log n) for time complexity and O(log n) for space complexity
- Worst case: Alway chooses a lousy pivot
  - It takes n – 1 comparisons to partition into two subarrays of size 0 and n – 1 (pivot is placed at the correct position)
  - It then works on the larger subarray
  - Total number of comparisons is (n – 1) + (n – 2) + … + 1 = n(n – 1)/2 = $O(n^2)$
  - Time complexity is $O(n^2)$ and space complexity is O(n)

Average-case time complexity

- Let C(n) be the expected number of comparisons

$$C(n) = n - 1 + \frac{1}{n}\sum_{i=0}^{n-1}(C(i) + C(n-1-i)) = n - 1 + \frac{2}{n}\sum_{i=0}^{n-1} C(i)$$

$$nC(n) = n(n-1) + 2\sum_{i=0}^{n-1} C(i) = n(n-1) + 2C(n-1) + 2\sum_{i=0}^{n-2} C(i)$$

$$(n-1)C(n-1) = (n-1)(n-2) + 2\sum_{i=0}^{n-2} C(i)$$

$$nC(n) = (n+1)C(n-1) + 2(n-1)$$

$$\frac{C(n)}{n+1} = \frac{C(n-1)}{n} + \frac{2}{n+1} - \frac{2}{n(n+1)} \leq \frac{C(n-1)}{n} + \frac{2}{n+1}$$

Average-case time complexity

$$\frac{C(n)}{n+1} = \frac{C(n-1)}{n} + \frac{2}{n+1} - \frac{2}{n(n+1)} \leq \frac{C(n-1)}{n} + \frac{2}{n+1}$$

$$\frac{C(n-1)}{n} = \frac{C(n-2)}{n-1} + \frac{2}{n} - \frac{2}{(n-1)n} \leq \frac{C(n-2)}{n-1} + \frac{2}{n}$$

$$\begin{aligned}
\tfrac{C(n)}{n+1} &\leq \tfrac{C(n-2)}{n-1} + \tfrac{2}{n} + \tfrac{2}{n+1} \\
&\vdots \\
&\leq \tfrac{C(1)}{2} + 2\textstyle\sum_{i=2}^{n} \tfrac{1}{i+1} \\
&\leq 2\textstyle\sum_{i=1}^{n-1} \tfrac{1}{i} \\
&\approx 2\int_1^n \tfrac{1}{x}dx = 2\ln n \approx 1.39 \log n \\
C(n) &\leq 1.39(n+1)\log n = O(n \log n)
\end{aligned}$$

Reducing space complexity

```
qsort (r[], lidx, ridx):
  if lidx ≥ ridx
    return;
  pivot_idx = partition(r, lidx, ridx);
  qsort(r, lidx, pivot_idx-1);
  qsort(r, pivot_idx+1, ridx);
```

Make the tail recursion the recursion that involves the larger subarray so that we can convert it into an iteration

```
qsort (r[], lidx, ridx):
  if lidx ≥ ridx
    return;
  pivot_idx = partition(r, lidx, ridx);
  if pivot_idx < (lidx + ridx) / 2
    qsort(r, lidx, pivot_idx-1);
    qsort(r, pivot_idx+1, ridx);
  else
    qsort(r, pivot_idx+1, ridx);
    qsort(r, lidx, pivot_idx-1);
```

Tail recursion removal

```
qsort (r[], lidx, ridx):
  while (lidx < ridx)
    pivot_idx = partition(r, lidx, ridx);
    if pivot_idx < (lidx + ridx) / 2
      qsort(r, lidx, pivot_idx-1);
      lidx = pivot_idx+1;
    else
      qsort(r, pivot_idx+1, ridx);
      ridx = pivot_idx-1;
```

- Given an array of n elements, the recursive call involves at most n/2 elements
- The sizes of qsort function calls are at most n, n/2, n/4, …, 1
- Space complexity is at worst O(log n)

Pivot selection strategies

- Median as pivot (O(n) but not practical)
- Middle element instead of the first element or last element
  - If the array is already sorted, the middle element is the median
- Median-of-three: median of first, middle, and last elements
  - If you are already comparing them, might as well put them in the correct relative positions
  - After that, the non-median elements are also in the correct partitions
- Use the mean of an array as your pivot
  - Mean is not an element in the array
  - If you use integer type for your calculation of mean (assuming keys are integer), beware of overflow/underflow issue
  - If you use floating point (float, double, long double), be aware of truncation errors and that a float cannot represent all possible values of int, a double cannot represent all possible values of long
  - Floating point operations may also add overhead
- Use median-of-three for the first pivot, and means of subarrays as subsequent pivots
- Pick a random pivot: O(n log n) with high probability
  - The rand function (or other pseudo-random number generator) may add substantial overhead

Partitioning strategies

- The partition function covered does not handle an array that contains many repeated elements well
- Hoare's partition function divides an array of repeated elements evenly
  - From two ends of the array, the function searches inward for a pair of elements, one greater than or equal to the pivot on the left, one lesser or equal on the right, that are in the wrong order relative to each other and exchange the inverted pair
- Perform one pass of bi-directional bubble sort as we partition to move the largest item of left subarray and the smallest item of right subarray to the correct positions
  - Detect sortedness
  - A subarray of size 2 is sorted
  - To sort a subarray of size 3, require only one more comparison

An improved quicksort

- Simple insertion sort is faster for smaller arrays, use it when an array is small enough

```
quicksort (r[], lidx, ridx):
  qsort_tr(r, lidx, ridx);
  insertion_sort(r, lidx, ridx);

qsort_tr(r[], lidx, ridx):
  while ridx – lidx > smallsize
    pivot_idx = partition(r, lidx, ridx);
    if pivot_idx < (lidx+ridx)/2
      qsort_tr(r, lidx, pivot_idx-1);
      lidx = pivot_idx+1;
    else
      qsort_tr(r, pivot_idx+1, ridx);
      ridx = pivot_idx-1
```