Objectives for Thu 9/21/2023

- Analysis of sorting algorithms
- Merge sort
  - Algorithm
  - Analysis
- Quick sort
  - Algorithm
  - Analysis
- Comparing Merge Sort and Quick Sort
- Coming soon: Heap Sort
- Divide and conquer algorithms

Merge Sort

Algorithm – Merge Sort

- Divide the sequence[1] into two subsequences
- Merge-Sort each subsequence
- Merge the two sorted subsequences into one sorted sequence

[1] sequence could be an array, linked list, file, or other sequence of data.

merge(…)

- Merge sorted subarrays r[lidx..mid] and r[mid+1..ridx] in ascending order
- Assume auxiliary space tmp[0..n-1], where n is the number of elements in the array
- The function memcpy copies the contents at second parameter to the location at first parameter; the number of elements copied is specified by third parameter
  - Not exactly the function in string.h

```
merge (r[], lidx, mid, ridx):
  memcpy(&tmp[lidx], &r[lidx], mid-lidx+1);
  memcpy(&tmp[mid+1], &r[mid+1], ridx-mid);
  i ← lidx; j ← mid+1;
  for k ←  lidx to ridx
    if (i > mid) r[k] ← tmp[j++];
    else if (j > ridx) r[k] ← tmp[i++];
    else if (tmp[j] < tmp[i])
           r[k] ← tmp[j++];
    else r[k] ← tmp[i++];
```


mergesort(…)

```
mergesort (r[], lidx, ridx):
  if lidx ≥ ridx
    return;
  // divide
  mid = (lidx+ridx)/2; // assume no overflow
  // conquer left subarray
  mergesort(r, lidx, mid);
  // conquer right subarray
  mergesort(r, mid+1, ridx);
  // merge subarrays
  merge(r, lidx, mid, ridx)
```

Properties

- Time complexity
  - Worst case: O(n log n)
  - Best case: Ω(n log n)
  - Average case: Θ(n log n)

- Space complexity: O(n)

Improvements

- By alternating the role of the output array and the auxiliary buffer (and proper initialization), we can avoid the memcpy operations in merge function

```
// store sorted left subarray in tmp using r as aux
mergesort(tmp, r, lidx, mid);
// store sorted right subarray in tmp using r as aux
mergesort(tmp, r, mid+1, ridx);
// merge sorted subarrays in tmp into r
merge(r, tmp, lidx, mid, ridx)
```

- In-place mergesort by Katajainen, Pasanen, and Tehuola, 1996
  - High overhead to be practical

Iterative version

- Bottom-up process
- Leaf nodes: Consider original array as n subarrays, each of size 1
- Scan through array performing n/2 times, merging of two 1-element arrays to produce n/2 sorted subarrays, each of size 2
- Scan through array performing n/4 times, merging of two 2-element arrays to produce n/4 sorted subarrays, each of size 4
- …
- Perform merging of two n/2-element arrays to produce the final sorted array of n elements

Comparing sorting algorithms

Algorithms	Best	Average	Worst
Insertion	O(n)	O(n^2)	O(n^2)
Bubble	O(n)	O(n^2)	O(n^2)
Quick	O(n log n)	O(n log n)	O(n^2)
Simple selection	O(n^2)	O(n^2)	O(n^2)
Heap	O(n)	O(n log n)	O(n log n)
Merge	O(n log n)	O(n log n)	O(n log n)

$O(n^2)$ vs. $O(n \log n)$

- How can mergesort, quicksort, and heapsort achieve $O(n \log n)$ when insertion sort and bubble sort run at $O(n^2)$?
  - Mergesort, quicksort, and heapsort move elements far distances, correcting multiple inversions (incorrect ordering) at a time
  - Insertion and bubble sort correct one inversion at a time
- Why is quicksort worst-case $O(n^2)$ while mergesort has no such problem?
  - The choice of pivot determines size of partitions, whereas mergesort cuts array in half every iteration
  - Simple selection sort uses the worst pivot in every iteration
- Is $O(n \log n)$ the best we can do?
  - Yes, if we use binary comparison on the key values

Quick Sort

quicksort(…)

- 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);
```

Credit: Andrew Jones & Cheng-Kok Koh

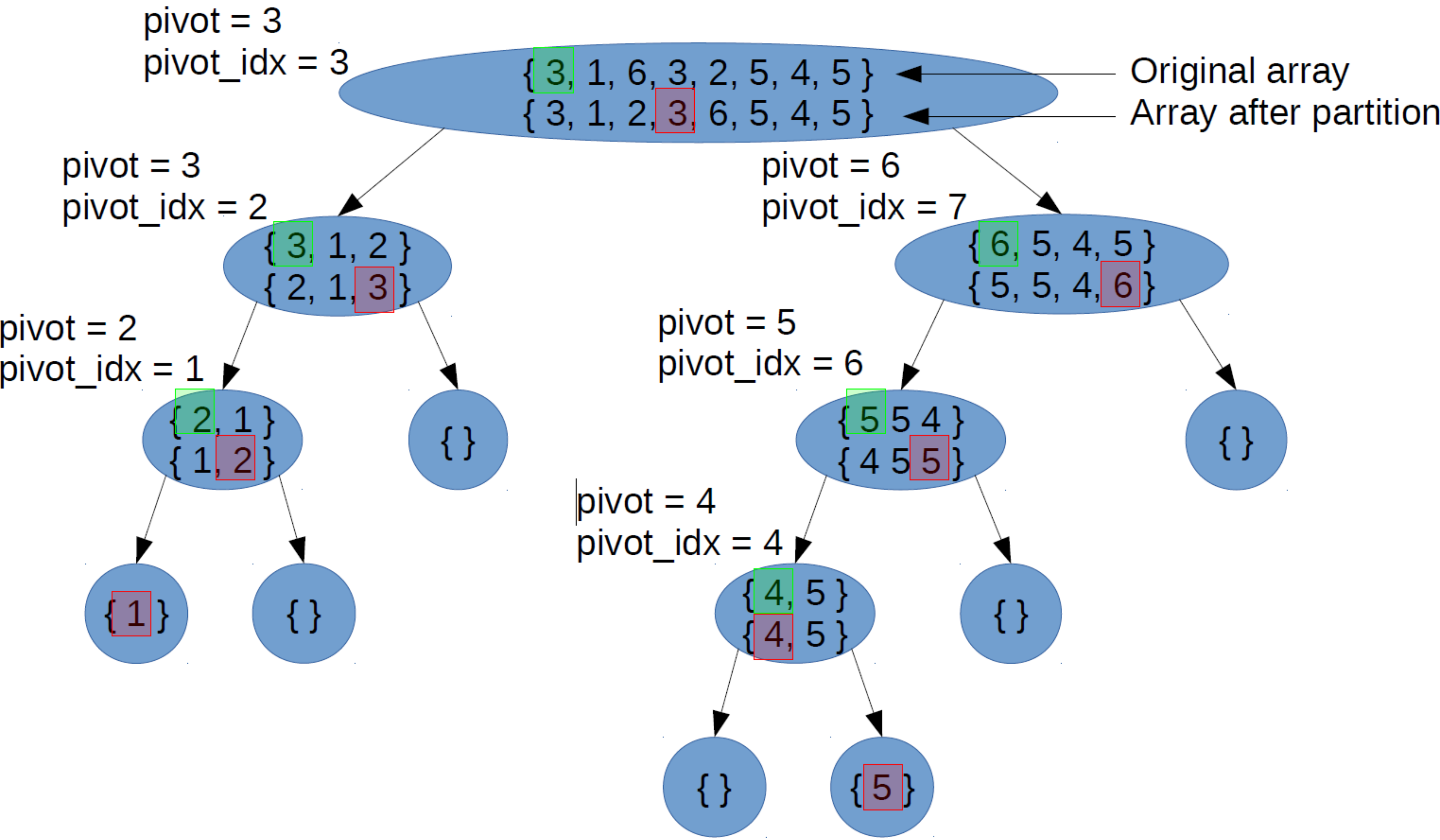
Complexity

- 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)

Credit: Andrew Jones & Cheng-Kok Koh

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
```


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

- 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

- 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 $(n - 1) + (n - 2) + \ldots + 1 = n(n - 1)/2 = O(n^2)$
  - Time complexity is $O(n^2)$ and space complexity is $O(n)$

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