Lecture10: Sorting II

Bohyung Han
CSE, POSTECH
bhhan@postech.ac.kr

Sorting

• Important operation when organizing data
  ▪ Ordering of elements
  ▪ Finding duplicate elements
  ▪ Ranking elements (i.e., nth largest)
  ▪ etc.

• An example
  
  input 3 6 1 9 4 2  
  output 1 2 3 4 6 9

• There are many different strategies and we need to care about
  ▪ Correctness
  ▪ Efficiency (in terms of time complexity)

Sorting Algorithms

• Bubble sort
• Selection sort
• Insertion sort

• Merge sort
• Quick sort

Performance of Sorting Algorithms

How about the performance of merge sort and quicksort?
Divide-and-Conquer

- A strategy to solve problems
  - Break problem into smaller problems
  - Solve the smaller problems
  - Combine results
- Strategy can be applied “recursively” to smaller problem
  - Continue dividing problem until solution is trivial.
  - Divide-and-conquer can be implemented by recursion effectively.
- Recursive sorting
  - Apply divide-and-conquer to sorting problem
  - How can we solve sorting recursively?

Merge Sort

- A sorting algorithm based on the divide-and-conquer paradigm
- Divide
  - Partition $S$ with $n$ elements into two sequences $S_1$ and $S_2$ of about $n/2$ elements each.
  - Recursion
    - Recursively sort $S_1$ and $S_2$
    - Performed implicitly by calling itself
  - Conquer
    - Merge $S_1$ and $S_2$ into a unique sorted sequence
- Conquer
  - Actual sorting is performed inside this function.
- Time complexity
  - Can be implemented with doubly linked list efficiently
  - $O(n)$

Merge Sort

Algorithm `mergeSort(S, C)`

- Input: sequence $S$ with $n$ elements, comparator $C$
- Output: sequence $S$ sorted according to $C$

```
if S.size() > 1
    (S1, S2) ← partition(S, n/2)
    mergeSort(S1, C)
    mergeSort(S2, C)
    S ← merge(S1, S2)
return
```

Merge Sort

Algorithm `merge(A, B)`

- Input: sequences $A$ and $B$ with $n/2$ elements each
- Output: sorted sequence of $A \cup B$

```
S ← empty sequence
while ¬A.isEmpty() ∧ ¬B.isEmpty()
    if A.first().element() < B.first().element()
        S.addLast(A.remove(A.first()))
    else
        S.addLast(B.remove(B.first()))
while ¬A.isEmpty()
    S.addLast(A.remove(A.first()))
while ¬B.isEmpty()
    S.addLast(B.remove(B.first()))
return S
```
Merge Sort Tree

- An execution of merge sort is depicted by a binary tree.
  - Each node represents a recursive call of merge-sort and stores
    - unsorted sequence before the execution and its partition
    - sorted sequence at the end of the execution
  - The root is the initial call
  - The leaves are calls on subsequences of size 0 or 1

An Execution Example

Merge Sort Algorithm

```java
public void merge_sort(int[] values)
{
    // check for termination condition
    if (values == null || values.length <= 1)
        return;

    // divide array into two arrays of half size
    int middle = values.length/2;
    int[] left = new int[middle];
    int[] right = new int[values.length-middle];

    for (int i=0; i<middle; i++)
        left[i] = values[i];
    for (int i=0; i<values.length-middle; i++)
        right[i] = values[middle+i];

    // recursively call sorting function on each smaller array
    merge_sort(left);
    merge_sort(right);

    // combine sorted arrays
    int l=0;
    int r=0;
    for (int i=0; i<values.length; i++)
    {
        if (r>=right.length || (l<left.length && left[l]<right[r]))
            values[i] = left[l++];
        else
            values[i] = right[r++];
    }
    return;
}
```
Analysis of Merge Sort

- The height $h$ of the merge-sort tree is $O(\log n)$
  - At each recursive call we divide in half the sequence.
- The amount of work done at the nodes of depth $i$ is $O(n)$.
  - We partition and merge $2^i$ sequences of size $n/2^i$.
  - We make $2^{i+1}$ recursive calls.
- The total running time of merge sort is $O(n \log n)$.

<table>
<thead>
<tr>
<th>depth</th>
<th>#seqs</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>$n$</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>$n/2$</td>
</tr>
<tr>
<td>$i$</td>
<td>$2^i$</td>
<td>$n/2^i$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Quicksort

- A divide and conquer algorithm.
- The steps are:
  - Pick an element, called a pivot, from the list.
  - Partitioning: Reorder the list so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it.
    - Equal values can go either way.
    - After this partitioning, the pivot is in its final position.
  - Recursively apply the above steps to the sub-lists of elements with smaller values and separately the sub-list of elements with greater values.
- The base case of the recursion are lists of size zero or one, which never need to be sorted.

Quicksort

- Divide
  - Pick a random element $x$ (called pivot) and partition $S$ into
    - $L$ elements less than $x$
    - $E$ elements equal $x$
    - $G$ elements greater than $x$
- Recurrence
  - Sort $L$ and $G$
- Conquer
  - Join $L$, $E$ and $G$

Quicksort

- Pseudo-code of quicksort

```
Algorithm quicksort(S, p)
Input sequence S, position p of pivot
Output sorted sequence S

L, E, G ← partition(S, p)
quicksort(L, p)
quicksort(G, p)
S ← join(L, E, G)
return
```
Partition

- Partition an input sequence
  - Remove each element $y$ from $S$.
  - Insert $y$ into $L$, $E$ or $G$, depending on the result of the comparison with the pivot $x$.
- Time complexity
  - Insertions and removals happen at the beginning or at the end of a sequence, and hence takes $O(1)$ time.
  - The partition step of quicksort takes $O(n)$ time.

Algorithm $\text{partition}(S, p)$

```plaintext
Input: sequence $S$, position $p$ of pivot
Output: subsequences $L$, $E$, $G$ of the elements of $S$ less than, equal to, or greater than the pivot, resp.

$L$, $E$, $G \leftarrow$ empty sequences
$x \leftarrow S.\text{remove}(p)$
while $\neg S.\text{isEmpty}()$
  $y \leftarrow S.\text{remove}(S.\text{first}())$
  if $y < x$
    $L.\text{addLast}(y)$
  else if $y = x$
    $E.\text{addLast}(y)$
  else { $y > x$ }
    $G.\text{addLast}(y)$
return $L$, $E$, $G$
```

Quicksort Tree

- An execution of quicksort is depicted by a binary tree
  - Each node represents a recursive call of quicksort and stores
    - Unsorted sequence before the execution and its pivot
    - Sorted sequence at the end of the execution
  - The root is the initial call
  - The leaves are calls on subsequences of size 0 or 1

In-Place Quicksort

- Quicksort can be implemented to run in-place.
  - In the partition step, we use replace operations to rearrange the elements.
  - The recursive calls consider
    - elements with rank less than $h$
    - elements with rank greater than $k$

Algorithm $\text{inPlaceQuickSort}(S, l, r)$

```plaintext
Input: sequence $S$, ranks $l$ and $r$
Output: sequence $S$ with the elements of rank between $l$ and $r$ rearranged in increasing order

if $l \geq r$
  return
$i \leftarrow a$ random integer between $l$ and $r$
$x \leftarrow S.\text{elemAtRank}(i)$
$(h, k) \leftarrow \text{inPlacePartition}(x)$
$\text{inPlaceQuickSort}(S, l, h - 1)$
$\text{inPlaceQuickSort}(S, k + 1, r)$
```
In-Place Quick-Sort

- Perform the partition using two indices to split $S$ into $L$, $E$, $G$

  $j \quad 3 \ 2 \ 5 \ 1 \ 0 \ 7 \ 3 \ 5 \ 9 \ 2 \ 7 \ 9 \ 8 \ 9 \ 7 \ 6 \ 9$ (pivot = 6)

- Repeat until $j$ and $k$ cross:
  - Scan $j$ to the right until finding an element greater than or equal to pivot or $j = k$.
  - Scan $k$ to the left until finding an element less than pivot or $j = k$.
  - Swap elements at indices $j$ and $k$ or swap pivot with $j$ when $j = k$.

\[ \[ 

In-Place Quicksort

```java
public void quickSort(int[] values, int left, int right) {
    if (right <= left)
        return;
    int pivotIndex = partition(values, left, right);
    quickSort(values, left, pivotIndex - 1);
    quickSort(values, pivotIndex + 1, right);
}
```

Analysis of Quicksort

- Depends on the choice of pivot element
  - Good choice: median value of array
  - Bad choice: max or min value of array
  - In practice, it is difficult to choose a good one without search.

- Time complexity
  - Worst case: $O(n^2)$
  - Average case: $O(n \log n)$
  - Randomized algorithm: $O(n \log n)$

```java
public int partition(int[] values, int leftBound, int rightBound) {
    int pivot = values[rightBound];
    int left = leftBound - 1;
    int right = rightBound; // rightmost is pivot
    while (true) {
        while (values[++left] < pivot) {
            ;
        }
        while (right > leftBound && values[--right] > pivot) {
            ;
        }
        if (left >= right)
            break;
        else
            swap(values, left, right); // swap misplaced items
    }
    swap(values, left, rightBound); // swap left and pivot
    return left; // return index of pivot
}
```
## Summary of Sorting Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time complexity</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bubble sort</td>
<td>$O(n^2)$</td>
<td>• in-place</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• slow (good for small inputs)</td>
</tr>
<tr>
<td>Selection sort</td>
<td>$O(n^2)$</td>
<td>• in-place</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• slow (good for small inputs)</td>
</tr>
<tr>
<td>Insertion sort</td>
<td>$O(n^2)$</td>
<td>• in-place</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• slow (good for small inputs)</td>
</tr>
<tr>
<td>Quicksort</td>
<td>$O(n \log n)$ expected</td>
<td>• in-place, randomized</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• fastest (good for large inputs)</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>$O(n \log n)$</td>
<td>• in-place</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• fast (good for large inputs)</td>
</tr>
</tbody>
</table>