Lecture 19

Sorting

7 2 | 9 4 \rightarrow 2 4 7 9

7 | 2 \rightarrow 2 7

7 \rightarrow 7

2 \rightarrow 2

9 | 4 \rightarrow 4 9

9 \rightarrow 9

4 \rightarrow 4
Review 3 simple sorting algorithms:
1. selection Sort (in previous course)
2. insertion Sort (in previous course)
3. heap-Sort (mentioned it earlier in the course)

“Divide and Conquer” algorithm design
- applies not just to sorting

Sorting algorithms today using divide and conquer
- Merge Sort
- Quick Sort
  - you should have had Quick Sort it in a previous course
General Comments on Sorting

- Given a sequence $S$ of elements to sort
  - for most general implementation, use comparator $C$, can reuse same code for non-decreasing and non-increasing order sorting

- **in-place vs. not in-place** sorting:
  - **in-place sorting**: need only a constant amount of additional space. This means sorting done in the original sequence $S$ itself
  - **not in-place sorting**: need more than a constant amount of additional space. For example need another sequence of size $S.size()$
  - use in-place algorithm if space is an issue

- **recursive vs. non-recursive**:
  - non-recursive and recursive algorithms may have the same asymptotic (big-O) complexity, but in practice, non-recursive code is usually faster and should be preferred
  - recursive code is easier to write and understand, with practice 😊
Heap Sort

- array $A$ of size $n$, indexed from 0 to $n - 1$
- Need to sort in non-decreasing order

$$H = \text{new empty min-heap}$$

for $i = 0$ to $n - 1$

$$H.insert(A[i])$$

for $i = 0$ to $n - 1$

$$A[i] = H.deleteMin()$$

- For non-increasing order, use max-heap
- Complexity: $O(n \log n)$
- Need $O(n)$ additional space for the heap data structure
- can implement heap-sort in-place by reusing the input array $A$ for the heap data structure
Selection Sort

- array $A$ of size $n$, indexes range from 0 to $n - 1$
- Iterate $n$ times:
  - find the $i$th smallest element in the array $A$
  - insert this element in the $i$th location of the array $A$
  - after $i$th iteration, elements in range 0...$i - 1$ of $A$ in correct position
Algorithm SelectSort(A,n)
Input: Array A and its size n
Output: Sorts elements of A in non-decreasing order

for $i = 0$ to $n - 2$
    // First find $i$th smallest element
    $minIndex = i$
    for $j = i + 1$ to $n-1$
        if $A[j] < A[minIndex]$ then
            $minIndex = j$
    // now swap the smallest element with $i$th element
    $temp = A[minIndex]$
    $A[minIndex] = A[i]$
    $A[i] = temp$
Selection Sort Complexity

- Outer `for` loop is performed \( n-1 \) times
- Inner `for` loop is performed \( n - 1 - i \) times for a fixed \( i \)
  - total time is
    \[
    \sum_{i=0}^{n-2} (n-1-i) = \sum_{i=1}^{n-1} i = \frac{(n-2)n}{2} - 1 = O(n^2)
    \]
  - \( \text{for } i = 0 \text{ to } n - 2 \)
    - \( \text{minIndex} = i \)
    - \( \text{for } j = i + 1 \text{ to } n-1 \)
      - if \( A[j] < A[\text{minIndex}] \) then
        - \( \text{minIndex} = j \)
      - \( \text{temp} = A[\text{minIndex}] \)
      - \( A[\text{minIndex}] = A[i] \)
      - \( A[i] = \text{temp} \)
- All other operations inside the `for` loops take constant amount of time. Thus total running time is \( O(n^2) \)
- Running time is independent of the contents of array \( A \), that is it is the best, worst, and average case running time
Insertion Sort

- after iteration $i$, array $A$ should be sorted in the range $0...i$
- but elements in range $0...i$ are not necessarily in their final correct positions after iteration $i$
Algorithm InsertSort(A,n)
Output: Sorts elements of A in non-decreasing order
for i = 1 to n - 1
    // first find correct position for element A[i] so that subarray A[0... i] stays sorted
    x = A[i]
    j = i - 1
    while j ≥ 0 and A[j] > x do
        j = j - 1

    // the correct position for x is j+1, put x in that position
    A[j + 1] = x
Insertion Sort Complexity

- **for** loop is performed $n - 1$ times
- **while** loop is performed, for each fixed $i$,
  - $i$ times in the worst case
  - 0 times in the best case, when sub-array $A[0...i]$ is already sorted
- All other statements take constant amount of time
- In the best case, insertion sort is $O(n)$
- In the worst case, insertion sort is $O(n^2)$

```plaintext
for i = 1 to n - 1
    x = A[i]
    j = i - 1
    while j ≥ 0 and A[j] > x do
        j = j - 1
    A[j + 1] = x
```

$$
\sum_{i=1}^{n-1} i = \frac{(n-1)n}{2}
$$
**Divide-and-Conquer Approach**

- **Divide-and-conquer** is a general algorithm design paradigm.
- Suppose need to solve some problem on a sequence $S$.
  - Example: sorting.
- Suppose solving the problem on $S$ is hard, but if we have a solution on subsequences $S_1$ and $S_2$ of $S$, then combining these solutions into solution on $S$ is easy.
  - **Divide**: divide the input data $S$ in two disjoint subsets $S_1$ and $S_2$.
  - **Recur**: solve the subproblems associated with $S_1$ and $S_2$.
  - **Conquer**: combine the solutions for $S_1$ and $S_2$ into solution for $S$.
- Base case for recursion are subproblems of size 0 or 1.
  - Base case is usually trivial to solve.
Divide-and-Conquer Example

- **sequence** $S$ to sort:

- **Divide**: split $S$ into subsequences $S_1$ and $S_2$

- **Recur**: sort subsequences $S_1$ and $S_2$

- **Conquer**: merge sorted $S_1$ and $S_2$ into sorted $S$

- **Base case**: sorting a sequence of size 1 is trivial
Divide-and-Conquer

- **Merge-sort** is a sorting algorithm based on the divide-and-conquer paradigm
- Like heap-sort it has $O(n \log n)$ running time
- Unlike heap-sort
  - does not use an auxiliary priority queue
  - accesses data sequentially, so it is suitable to sort data on a disk
Merge-Sort

- Merge-sort on an input sequence $S$ with $n$ elements has three steps:
  - **Divide**: partition $S$ into two sequences $S_1$ and $S_2$ of about $n/2$ elements each
  - **Recur**: recursively sort $S_1$ and $S_2$
  - **Conquer**: merge $S_1$ and $S_2$ into single sorted sequence

**Algorithm** $mergeSort(S, C)$

**Input** sequence $S$ with $n$ elements, comparator $C$

**Output** sequence $S$ sorted according to $C$

if $S.size() > 1$

$(S_1, S_2) \leftarrow partition(S, n/2)$

$mergeSort(S_1, C)$

$mergeSort(S_2, C)$

$S \leftarrow merge(S_1, S_2)$
Merging Two Sorted Sequences

- Conquer step: merge two sorted sequences $S_1$ and $S_2$ into a sorted sequence $S$
- Assume linked list implementation of a sequence
Merging Two Sorted Sequences

- merging two sorted sequences, each with \( n/2 \) elements and implemented with a doubly linked list is \( O(n) \)
- array implementation is \( O(n) \)

Algorithm \( merge(A, B) \)

**Input** sequences \( A \) and \( B \) with \( n/2 \) elements each

**Output** sorted sequence of \( A \cup B \)

\[
S \leftarrow\text{empty sequence}
\]

while \( \neg A.isEmpty() \land \neg B.isEmpty() \)

if \( A.first().element() < B.first().element() \)

\[
S.insertLast(A.removeFirst())
\]

else

\[
S.insertLast(B.removeFirst())
\]

while \( \neg A.isEmpty() \)

\[
S.insertLast(A.removeFirst())
\]

while \( \neg B.isEmpty() \)

\[
S.insertLast(B.removeFirst())
\]

return \( S \)
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 (after →).
- The root is the initial call.
- The leaves are calls on subsequences of size 0 or 1.
Execution Example

- **Partition**

```
7  2  9  4
```

```
3  8  6  1
```

```
7  2
```

```
3  8
```

```
3  8  6  1
```

```
1  6
```

```
1  2  3  4  6  7  8  9
```
Execution Example (cont.)

- Recursive call, partition

```
7 2 9 4 | 3 8 6 1
```

```
<table>
<thead>
<tr>
<th>7 2</th>
<th>9 4</th>
</tr>
</thead>
</table>
```

```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```

```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
Execution Example (cont.)

- Recursive call, partition

```
| 7 2 9 4 | 3 8 6 1 |
```

```
| 7 2 | 9 4 |
```

```
| 7 | 2 |
```

```
```

```
```

```
```

```
```

```
```

Execution Example (cont.)

- Recursive call, base case

```
    7 2 9 4 | 3 8 6 1
```

```
    7 2 | 9 4
```

```
    7 | 2
```

```
    7 → 7
```
Execution Example (cont.)

- Recursive call, base case

```
7 2 9 4 | 3 8 6 1
```

```
7 2 9 4
```

```
7 2 | 9 4
```

```
7 2
```

```
7 \rightarrow 7
```

```
2 \rightarrow 2
```

Execution Example (cont.)

- Merge

```
7 2 9 4 | 3 8 6 1
7 2 | 9 4
7 | 2 → 2 7
7 → 7 2 → 2
```

```
Execution Example (cont.)

- Recursive call, ..., base case, merge
Execution Example (cont.)

- Merge

```
7 2 9 4 3 8 6 1
```

```
7 2 9 4 2 4 7 9
```

```
7 2 2 7
9 4 4 9
```

```
7 7 2 2 9 9 4 4
```

```
7 7 2 2 9 9 4 4
```

```
7 9 4 2 4 9

7 9 4 4 2 9
```

```
7 7 2 9 4 4

7 7 2 9 4 4
```

```
7 7 2 7 7 2 9 4

7 7 2 9 4 4
```
Execution Example (cont.)

- Recursive call, ..., merge, merge
Execution Example (cont.)

- Merge

```
    7 2 9 4 | 3 8 6 1 \rightarrow 1 2 3 4 6 7 8 9
     /
 7 2 | 9 4 \rightarrow 2 4 7 9
     |
 7   | 2  \rightarrow 2 7
 9 4  \rightarrow 4 9
     |
 9   | 4  \rightarrow 9
 4   | 4  \rightarrow 4

    3 8 6 1 \rightarrow 1 3 6 8
     /
 3 8 \rightarrow 3 8
     |
 3   \rightarrow 3
 8   \rightarrow 8

    6 1 \rightarrow 1 6
     /
 6   \rightarrow 6
 1   \rightarrow 1
```
Non-Recursive Merge Sort

- Recursive implementation is less efficient (by a constant factor) than non-recursive implementation
- Merge-sort can be implemented non-recursively
  - at iteration $i$, break the sequence into groups of size $2^{i-1}$
    - groups of 1, then 2, then 4, ...
  - merge 2 nearby groups together

$i = 1$

\[
\begin{align*}
7 & 2 9 4 3 8 6 1 \\
7 & 2 9 4 3 8 6 1 \\
7 & 2 9 4 3 8 6 1 \\
\end{align*}
\]

\[
\begin{align*}
2 & 7 4 9 3 8 1 6 \\
2 & 7 4 9 3 8 1 6 \\
2 & 4 7 9 1 3 6 8 \\
1 & 2 3 4 6 7 8 9 \\
\end{align*}
\]
Analysis of Merge-Sort

- Recurrence equations
  
  \[
  T(1) = c \\
  T(n) = kn + 2T(n/2)
  \]

- Solve recurrence equations:
  
  \[
  T(n) = kn + 2T(n/2) = kn + 2[kn/2 + 2T(n/4)] = kn + kn + 4T(n/4) = 2kn + 4T(n/4) = 2kn + 4[kn/4 + 2T(n/8)] = 3kn + 8T(n/8) = \ldots = ikn + 2^iT(n/2^i)
  \]

  Unwrapping stops when \(n/2^i = 1\), i.e. when \(i = \log n\)

  Thus \(T(n) = (\log n) kn + 2^{\log n}T(1) = kn(\log n) + cn\)

  Thus running time is \(O(n \log n)\)

Algorithm `mergeSort(S, C)`

- if `S.size() > 1`
  
  \((S_1, S_2) \leftarrow \text{partition}(S, n/2)\)
  
  `mergeSort(S_1, C)`
  
  `mergeSort(S_2, C)`

  \(S \leftarrow \text{merge}(S_1, S_2)\)
Part 2: Quick-Sort
Quick-Sort

- Randomized sorting algorithm based on the divide-and-conquer paradigm
  - 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 $x$
  - Recur: sort $L$ and $G$
  - Conquer: join $L$, $E$ and $G$
Partition

- To partition, iterate
  - remove each element \( y \) from \( S \)
  - insert \( y \) into \( L, E \) or \( G \), depending on result of the comparison with pivot \( x \)
- Each insertion and removal is at the beginning or end of sequence, and hence takes \( O(1) \)
- Thus, partition step is \( O(n) \)

Algorithm partition\((S, p)\)

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
\( x \leftarrow S\.elementAtPosition(p) \)
while \( \neg S\.isEmpty() \)

\( y \leftarrow S\.remove(S\.first()) \)
if \( y < x \)
   \( L\.insertLast(y) \)
else if \( y = x \)
   \( E\.insertLast(y) \)
else // \( y > x \)
   \( G\.insertLast(y) \)
return \( L, E, G \)
Quick-Sort Tree

- An execution of quick-sort is depicted by a binary tree
  - Each node represents a recursive call of quick-sort 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

```
7 4 9 6 2 → 2 4 6 7 9
```

```
4 2 → 2 4
```

```
7 9 → 7 9
```

```
2 → 2
```

```
9 → 9
```
Execution Example

- Pivot selection

![Diagram showing pivot selection process]
Execution Example

- Partition, recursive call, pivot selection
Execution Example

- Partition, recursive call, base case

```
7 2 9 4 3 7 6 1
```
```
2 4 3 1
```
```
1 → 1
```

1 → 1
Execution Example

- Recursive call, pivot selection

![Recursive call diagram](image-url)
Execution Example

- Recursive call, ..., base case, join
Execution Example

- Recursive call, pivot selection
Execution Example

- Partition, ..., recursive call, base case
Execution Example

- Join, join

```
7 2 9 4 3 7 6 1 → 1 2 3 4 6 7 7 9
```

```
2 4 3 1 → 1 2 3 4
```

```
7 9 7 → 7 7 9
```

```
1 → 1
```

```
4 3 → 3 4
```

```
4 → 4
```

```
9 → 9
```
Worst-case Running Time

- Happens if pivot is the unique minimum or maximum element
- One of $L$ and $G$ has size $n - 1$ and the other has size 0
- The running time is proportional to the sum
  \[n + (n - 1) + \ldots + 2 + 1\]
- Worst-case time is $O(n^2)$
Expected Running Time

- Consider a recursive call on a sequence of size $s$
  - **good call**: the sizes of $L$ and $G$ are each less than $\frac{3}{4} s$
  - **bad call**: one of $L$ and $G$ has size greater than $\frac{3}{4} s$

- A call is **good** with probability $\frac{1}{2}$, since half of elements are good pivots
Expected Running Time

- Each good pivot reduces sequence size by at least \( \frac{3}{4} \)
- For a node of recursion depth \( i \), we expect
  - \( i/2 \) ancestors are good calls
  - The size of the input sequence for the current call is at most \( (\frac{3}{4})^{i/2}n \)
- Solving \( (\frac{3}{4})^{i/2}n = 1 \), for \( i \)
  - for a node of depth \( i = 2\log_{4/3}n \), the expected input size is one
  - the expected height of the quick-sort tree is \( O(\log n) \)
- The amount or work done at the nodes of the same depth is \( O(n) \)
- Thus, the expected running time of quick-sort is \( O(n \log n) \)
In-Place Quick-Sort

- Can implement quick-sort in-place
- In partition step, rearrange elements of input sequence s.t.
  - the elements less than pivot have rank less than \( j \)
  - the elements equal to pivot have rank between \( j \) and \( k \)
  - the elements greater than pivot have rank greater than \( k \)
- Recursive calls consider
  - elements with rank less than \( j \)
  - elements with rank greater than \( k \)

**Algorithm inPlaceQuickSort(S, l, r)**

**Input** sequence \( S \), ranks \( l \) and \( r \)

**Output** sequence \( S \) with the elements of rank between \( l \) and \( r \) rearranged in increasing order

1. if \( l \geq r \) //base case
   return
2. \( i \leftarrow \) random integer between \( l \) and \( r \)
3. \( x \leftarrow S.elemAtRank(i) \)
4. \((j, k) \leftarrow \) inPlacePartition\((x, l, r)\)
5. \( \) inPlaceQuickSort\((S, l, j - 1)\)
6. \( \) inPlaceQuickSort\((S, k + 1, r)\)
In-Place Partitioning \((x, l, r)\)

- First partition \(S\) (between ranks \(l\) and \(r\)) into \(L\) \(< x\) and \(EG\) \(\geq x\)

\[
\begin{array}{ccccccccccccc}
3 & 2 & 5 & 1 & 0 & 7 & 3 & 5 & 9 & 2 & 7 & 9 & 8 & 9 & 6 & 6 \\
\end{array}
\]

- Repeat until \(j\) and \(k\) cross
  - scan \(j\) to the right until find element \(\geq x\)
  - scan \(k\) to the left until find element \(< x\)
  - swap elements at ranks \(j\) and \(k\)
In-Place Partitioning \((x, l, r)\)

- Next partition \(EG\) into \(E\) and \(G\)

  \[
  \begin{array}{cccccccccc}
  3 & 2 & 5 & 1 & 0 & 2 & 3 & 5 & 9 & 7 & 7 & 9 & 8 & 9 & 6 & 9
  \end{array}
  \]

  \(j\) \hspace{1cm} \(k\)

  \(l\) \hspace{1cm} \(r\)

- Repeat until \(j\) and \(k\) cross
  - scan \(j\) to the right until find element \(> x\)
  - scan \(k\) to the left until find element \(= x\)
  - swap elements at indices \(j\) and \(k\)
### Summary of Sorting Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>selection-sort</td>
<td>$O(n^2)$</td>
<td>in-place, slow (good for small inputs)</td>
</tr>
<tr>
<td>insertion-sort</td>
<td>$O(n^2)$</td>
<td>in-place, slow (good for small inputs)</td>
</tr>
<tr>
<td>quick-sort</td>
<td>$O(n \log n)$ expected</td>
<td>in-place, randomized, fastest in practice (good for large inputs)</td>
</tr>
<tr>
<td>heap-sort</td>
<td>$O(n \log n)$</td>
<td>in-place, fast (good for large inputs)</td>
</tr>
<tr>
<td>merge-sort</td>
<td>$O(n \log n)$</td>
<td>sequential data access, fast (good for huge inputs when data must be stored on a disk)</td>
</tr>
</tbody>
</table>