Sorting (2)

- Sort examples
- Merge sort
- Complexity, Big O notation
**Selection Sort**

Decompose:

```
  5 1 3 9 8 2
  1 5 3 9 8 2
  2 5 3 9 8
  3 5 9 8
  5 8 9
  8 9
```

Compose:

```
  8 9
  5 8 9
  3 5 8 9
  2 3 5 8 9
  1 2 3 5 8 9
```

Appending 2 queues, one in front of the other.
How to achieve queue append in unit cost?

- Each queue $q_1$, $q_2$ has a front and a back
- Requires knowledge of the representation of Queue objects
- Use the following code:
  ```java
  Queue qresult = new Queue();
  qresult.front = q1.front;
  (q1.back).subList = (q2.front).subList;
  qresult.back = q2.back;
  ```
Append in unit cost

\[
\begin{array}{c|c|c}
\text{q1.front} & 8 & \lambda \\
\hline
\text{q1.back} & ? & 10 \\
\end{array}
\quad
\begin{array}{c|c|c}
\text{q2.front} & 9 & \lambda \\
\hline
\text{q2.back} & ? & 10 \\
\end{array}
\]

\[
\begin{array}{c|c|c}
\text{qresult.front} & 8 \\
\hline
\text{qresult.back} & 9 & \lambda \\
\end{array}
\]
Quick Sort

Decompose:

Compose: (append)
Merge Sort

• Similar to selection and quick sorts
• Selection and quick use data value comparison; merge just splits the queue

```java
public SortProblem mergeSort() throws QueueException {
    if (getLength() == 1) return this;
    SortProblem sp1 = new SortProblem(),
    sp2 = new SortProblem();
    SortProblem sp1sorted, sp2sorted;
    inHalf(sp1,sp2);
    sp1sorted = sp1.mergeSort();
    sp2sorted = sp2.mergeSort();
    SortProblem ret = sp1sorted.merge(sp2sorted);
    return ret;
}
```
Decompose: halving the queue

private void inHalf(SortProblem halfA,
                     SortProblem halfB) throws QueueException {
    // halfA, halfB are input as empty Queues
    while (true) {
        // alternate elements from this into each
        // of halfA, halfB
        if (this.empty()) return;
        halfA.enter(this.remove());
        if (this.empty()) return;
        halfB.enter(this.remove());
    }
}
Compose

• Former append doesn’t work because need to interleave elements from both subproblems
• Must merge two ordered queues
  – First item in each queue is smallest
  – Smallest of two first items is smallest in both
• Always compare first item in each queue
  – Remove smaller and place on new queue
  – Repeat until one queue becomes empty
  – Append what remains
private SortProblem merge(SortProblem other) throws QueueException {
    SortProblem ret = new SortProblem();
    while ( !(this.empty()) && !(other.empty()) ) {
        Sortable vThis = (Sortable) this.peek(),
                  vOther = (Sortable) other.peek();
        if (vThis.lessThan(vOther))
            ret.enter(this.remove());
        else ret.enter(other.remove());
    }
    if (this.empty())
        while ( !(other.empty()) )
            ret.enter(other.remove());
    else while ( !(this.empty()) )
        ret.enter(this.remove());
    return ret;
}
Merge Sort

Decompose:

5 1 3 9 8 2

5 3 8          1 9 2

5 8       3

1 2     9

5      8

1      2

Compose: (merge)

5 8 1 2

3 5 8

1 2 9

1 2 3 5 8 9
Worst Case Cost - mergeSort()

- Costs for a queue of length n
  - Decomposition (halving) - 1
    - With creating new queues this is cost n
    - Can be reimplemented similarly to binary search
  - Compose - n
    - Fixed amount of work for every item in the result
### mergeSort() - worst case cost

<table>
<thead>
<tr>
<th>Length of Queue</th>
<th>Worst case cost of mergeSort</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(1) = 1</td>
</tr>
<tr>
<td>2</td>
<td>(1+2) + C(1) + C(1) = 3 + 1 + 1 = 5</td>
</tr>
<tr>
<td>3</td>
<td>(1+3) + C(2) + C(1) = 4 + 5 + 1 = 10</td>
</tr>
<tr>
<td>4</td>
<td>(1+4) + C(2) + C(2) = 5 + 5 + 5 = 15</td>
</tr>
<tr>
<td>5</td>
<td>(1+5) + C(3) + C(2) = 6 + 10 + 5 = 21</td>
</tr>
<tr>
<td>6</td>
<td>(1+6) + C(3) + C(3) = 7 + 10 + 10 = 27</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>n</td>
<td>(n+1) + 2*C(n/2) for n assumed even</td>
</tr>
</tbody>
</table>

Note: this calculation is SAME as best case quickSort()!
How to obtain a closed form?

\[ C(n) = (n+1) + 2*C(n/2) \]
\[ C(n/2) = (n/2+1) + 2*C(n/4) \]
\[ C(n/4) = (n/4+1) + 2*C(n/8) \]

Assumes length of the queue is a power of 2

\[ C(n) = (n+1) + 2*C(n/2) \]
\[ = (n+1) + 2*( (n/2+1) + 2*C(n/4) ) \]
\[ = (n+1) + (n+2) + 4*C(n/4) \]
\[ = (n+1) + (n+2) + 4*( (n/4+1) + 2*C(n/8) ) \]
\[ = (n+1) + (n+2) + (n+4) + 8*C(n/8) \]
Closed form: worst case

mergeSort()

\[ C(n) = (n+1) + (n+2) + (n+4) + 8 \times C(n/8) \]

\[ = (n+1) + (n+2) + (n+4) + \ldots + (2n) \]

\[ < (2n) \times \log_2(n) \]

C(n) = \( O(n \times \log_2(n)) \) in worst case
Work at each recursive step

- How much work at each level \( n \)
- How many levels \( \log_2(n) \)
- Total work is \( n \times \log_2(n) = \mathcal{O}(n \log_2(n)) \)

<table>
<thead>
<tr>
<th>Work</th>
<th>(1 \times n)</th>
<th>(2 \times \frac{n}{2})</th>
<th>(4 \times \frac{n}{4})</th>
<th>(8 \times \frac{n}{8})</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\log_2(n))</td>
<td>\n/2</td>
<td>\n/4</td>
<td>\n/8</td>
<td>\n/8</td>
</tr>
<tr>
<td>(n/2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(n/2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(n/4)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(n/4)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(n/4)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(n/4)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(n/4)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(n/4)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Randomly selected values; varied number of values and type of sort applied; counted instructions executed

Experiments

Number of Instructions Executed

- optimally implemented selectionSort
- poorly implemented mergeSort

Size of Queue
Experiments

randomly selected values; varied number of values and type of sort applied; counted instructions executed

Number of Instructions Executed

Size of Queue

quickSort
mergeSort
Histogram of quickSort()

10,000 quickSorts of 200 elements each; data randomly generated; observing $n \lg_2 n$ behavior

Number of problems that required the given instruction count

Number of instructions executed

worst case would yield 20,000
Asymptotic Complexity

\[ g(n) = O(f(n)) \]

pronounced: \( g(n) \) is order \( f(n) \)
meaning: for large \( n \), \( g(n) \) grows no faster than \( f(n) \)
definition: \( \exists \ n_0, C \ \forall \ n > n_0 \quad g(n) \leq C \times f(n) \)

\( f(n) \) is usually expressed in reduced form as a function of \( n \)
\[
\begin{align*}
1 \quad \lg(n) \quad n^{1/2} \quad n \quad n \times \lg(n) \quad n^2
\end{align*}
\]
Graphical Interpretation

\[ \exists \ n_0, C \ \forall \ n > n_0 \ \ g(n) \leq \ C \cdot f(n) \]

\( g(n) \) is order \( f(n) \)
Examples

• Is $3n^2$ order $n^2$?
  – Can a constant be found to make it so?
  – Yes, for any $n$, $3n^2 < 4n^2$

• Is $17n$ order $n^2$?
  – Yes, for $n \geq 18$, $17n < n^2$

• Big O notation bounds from above
  – Means algorithm behaves no worse than this

• Other notations exist to bound from below and to talk about on average performance