Sorting

- Sorting
  - Selection Sort
  - Quicksort
- Complexity
- Sortable Interface

Sorting

- Definition
  - Input
    - Unordered collection of items
    - Method that can compare two of the items (i.e., lessThan())
  - Output
    - Ordered collection of items
- Very useful problem
- Several algorithms developed and studied

Sorting Algorithms

- Fast versus slow
  - $O(n^2)$ on US population takes 20 years
  - $O(n*lg_2 n)$ on US population takes 1 minute
- Emphasize problem decomposition and speed but not memory usage
  - Approaches can be made more space efficient
- In-memory sorting, rather than using lots of data on external devices

Sorting Algorithms

- Our approach
  - Use queues to hold data
  - Some sort methods can’t be done this way
- Usual approach (in procedural language)
  - Use arrays to hold data
  - Sort method explained in terms of array subscript operations
  - More efficient in storage usage, but hard to see similarities among methods

Problem Decomposition

```
public SortProblem Sort() throws QueueException {
```

```
  if (getLength() == 1) return this;
```

Terminal case

```
  SortProblem sp1 = new SortProblem(),
      sp2 = new SortProblem();
      SortProblem sp1sorted, sp2sorted;
  Decompose(sp1,sp2);
```

Decompose into
2 subproblems

```
  sp1sorted = sp1.Sort();
  sp2sorted = sp2.Sort();
```

Solve
subproblems

```
  return sp1sorted.Compose(sp2sorted);
```

Compose
Solutions

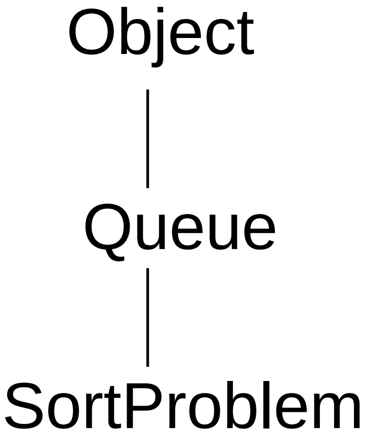
```
  }
```

SortProblem Class extends Queue

```
package cs111.util;
import java.util.Enumeration;

public class SortProblem extends Queue{

  public SortProblem() {
      super();
  }
  public SortProblem(Queue q) {
      super();
      Enumeration qe = q.getEnumeration();
      while (qe.hasMoreElements())
            enter(qe.nextElement());
  }
```



Object

Queue

SortProblem

Selection Sort

- Seen previously as an in-place sorting method using arrays
- At each step found smallest element in the remaining elements
- Grew sorted array elements from left to right in the array, using swap operations
- Here is different formulation using recursion and a “copy”of the to-be-sorted numbers

Previous Selection Sort

```
//to sort descending exchange > for <
void selection Sort(int [] a){
  int tmp, chosen;
  for(int left=0; left<a.length-1; left++){
      chosen = left;//first unsorted number
      for (int j=left+1; j<a.length; j++){
      //find smallest unsorted element
            if (a[j]<a[chosen]) chosen=j;}
      //exchange a[chosen] with a[left]
      tmp = a[chosen];
      a[chosen] = a[left];
      a[left] = tmp;
  }
}
```

selectionSort() method

```
public SortProblem selectionSort() throws
      QueueException {
    if (getLength() == 1) return this;

    SortProblem  sp1 = new SortProblem(),
                 sp2 = new SortProblem();
    SortProblem sp1sorted, sp2sorted;
    smallestAndRest(sp1,sp2);

    sp1sorted = sp1.selectionSort();
    sp2sorted = sp2.selectionSort();

    return sp1sorted.append(sp2sorted);
  }
```

- **What are smallestAndRest() and append()?**

smallestAndRest()

- **Decomposes original queue into two smaller queues**
  - ***small*** **contains the smallest element**
  - ***large*** **contains everything else**
- **Sortable is an interface which requires a lessThan() method**
  - **Objects removed from the queue must be Sortable to be compared -- notice the necessary cast**

smallestAndRest()

```
private void smallestAndRest(SortProblem small,
      SortProblem large) throws QueueException {

    Sortable smallest = (Sortable)remove();
    while (!empty()) {
      Sortable nxt = (Sortable)remove();
      if (nxt.lessThan(smallest))
      {
        large.enter(smallest);
        smallest = nxt;
      }
      else large.enter(nxt);
    }
    small.enter(smallest);
  }
```

append()

- Have to concatenate two queues when they are returned sorted
- Requires creation of a new queue
  - Can be implemented without a new queue
    - Recall original where newly found smallest is exchanged with another element as sorted array grows from left to right in the array

append()

```
private SortProblem append(SortProblem suffix)
           throws QueueException {

    SortProblem ret = new SortProblem();
    while (!empty())
      ret.enter(remove());
    while (!suffix.empty())
      ret.enter(suffix.remove());
    return ret;
 }
```

Note: time to append is linear in size of result; but this job also can be done in constant time. How?

What does selectionSort() cost?

- Cost for n element queue
  - Decomposition - n
  - Append - 1 (our method uses n, but 1 is possible)
  - Subproblems
    - Small requires 1 instruction
    - Large uses selectionSort() to sort a queue of n-1 elements

What does selectionSort() cost?

Cost(decomp)+Cost(append)+Cost(subprobs solution)

Length of Queue	Cost of Selection Sort
1	$(1) = 1$
2	$(2 +1) + C(1) + C(1) = 3 + 1 + 1 = 5$
3	$(3 +1) + C(2) + C(1) = 4 + 5 + 1 = 10$
4	$(4 +1) + C(3) + C(1) = 5 +10 +1 = 16$
...	...
n	$(n +1) + C(n-1) + 1$

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

Getting a closed form

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

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

$$C(n-2) = (n) + C(n-3)$$

$$\begin{aligned} C(n) &= (n+2) + C(n-1) \\ &= (n+2) + (n+1) + C(n-2) \\ &= (n+2) + (n+1) + (n) + C(n-3) \\ &\ldots \end{aligned}$$

Getting a closed form

$$C(n) = (n+2) + (n+1) + (n) + C(n-3)$$

$$= (n+2) + (n+1) + (n) + \ldots\ (5) + (4) + (1)$$

We’ve seen this sum before, but here terms 2 and 3 are missing: $1+2+3+4+\ldots+n = (n+1)n/2$

So, $$C(n) = (((n+2) +1) *(n+2) / 2) - 2 - 3$$

$$= (.5*(n+3)) * (n+2) - (2+3)$$

$\boxed{O(n^2)}$

Average Value

Number of Values

Correction

quickSort()

- **How to decompose?**
  - **Select a value**
  - **Put all elements larger than value in *large***
  - **Put all elements smaller than value in *small***
  - **Want selected value to be a “middle” value to divide elements into close to evenly sized sets**
- **Get two subproblems easy to combine into answer**

quickSort()

- How do we get a “middle” value?
  - Guess
  - Use any element as possible middle value
  - Use first element as possible middle value
- Use an arbitrary value and “on average” be close to middle value
- Pathologically bad cases exist with this method
  - If guessed value is smallest or largest, we will be doing selectionSort()

quickSort()

```
public SortProblem quickSort() throws QueueException {
    if (getLength() == 1) return this;

    SortProblem  sp1 = new SortProblem(),
                 sp2 = new SortProblem();
    SortProblem sp1sorted, sp2sorted;
    nearMiddle(sp1,sp2);

    sp1sorted = sp1.quickSort();
    sp2sorted = sp2.quickSort();
    return sp1sorted.append(sp2sorted);
  }
```

Different than selectionSort

Same as selectionSort

nearMiddle()

```
private void nearMiddle(SortProblem small,
            SortProblem large) throws QueueException
{
    Sortable middleValue = (Sortable)remove();
    while (!empty()) {
      Sortable nxt = (Sortable)remove();
      if (nxt.lessThan(middleValue))
        small.enter(nxt);
      else
        large.enter(nxt);
    }
    if (small.getLength() < large.getLength())
      small.enter(middleValue);
    else
      large.enter(middleValue);
}
```

Cost of quickSort()

- Depends on middle value
  - *Best case* (cheapest) occurs when middle guess is correct
  - *Worst case* (most expensive) occurs when middle guess is very wrong
    - Worst case when guess is largest or smallest element
    - Get selection sort which is $O(n^2)$
  - *Average case* - if know distribution of elements to be sorted, can argue what happens *on average* if sort many, many sets of elements (we won't examine this in cs111)

quickSort() - Best Case cost

Length of Queue	Best case cost of Quick Sort divides data in half		
1	(1)	=	1
2	(2+1) + C(1) + C(1) = 3 + 1 + 1	=	5
3	(3+1) + C(2) + C(1) = 4 + 5 + 1	=	10
4	(4+1) + C(2) + C(2) = 5 + 5 + 5	=	15
5	(5+1) + C(3) + C(2) = 6 + 10 + 5	=	21
6	(6+1) + C(3) + C(3) = 7 + 10 +10	=	27
...	...		
n	(n+1) + 2*C(n/2) for n assumed even		

How to get 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 queue is a power of 2

$$
\begin{aligned}
C(n) &= (n+1) + 2*C(n/2) \\
&= (n+1) + 2*(\ (n/2+1) + 2*C(n/4)\ ) \\
&= (n+1) + (n+2) + 4*C(n/4) \\
&= (n+1) + (n+2) + 4*(\ (n/4+1) + 2*C(n/8)\ ) \\
&= (n+1) + (n+2) + (n+4) + 8*C(n/8)
\end{aligned}
$$

…

How to get a closed form?

$$C(n) = (n+1) + (n+2) + (n+4) + 8*C(n/8)$$

$$= (n+1) + (n+2) + (n+4) + \;...\; + (2n)$$

$$< (2n)* \lg_2(n)$$

largest
value

number of
values

one value for each power of 2 up to k where $2^k = n$

$C(n) = O(n*\lg_2(n))$ in best case for quickSort()

(and is average case as well)

Selection vs Quick Sort

- Methods differ only in decomposition step
- quickSort() degenerates in worst case into selectionSort()
  - Middle guess may be very bad
- Decomposition choice makes difference between $O(n^2)$ and $O(n \lg_2 n)$
  - smallestAndRest(): 9 lines of code (20 years)
  - nearMiddle(): 11 lines of code (1 minute)