CS203 Programming with Data Structures
Sorting

Chengyu Sun
California State University, Los Angeles

Sorting
◆ Given an collection of elements, rearrange the elements so that they are in ascending or descending order
  ◆ The collection is usually an array
  ◆ The elements are comparable
◆ For example:
    Before sorting: 30 10 15 21 18 25
    After sorting: 10 15 18 21 25 30

Bubble Sort
◆ Given an array of size N
  For i=0 to N-1
    find the smallest element in the range i+1 to N-1,
    and swap it with the element at position i
  Done
◆ Or in other words
  . Find the smallest element and put it at 1\textsuperscript{st} position
  . Find the second smallest element and put it ant 2\textsuperscript{nd} position
  . ...

Bubble Sort Example
0 1 2 3 4 5
30 10 15 21 18 25
After 1\textsuperscript{st} iteration:
10 30 15 21 18 25
First iteration:
  i=0
  . The smallest element in the range 1 to 5 is 10
  . Swap 10 and 30

Comparison and Swap
◆ Bubble sort
  . Number of comparisons??
  . Number of swaps??
◆ Can we improve on Bubble Sort??

Decision Tree

a,b,c a,c,b b,c,a c,b,a
a,b,c a,c,b b,c,a c,b,a
a,c,b c,b,a
b,c,a
Properties of A Decision Tree

- Number of leaves: ??
- Height of the tree: ??

Any sorting algorithm that only uses comparisons between elements require at least $O(N \log N)$ comparisons.

Insertion Sort

- Make $N-1$ passes of the array
- At pass $p$,
  - the first $p$ elements of the array are already sorted.
  - "insert" $a[p+1]$ so the first $p+1$ elements are sorted.

Insertion Sort Example

1st Pass: sorted | unsorted

Take 10 and insert it into the sorted portion:

After insertion:

Insertion Sort Implementation

Implementation #1:
- Binary search for insertion position
- Shift elements to the right
- Insert

Complexity of pass $P$
- $\log P$ comparisons
- $P/2$ assignments

Implementation #2:
- Pair-wise swap

Complexity of pass $P$
- $P/2$ comparisons
- $P/2$ swaps

Insertion Sort Complexity

- Best case: ??
- Worst case: ??
- Average case: $O(N^2)$
Heap Sort

- Heap sort strategy
  - Construct a heap with \( N \) insertions: \( O(??) \)
  - Construct a sorted array with \( N \) removeMin: \( O(??) \)

- Can we construct the heap more efficiently (in linear time)??
- Can we perform heap sort without the extra space requirement??

Percolate Down

```java
void percolateDown( int pos )
{
  int child;
  Comparable tmp = array[pos];
  while( pos*2 <= size )
  {
    child = pos * 2;
    if( child != size && array[child+1].compareTo(array[child]) < 0 ) child++;
    if( array[child].compareTo(tmp) < 0 ) array[pos] = array[child];
    else break;
    pos = child;
  }
  array[pos] = tmp;
}
```

Building A Heap

- Percolate down the non-leaf nodes in reverse order
Heap Building Example

Heap Building Complexity
- The complexity of percolate down one node is: ??
- The complexity of percolate down all non-leaf nodes is: O(N)

Heap Sort Algorithm
- Build a MaxHeap
- removeMax() then put the removed value into the last position

Merge Sort
- Merging two sorted arrays takes linear time

Merge Sort Example

Merge Sort Code ...
```java
Comparable tmpArray[];
void mergeSort( Comparable a[] )
{
    tmpArray = new Comparable[a.length];
    mergeSort( a, 0, a.length-1 );
}
```
... Merge Sort Code
void mergeSort( Comparable a[], int left, int right )
{
    if( left < right )
    {
        int mid = (left+right) / 2;
        mergeSort( a, left, mid );
        mergeSort( a, mid+1, right );
        merge( a, left, mid, right );
    }
}

About Merge Sort
◆ Complexity
  ◦ T(1) = 1
  ◦ T(N) = 2T(N/2) + N
  ◦ O(NlogN)
◆ Rarely used in practice
  ◦ Require extra space for the temporary array
  ◦ Copying to and from the temporary array is costly

Quick Sort
◆ Fastest sorting algorithm in practice
◆ Complexity
  ◦ Average case: O(NlogN)
  ◦ Worst case: O(N^2)
◆ Easy to understand, very hard to code correctly

Quick Sort Algorithm
Given array A
1. If |A| = 1 or 0, return
2. Pick any element v in A. v is called the pivot.
3. Partition A-{(v)} (the remaining elements in A) into two disjoint groups A1 and A2
   ◦ A1 = {x ∈ A-{(v)} | x ≤ v}
   ◦ A2 = {x ∈ A-{(v)} | x ≥ v}
4. Return {quicksort(A1), v, quicksort(A2)}

Understand The Notations
A:  30, 10, 15, 21, 18, 25
v:  25
A1:  10, 15, 21, 18
A2:  30

Observations About The Quick Sort Algorithm
◆ It should work
◆ It’s not very clearly defined
  ◦ How do we pick the pivot?
  ◦ How do we do the partitioning?
  ◦ How do we handle duplicate values?
◆ Why is it more efficient than Merge Sort?
Picking the Pivot

- Ideally, the pivot should lead to two equal-sized partitions
  - First element??
  - Random pick??
  - Median of (first, middle, last)

Partitioning ...

A: 

\[ \begin{array}{llllll}
30 & 15 & 21 & 18 & 25 & 10 \\
\end{array} \]

Pivot = median(30,21,10) = 21

1. Swap pivot to the last position

\[ \begin{array}{llllll}
30 & 15 & 10 & 18 & 25 & 21 \\
\end{array} \]

\[ \begin{array}{llll}
\text{i} & \text{j} \\
\end{array} \]

... Partitioning

We want to move smaller elements to the left part of the array and larger elements to the right part, so:

2. Increase \( i \) until \( a[i] > \text{pivot} \)
   - Decrease \( j \) until \( a[j] < \text{pivot} \)
   - Swap \( a[i] \) and \( a[j] \)

3. Repeat 2 until \( i > j \)

4. Swap \( a[i] \) and pivot

More Details

- Handling duplicates
- Small arrays
  - \( N > 10 \): quick sort
  - \( N \leq 10 \): insertion sort

Exercise

- Implement quickSort