Quicksort, Mergesort, and Heapsort
Quicksort

- Fastest known sorting algorithm in practice
  - Caveats: not stable
  - Vulnerable to certain attacks

- Average case complexity $\Rightarrow O(N \log N)$

- Worst-case complexity $\Rightarrow O(N^2)$
  - Rarely happens, if coded correctly
QuickSort Outline

- Divide and conquer approach
- Given array $S$ to be sorted
  - If size of $S \leq 1$ then done;
  - Pick any element $v$ in $S$ as the pivot
  - Partition $S - \{v\}$ (remaining elements in $S$) into two groups
    - $S_1 = \{\text{all elements in } S - \{v\} \text{ that are smaller than } v\}$
    - $S_2 = \{\text{all elements in } S - \{v\} \text{ that are larger than } v\}$
  - Return $\{\text{quickSort}(S_1) \text{ followed by } v \text{ followed by } \text{quickSort}(S_2)\}$
- Trick lies in handling the partitioning (step 3).
  - Picking a good pivot
  - Efficiently partitioning in-place
Quicksort example

Select pivot

partition

Recursive call

Merge
Picking the Pivot
Picking the Pivot

- How would you pick one?
Picking the Pivot

- How would you pick one?

- Strategy 1: Pick the first element in s
  - Works only if input is random
  - What if input s is sorted, or even mostly sorted?
    - All the remaining elements would go into either $s_1$ or $s_2$!
    - Terrible performance!
  - Why worry about sorted input?
    - Remember $\rightarrow$ Quicksort is recursive, so sub-problems could be sorted
    - Plus mostly sorted input is quite frequent
Picking the Pivot (contd.)

- Strategy 2: Pick the pivot randomly
  - Would usually work well, even for mostly sorted input
  - Unless the random number generator is not quite random!
  - Plus random number generation is an expensive operation
Picking the Pivot (contd.)

- **Strategy 3: Median-of-three Partitioning**
  - *Ideally*, the pivot should be the median of input array $S$
    - Median = element in the middle of the sorted sequence
  - Would divide the input into two almost equal partitions
  - Unfortunately, it's hard to calculate median quickly, without sorting first!
  - So find the approximate median
    - Pivot = median of the left-most, right-most and center element of the array $S$
    - Solves the problem of sorted input
Picking the Pivot (contd.)

- Example: Median-of-three Partitioning
  - Let input $S = \{6, 1, 4, 9, 0, 3, 5, 2, 7, 8\}$
  - $left=0$ and $S[left] = 6$
  - $right=9$ and $S[right] = 8$
  - $center = (left+right)/2 = 4$ and $S[center] = 0$
  - Pivot
    - $= \text{Median of } S[left], S[right], \text{ and } S[center]$
    - $= \text{median of } 6, 8, \text{ and } 0$
    - $= S[left] = 6$
Partitioning Algorithm

- Original input: \( S = \{6, 1, 4, 9, 0, 3, 5, 2, 7, 8\} \)
- Get the pivot out of the way by swapping it with the last element
  
  \[
  8 \ 1 \ 4 \ 9 \ 0 \ 3 \ 5 \ 2 \ 7 \ \boxed{6} \]
  
  pivot

- Have two ‘iterators’ – \( i \) and \( j \)
  - \( i \) starts at first element and moves forward
  - \( j \) starts at last element and moves backwards

\[
\boxed{8} \ 1 \ 4 \ 9 \ 0 \ 3 \ 5 \ 2 \ \boxed{7} \ \boxed{6} \\
\text{i} \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{j} \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{pivot}
\]
Partitioning Algorithm (contd.)

- While \((i < j)\)
  - Move \(i\) to the right till we find a number greater than \(\text{pivot}\)
  - Move \(j\) to the left till we find a number smaller than \(\text{pivot}\)
  - If \((i < j)\) \(\text{swap}(S[i], S[j])\)
  - (The effect is to push larger elements to the right and smaller elements to the left)

Swap the \(\text{pivot}\) with \(S[i]\)
## Partitioning Algorithm Illustrated

<table>
<thead>
<tr>
<th>Move</th>
<th>Swap</th>
<th>Move</th>
<th>Swap</th>
<th>Move</th>
<th>Swap</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 1 4 9 0 3 5 2 7 6</td>
<td>2 1 4 9 0 3 5 8 7 6</td>
<td>2 1 4 9 0 3 5 8 7 6</td>
<td>2 1 4 5 0 3 9 8 7 6</td>
<td>2 1 4 5 0 3 9 8 7 6</td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>j</td>
<td>i</td>
<td>j</td>
<td>j</td>
<td>i</td>
</tr>
</tbody>
</table>

Swap $S[i]$ with pivot

2 1 4 5 0 3 9 8 7 6

i and j have crossed
Dealing with small arrays

- For small arrays \((N \leq 20)\),
  - Insertion sort is faster than quicksort

- Quicksort is recursive
  - So it can spend a lot of time sorting small arrays

- Hybrid algorithm:
  - Switch to using insertion sort when problem size is small
    (say for \(N < 20\))
Quicksort Driver Routine

1  /**
2   * Quicksort algorithm (driver).
3   */
4  template<typename Comparable>
5  void quicksort( vector<Comparable> & a )
6  {
7      quicksort( a, 0, a.size() - 1 );
8  }
QuickSort Pivot Selection Routine

```cpp
/**
 * Return median of left, center, and right.
 * Order these and hide the pivot.
 */

template<typename Comparable>
const Comparable & median3( vector<Comparable> & a, int left, int right )
{
    int center = ( left + right ) / 2;
    if( a[ center ] < a[ left ] )
        swap( a[ left ], a[ center ] );
    if( a[ right ] < a[ left ] )
        swap( a[ left ], a[ right ] );
    if( a[ right ] < a[ center ] )
        swap( a[ center ], a[ right ] );

    // Place pivot at position right - 1
    swap( a[ center ], a[ right - 1 ] );
    return a[ right - 1 ];
}
```

Swap `a[left]`, `a[center]` and `a[right]` in-place

Pivot is in `a[center]` now

Swap the pivot `a[center]` with `a[right-1]`
Quicksort routine

```cpp
/**
 * Internal quicksort method that makes recursive calls.
 * Uses median-of-three partitioning and a cutoff of 10.
 * a is an array of Comparable items.
 * left is the left-most index of the subarray.
 * right is the right-most index of the subarray.
 */

template <typename Comparable>
void quicksort( vector<Comparable> & a, int left, int right )
{
    if( left + 10 <= right )
    {
        Comparable pivot = median3( a, left, right );

        // Begin partitioning
        int i = left, j = right - 1;
        for( ; ; )
        {
            while( a[++i] < pivot ) { }
            while( pivot < a[ --j ] ) { }
            if( i < j )
                swap( a[i], a[j] );
            else
                break;
        }

        swap( a[i], a[right - 1] ); // Restore pivot

        quicksort( a, left, i - 1 ); // Sort small elements
        quicksort( a, i + 1, right ); // Sort large elements
    } else // Do an insertion sort on the subarray
        insertionSort( a, left, right );
```
Exercise: Runtime analysis

- Worst-case
- Average case
- Best case
Heapsort

- Build a binary heap of N elements
  - $O(N)$ time

- Then perform $N$ \texttt{deleteMax} operations
  - $\log(N)$ time per \texttt{deleteMax}

- Total complexity $O(N \log N)$
Example

After BuildHeap

After first deleteMax
Heapsort Implementation

```cpp
/**
 * Standard heapsort.
 */
template <typename Comparable>
void heapsort( vector<Comparable> & a )
{
    for( int i = a.size() / 2; i >= 0; i-- )
        percDown( a, i, a.size() );
    for( int j = a.size() - 1; j > 0; j-- )
    {
        swap( a[ 0 ], a[ j ] );
        percDown( a, 0, j );
    }
}

/**
 * Internal method for heapsort that is used in deleteMax and buildHeap
 * i is the position from which to percolate down.
 * n is the logical size of the binary heap.
 */
template <typename Comparable>
void percDown( vector<Comparable> & a, int i, int n )
{
    int child;
    Comparable tmp;
    for( tmp = a[ i ]; leftChild( i ) < n; i = child )
    {
        child = leftChild( i );
        if( child != n - 1 && a[ child ] < a[ child + 1 ] )
            child++;
        if( tmp < a[ child ] )
            a[ i ] = a[ child ];
        else
            break;
    }
    a[ i ] = tmp;
}

// Internal method for heapsort.
// i is the index of an item in the heap.
// Returns the index of the left child.

inline int leftChild( int i )
{
    return 2 * i + 1;
}
```
Mergesort

- Divide the $N$ values to be sorted into two halves

- Recursively sort each half using Mergesort
  - Base case $N=1 \implies$ no sorting required

- Merge the two halves
  - $O(N)$ operation

- Complexity??
  - We’ll see
Mergesort Implementation

```cpp
/**
 * Mergesort algorithm (driver).
 */
template<typename Comparable>
void mergeSort( vector<Comparable> & a )
{
    vector<Comparable> tmpArray( a.size() );

    mergeSort( a, tmpArray, 0, a.size() - 1 );
}

/**
 * Internal method that makes recursive calls.
 * a is an array of Comparable items.
 * tmpArray is an array to place the merged result.
 * left is the left-most index of the subarray.
 * right is the right-most index of the subarray.
 */
template<typename Comparable>
void mergeSort( vector<Comparable> & a,
                vector<Comparable> & tmpArray, int left, int right )
{
    if( left < right )
    {
        int center = ( left + right ) / 2;
        mergeSort( a, tmpArray, left, center );
        mergeSort( a, tmpArray, center + 1, right );
        merge( a, tmpArray, left, center + 1, right );
    }
}
```
Complexity analysis

- \( T(N) = 2T(N/2) + N \)
  - Recurrence relation
- \( T(N) = 4T(N/4) + 2N \)
- \( T(N) = 8T(N/8) + 3N \)
- ...
- \( T(N) = 2^k T(N/2^k) + k*N \)

For \( k = \log N \)
  - \( T(N) = N \ T(1) + N \ \log N \)