Advanced Data Structures and Algorithms

CS 361 – Fall 2013

Lec. #10: Sorting II

Tamer Nadeem
Dept. of Computer Science
Class Objective/Overview

• Familiarize with **Sorting Speed Limit**

• Understand **Merge Sort and its Time/Space Analysis**

• Familiarize with **Divide and Conquer**

• Understand **Quick Sort and its Best/Average/Worst Case Analysis**

• Understand **Binary Heap Trees**

• Understand **Heap Sort and its Time Analysis**
Sorting
Sorting Speed Limits

• We’ve gone from sorting in $O(n^2)$ time to sorting in some rather odd times: $O(n^{5/4})$, $O(n^{3/2})$, etc.

• **How fast can** a sorting algorithm get?

• Given set of *sorted n-1 elements*, how many comparisons are needed to determine the proper order of the $n^{th}$ element?
  
  • $n = 2$ needs 1 comparison
  
  • $n = 3$ needs *between 2 and 3* comparisons
  
  • $n = k + 1$
    
    • Using linear search $\rightarrow$ needs $k$ comparisons
    
    • Using binary search $\rightarrow$ needs $\log k$ comparisons
Sorting Speed Limits

- Then, sorting \( n \) elements requires \( \sum_{i=1}^{n} (\log i) \) comparisons.

- Then:
  \[
  \sum_{i=1}^{n} \log(i) \geq \sum_{i=n/2}^{n} \log(i) \\
  \geq \sum_{i=n/2}^{n} \log(n/2) \\
  = n/2 \log(n/2) \\
  = n/2(\log(n) - \log(2)) \\
  = O(n \log(n))
  \]

Therefore, no sorting algorithm that works by pair-wise comparison (i.e., comparing elements 2 at a time) can be faster than \( O(n \log(n)) \) (worst or average case).
Merge Sort
Merge Sort

- Assume our set of elements are divided into two sorted parts; 

\[ \text{[first..mid-1]} \] and \([\text{mid..last-1}]\). Then we could merge the two parts into a combined sorted sequence using the following code:

```cpp
template<typename T>
void merge(vector<T>& v, int first, int mid, int last)
{
    // temporary vector to merge the sorted sub lists
    vector<T> tempVector;
    int indexA, indexB, indexV;
    // set indexA to scan sublistA ( index range [first, mid) )
    // and indexB to scan sublistB ( index range [mid, last] )
    indexA = first;
    indexB = mid;
    // while both sub lists are not exhausted , compare
    // v[indexA] and v[indexB]
    // copy the smaller to vector temp using push_back ( )
    while (indexA < mid && indexB <= last)
    {
        if (v[indexA] < v[indexB])
        {
            // copy element to temp
            tempVector.push_back(v[indexA]);
            indexA++;
            // increment indexA
        }
        else
        {
            // copy element to temp
            tempVector.push_back(v[indexB]);
            indexB++;
            // increment indexB
        }
    }
    // copy the tail of the sublist that is not exhausted
    while (indexA < mid)
    {
        tempVector.push_back(v[indexA]);
        indexA++;
    }
    while (indexB <= last)
    {
        tempVector.push_back(v[indexB]);
        indexB++;
    }
    // copy tempVector using indexV to v using indexA
    indexA = first;
    // copy elements from temporary vector to original list
    for (indexV=0; indexV < tempVector.size(); indexV++)
    {
        v[indexA] = tempVector[indexV];
        indexA++;
    }
}
```
Understanding the Merge Algorithm

- The heart of the merge algorithm is the first loop (highlighted) that **merges** two sorted subsequences into a **single** sorted **tempVector**.
- The way to do this is by **comparing** the **first element in each** of the two input (sub)sequences and **copy the smaller one**.
- We **exit from the loop** when **one** of the arrays has been completely **emptied out**.

```
template <typename T>
void merge(vector <T>& v, int first, int mid, int last) {
    // temporary vector to merge the sorted sub lists
    vector <T> tempVector ;
    int indexA , indexB , indexV ;
    // set indexA to scan sublistA ( index range [first, mid) )
    // and indexB to scan sublistB ( index range [mid, last) )
    indexA = first;
    indexB = mid;
    // while both sub lists are not exhausted , compare
    // v[indexA] and v[indexB]
    // copy the smaller to vector temp using push_back ( )
    while ( indexA < mid && indexB <= last)
        if ( v[indexA] < v[indexB] ) {
            // copy element to temp
            tempVector.push_back (v[indexA]) ;
            indexA++; // increment indexA
        }
        else {
            // copy element to temp
            tempVector.push_back (v[indexB]);
            indexB++; // increment indexB
        }
}
```
Understanding the Merge Algorithm

v[A]  
3  10  43  54

v[B]  
1  5  25  30

tempVector

indexA

indexB

indexV
Understanding the Merge Algorithm

\( v[A] \)

\[ \begin{array}{cccc}
3 & 10 & 43 & 54 \\
\end{array} \]

\( v[B] \)

\[ \begin{array}{ccc}
5 & 25 & 30 \\
\end{array} \]

tempVector

\[ \begin{array}{cccc}
1 & & & \\
\end{array} \]

indexA

indexB

indexV
Understanding the Merge Algorithm

v[A]  

10  43  54

v[B]

5  25  30

tempVector

1  3

indexA

indexB

indexV
Understanding the Merge Algorithm

<table>
<thead>
<tr>
<th>v[A]</th>
<th>10</th>
<th>43</th>
<th>54</th>
</tr>
</thead>
<tbody>
<tr>
<td>v[B]</td>
<td>25</td>
<td>30</td>
<td></td>
</tr>
</tbody>
</table>

```
1 3 5
```

```
indexA
```

```
indexB
```

```
tempVector
```

```
indexV
```
Understanding the Merge Algorithm

v[A]  

|   |   | 43 | 54 |

v[B]  

|   |   | 25 | 30 |

indexA

indexB

tempVector

| 1 | 3 | 5 | 10 |

indexV
Understanding the Merge Algorithm

\[ v[A] \]
\[
\begin{array}{|c|c|}
\hline
43 & 54 \\
\hline
\end{array}
\]

\[ v[B] \]
\[
\begin{array}{|c|}
\hline
30 \\
\hline
\end{array}
\]

\[ \text{tempVector} \]
\[
\begin{array}{|c|c|c|c|c|}
\hline
1 & 3 & 5 & 10 & 25 \\
\hline
\end{array}
\]

indexA

indexB

indexV
Understanding the Merge Algorithm

v[A]

v[B]

indexA

indexB

tempVector

1 3 5 10 25 30
Understanding the Merge Algorithm

• The *rest* of the algorithm is “*cleanup*”.

• After exiting main loop, there is a possibility that *one* (at most) of the subsequences still has data.

• The *next two loops* copy that data from the remainder of the two subsequences.
  • Because one of those subsequences has been emptied, one of these loops will execute zero times.

• Finally, the *last loop* copies the *entire merged data set* back out of the temporary vector into the original vector.

```cpp
// copy the tail of the sublist that is not exhausted
while (indexA < mid) {
    tempVector.push_back(v[indexA]);
    indexA++;
}

while (indexB <= last) {
    tempVector.push_back(v[indexB]);
    indexB++;
}

// copy tempVector using indexV to v using indexA
indexA = first;
for (indexV = 0; indexV < tempVector.size(); indexV++) {
    v[indexA] = tempVector[indexV];
    indexA++;
}
```
Understanding the Merge Algorithm

\[ v[A] \]

\[ \begin{array}{c}
43 \\
54
\end{array} \]

\[ v[B] \]

\[ \begin{array}{c}
\text{indexA}
\end{array} \]

\[ \begin{array}{c}
\text{indexB}
\end{array} \]

\[ \begin{array}{c}
tempVector
\end{array} \]

\[ \begin{array}{cccccc}
1 & 3 & 5 & 10 & 25 & 30
\end{array} \]

\[ \text{indexV} \]
Understanding the Merge Algorithm

tempVector

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>5</td>
<td>10</td>
<td>25</td>
<td>30</td>
<td>43</td>
<td></td>
</tr>
</tbody>
</table>

v[A]

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>54</td>
</tr>
</tbody>
</table>

v[B]

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

indexA

indexB

indexV
Understanding the Merge Algorithm

tempVector

v[A]

v[B]

indexA

indexB

indexV

1 3 5 10 25 30 43 54
Merge Algorithm - Analysis

• Looking at the code for the first 3 loops, note that
  • each one adds one element into tempVector.
  • no element is copied multiple times.
    • If we copy the element at indexA or indexB, we also increment indexA or indexB, so we will not copy that element again.

• Given total of last-first elements, each loop can repeat no more than last-first times.
  • So all three loops are $O(last-first)$.

• The last loop copy last-first elements of tempVector
  • Loop is $O(last-first)$.

```cpp
... while (indexA < mid && indexB <= last) // O(last – first)
  if (v[indexA] < v[indexB]){
    tempVector.push_back(v[indexA]); // O(1)
    indexA++;
  }
  else{
    tempVector.push_back(v[indexB]); // O(1)
    indexB++;
  }
while (indexA < mid) {
  tempVector.push_back(v[indexA]); // O(1)
  indexA++;
}
while (indexB <= last) {
  tempVector.push_back(v[indexB]); // O(1)
  indexB++;
}
indexA = first;
for (indexV=0; indexV < tempVector.size (); indexV++) {
  v[indexA] = tempVector[indexV]; // O(1)
  indexA++;
}
```

• Given total of last-first elements, each loop can repeat no more than last-first times.
  • So all three loops are $O(last-first)$.

• The last loop copy last-first elements of tempVector
  • Loop is $O(last-first)$.
Merge Sort

- Can we use the merging algorithm that we just discussed to sort a set of elements?
Merge Sort - Example

Original Sequence

Sorted Sequence

18 26 32 6 43 15 9 1

1 6 9 15 18 26 32 43

6 18 26 32

1 9 15 43

18 26 6 32

15 43 1

18 26 32 6

43 15 9 1
Merge Sort

**Sorting Problem:**
Sort a sequence of \( n \) elements into non-decreasing order.

- **Divide:** Divide the \( n \)-element sequence to be sorted into two subsequences of \( n/2 \) elements each
- **Conquer:** Sort the two subsequences *recursively* using merge sort.
- **Combine:** Merge the two sorted subsequences to produce the sorted answer.

**Divide and Conquer**
Divide and Conquer

• Recursive in structure

  • *Divide* the problem into sub-problems that are similar to the original but smaller in size

  • *Conquer* the sub-problems by solving them *recursively*. If they are small enough, just solve them in a straightforward manner.

  • *Combine* the solutions to create a solution to the original problem
Merge Sort – The Algorithm

- It is almost amazingly simple
- Consists simply of two recursive calls to itself
  - Each attempting to sort half the vector,
- Followed by a call to merge
  - Combine the two sorted halves into a single sorted sequence.

```cpp
template <typename T>
void mergeSort ( vector <T>& v , int first , int last)
{
  // if the sublist has more than 1 element continue
  if ( first + 1 <= last )
  {
    // for sub lists of size 2 or more , call mergeSort ( )
    // for the left and right sublists and then
    // merge the sorted sublists using merge ( )
    int midpt = (last + first + 1) / 2;
    mergeSort (v, first , midpt-1) ;
    mergeSort (v, midpt , last) ;
    merge(v, first , midpt, last) ;
  }
}
```
Merge Sort – The Analysis (Time)

• Each call to mergeSort is either done in $O(1)$ time (if first+1 $\geq$ last) or divides the array into two sub-arrays.

• We split up to $\log N$ times.

• We merge at each level $N$ elements

Because we have $\log N$ levels, each level taking $O(N)$ work, the overall merge sort code is: $O(N \log N)$ (worst & average case).

template <typename T>
void mergeSort ( vector <T>& v , int first , int last )
{
    // if the sublist has more than 1 element continue
    if ( first + 1 < last )
    {
        // for sub lists of size 2 or more , call mergeSort ( )
        // for the left and right sublists and then
        // merge the sorted sublists using merge ( )
        int midpt = ( last + first + 1 ) / 2;
        mergeSort(v, first , midpt);
        mergeSort(v, midpt , last);
        merge(v, first , midpt , last);
    }
}
Implementing Merge Sort

• There are two basic ways to implement merge sort:
  
  • **In Place**: Merging is done with only the input array
    
    • **Pro**: Requires only the space needed to hold the array
    
    • **Con**: Takes longer to merge since placing a single element in right place might require shift of the sublist → worst case of merging two $K$ sublists is $O(K^2)$
  
  • **Double Storage**: Merging is done with a temporary array of the same size as the input array.
    
    • **Pro**: Faster than In Place since the temp array holds the resulting array until both left and right sides are merged into the temp array, then the temp array is appended over the input array.
    
    • **Con**: The memory requirement is doubled ($O(N)$).
Merge Sort – Final Note

- Merge Sort does the full set of comparisons and copies even when applied to arrays that are already sorted.
- The merge routine itself moves sequentially through its working arrays, not jumping from place to place.
- There are other variants of Merge Sorts including k-way merge sorting, but the common variant is the Double Memory Merge Sort.
- Though the running time is $O(N \log N)$ and runs much faster than insertion sort and bubble sort, merge sort’s large memory demands makes it *not very practical for main memory sorting.*
Quick Sort
Quick Sort

• Quick sort is another method that uses Divide and Conquer by dividing the array into progressively smaller parts.

• Instead of chopping at an arbitrary point (e.g., halfway), select any arbitrary element to be a **pivot** in dividing into sub-arrays.
  - All items in the left section are **less than or equal** to the pivot value.
  - All items on the right are **greater than** the pivot value.
  - This is often called partitioning the array.

• Quick Sort Overview:
  1) Pick a pivot element (x)
  2) Rearrange elements so that x goes to its **final position E**
  3) Recursively sort both the sublist L less than x and the sublist G greater than x.
Quick Sort – The Partitioning Algorithm

```cpp
template <typename T>
int pivotIndex(vector<T>& v, int first, int last)
{
    // index for the midpoint of [first, last] and the
    // indices that scan the index range in tandem
    int mid, scanUp, scanDown;
    // pivot value and object used for exchanges
    T pivot, temp;
    if (first - 1 == last)
        return last;
    else if (first == last)
        return first;
    else {
        mid = (last + first) / 2;
        pivot = v[mid];
        // exchange the pivot and the low end of the range
        // and initialize the indices scanUp and scanDown.
        v[mid] = v[first];
        v[first] = pivot;
        scanUp = first + 1;
        scanDown = last;
        // manage the indices to locate elements that are in
        // the wrong sublist; stop when scanDown <= scanUp
        for(;;) {
            // move up lower sublist; stop when scanUp enters
            // upper sublist or identifies an element >= pivot
            while (scanUp <= scanDown && v[scanUp] < pivot)
                scanUp++;
            // scan down upper sublist; stop when scanDown
            // locates an element <= pivot; we guarantee we
            // stop at arr[first]
            while (pivot < v[scanDown])
                scanDown--;
            // if indices are not in their sublists, partition
            complete
            if (scanUp >= scanDown)
                break;
            // indices are still in their sublists and identify
            // two elements in wrong sublists. exchange
            swap(v[scanUp], v[scanDown]);
            scanUp++;
            scanDown--;
        }
    // copy pivot to index (scanDown) that partitions
    // sublists and return scanDown
    v[first] = v[scanDown];
    v[scanDown] = pivot;
    return scanDown;
}
```

Quick Sort – The Partitioning Algorithm

- First, we choose a value to serve as the pivot element.
- In this version, we are using the midpoint of the data sequence.
- There are other possibilities, some of which will be discussed later.
- The pivotIndex algorithm basically starts with the scanUp and scanDown indices at opposite ends of the array.
  - These are moved towards each other until they meet.

```cpp
template <typename T>
int pivotIndex(vector<T>& v, int first, int last)
{
    // index for the midpoint of [first,last) and the
    // indices that scan the index range in tandem
    int mid, scanUp, scanDown;
    // pivot value and object used for exchanges
    T pivot, temp;
    if (first-1 == last)
        return last;
    else if (first == last)
        return first;
    else {
        mid = (last + first)/2;
        pivot = v[mid];
        // exchange the pivot and the low end of the range
        // and initialize the indices scanUp and scanDown.
        v[mid] = v[first];
        v[first] = pivot;
        scanUp = first + 1;
        scanDown = last;
        // manage the indices to locate elements that are in
        // the wrong sublist; stop when scanDown <= scanUp
        for(;;) {
            // move up lower sublist; stop when scanUp enters
            // upper sublist or identifies an element >= pivot
```
Quick Sort – The Partitioning Algorithm

- scanUp is moved up first, until it either meets scanDown or hits an element greater than or equal to the pivot.
- Then, scanDown is moved down until it hits an element less than or equal to the pivot.
- If scanUp is pointing to an element greater than the pivot, and scanDown is pointing to an element less than the pivot, those two elements are clearly out of order with respect to each other.
- So, we swap the two, and then resume moving scanUp and scanDown towards each other again.

```c
  for(;;) {
    // move up lower sublist; stop when scanUp enters upper sublist or identifies an element >= pivot
    while (scanUp <= scanDown && v[scanUp] < pivot)
      scanUp++;
    // scan down upper sublist; stop when scanDown locates an element <= pivot; we guarantee we stop at arr[first]
    while (pivot < v[scanDown])
      scanDown--;
    // if indices are not in their sublists, partition complete
    if (scanUp >= scanDown)
      break;
    // indices are still in their sublists and identify two elements in wrong sublists; exchange
    swap(v[scanUp], v[scanDown]);
    scanUp++;
    scanDown--;
  }
```

- \(\text{scanUp}\) is moved up first, until it either meets \(\text{scanDown}\) or hits an element greater than or equal to the pivot.
- Then, \(\text{scanDown}\) is moved down until it hits an element less than or equal to the pivot.
- If \(\text{scanUp}\) is pointing to an element greater than the pivot, and \(\text{scanDown}\) is pointing to an element less than the pivot, those two elements are clearly out of order with respect to each other.
- So, we swap the two, and then resume moving \(\text{scanUp}\) and \(\text{scanDown}\) towards each other again.
The Partitioning Algorithm - Analysis

• The loop bodies of the two while loops are clearly $O(1)$.

• How many times does each while loop execute (in the worst case)?
  
  **Answer:** Each time around the loop, we either increment scanUp or decrement scanDown. So the loop can only repeat $scanDown_0 - scanUp_0$ times, where $scanDown_0$ and $scanUp_0$ denote the starting values of scanDown and scanUp.

• How many times does *for* loop repeat?
  
  **Answer:** The total number of times the for loop body gets executed can be no more than scanDown-scanUp times

  • Looking at the initial values, the loop the loop (and the pivotIndex algorithm) are $O(last - first)$.  

```
for(;;) {
    // move up lower sublist; stop when scanUp enters
    // upper sublist or identifies an element >= pivot
    while (scanUp <= scanDown &&
           v[scanUp] < pivot)  // O(scanDown-scanUp)
         scanUp++;          // O(1)
    // scan down upper sublist; stop when scanDown
    // locates an element <= pivot; we guarantee we
    // stop at arr[first]
    while (pivot < v[scanDown])  // O(scanDown-scanUp)
        scanDown--;             // O(1)
    // if indices are not in their sublists, partition complete
    if (scanUp >= scanDown)
        break;
    // indices are still in their sublists and identify
    // two elements in wrong sublists. exchange
    swap(v[scanUp], v[scanDown]);
    scanUp++;
    scanDown--;
}
```
Quick Sort

• To actually sort data, we use the pivotIndex function within the quicksort routine shown here.

  • **First**, we check to be sure this is not a “degenerate” case of 0, 1, or 2 elements.

  • **Second**, we use the pivotIndex function to partition the array.

• By then, *all of the elements on the left are less than any element on the right.*

• Hence, by sorting the left part and the right part separately, *by recursively calling quicksort*, the array as a whole would be sorted.

```cpp
template<typename T>
void quicksort(vector<T>& v, int first, int last)
{
    // index of the pivot
    int pivotLoc;
    // temp used for an exchange when [first,last] has two elements
    T temp;
    // if the range is not at least two elements, return
    if (last - first < 1)
        return;
    // if sublist has two elements, compare v[first] and v[last] and exchange if necessary
    else if (last - first == 1){
        if (v[last] < v[first]) {
            temp = v[last];
            v[last] = v[first];
            v[first] = temp;
        }
        return;
    }
    else {
        pivotLoc = pivotIndex(v, first, last);
        quicksort(v, first, pivotLoc - 1); // make the recursive call
        quicksort(v, pivotLoc +1, last);  // make the recursive call
    }
}
```
Quick Sort – Best Case Analysis

• Each call to pivotIndex will divide the array exactly in half.
  • Each recursive call will work on exactly half of the array.

• Suppose we start with \( N \) items. Partitioning takes \( O(N) \) time.

• Then, two sub-arrays, each with \( N/2 \) elements.
  • Each of these will be partitioned at a cost of \( O(N/2) \) with total \( 2*O(N/2) = O(N) \) effort.

• Then, 4 sub-arrays of size \( N/4 \).
  • Each of these will be partitioned at a cost of \( O(N/4) \) with total \( 4*O(N/4) = O(N) \) effort.

```cpp
template <typename T>
void quicksort(vector<T>& v, int first, int last)
{
    // index of the pivot
    int pivotLoc;
    // temp used for an exchange when vector has two elements
    T temp;
    // if the range is not at least two elements, return
    if (last - first < 1)
        return;
    // if sublist has two elements, compare v[first] and // v[last] and exchange if necessary
    else if (last - first == 1){
        if (v[last] < v[first]) {
            temp = v[last];
            v[last] = v[first];
            v[first] = temp;
        }
        return;
    }
    // make the recursive call
    quicksort(v, first, pivotLoc-1);
    quicksort(v, pivotLoc+1, last);
}
```
Quick Sort – Best Case Analysis

• In general, at the $k^{th}$ level of recursion, there will be $2^k$ sub-arrays of size $N/(2^k)$.

  • Total cost of $\sum_{i=1}^{2^k} O(N/(2^k)) = O(N)$

  • This continues until $N/(2^k)$ has been reduced to 1.

• So, at each level all calls on sub-arrays of the same size add up to $O(N)$.

• Given, there are $\log(N)$ levels, so the best case is $O(N \log(N))$.

```c++
template <typename T>
void quicksort(vector<T>& v, int first, int last) {
  // index of the pivot
  int pivotLoc;
  // temp used for an exchange when vector has two elements
  T temp;
  // if the range is not at least two elements, return
  if (last - first < 1)
    return;
  // if sublist has two elements, compare v[first] and // v[last] and exchange if necessary
  else if (last - first == 1) {
    if (v[last] < v[first]) {
      temp = v[last];
      v[last] = v[first];
      v[first] = temp;
    }
    return;
  } else {
    pivotLoc = pivotIndex(v, first, last);
    quicksort(v, first, pivotLoc-1); // make the recursive call
    quicksort(v, pivotLoc+1, last); // make the recursive call
  }
}
```
Quick Sort – Average Case Analysis

• The average case for quick sort is $O(N \log (N))$. The proof of this is rather heavy going and out of the scope of this course.

```cpp
template <typename T>
void quicksort(vector<T>& v, int first, int last)
{
    // index of the pivot
    int pivotLoc;
    // temp used for an exchange when vector has two elements
    T temp;
    // if the range is not at least two elements, return
    if (last - first < 1)
        return;
    // if sublist has two elements, compare v[first] and v[last] and exchange if necessary
    else if (last - first == 1){
        if (v[last] < v[first]) {
            temp = v[last];
            v[last] = v[first];
            v[first] = temp;
        }
        return;
    }
    else {
        pivotLoc = pivotIndex(v, first, last);
        quicksort(v, first, pivotLoc-1); // make the recursive call
        quicksort(v, pivotLoc+1, last); // make the recursive call
    }
}
```
Quick Sort – Worst Case Analysis

- In the worst case, each call to pivotIndex would divide the array with one element on one side and all of the other elements on the other side.
- Then, for the recursive calls
  - One call returns immediately, but the other has almost as much work left to do as before we did the pivot.
  - The recursive calls go $N - 1$
- Hence, we get a total effort from these calls of:
  
  $O(N - 1) + O(N - 2) + \ldots + O(1) = O(\sum_{i=1}^{n-1} i) = O(N^2)$

```cpp
template <typename T>
void quicksort(vector<T>& v, int first, int last) {
    int pivotLoc;
    T temp;
    if (last - first < 1) return;
    else if (last - first == 1) {
        if (v[last] < v[first]) {
            temp = v[last];
            v[last] = v[first];
            v[first] = temp;
        }
    }
    else {
        pivotLoc = pivotIndex(v, first, last);
        quicksort(v, first, pivotLoc-1); // make the recursive call
        quicksort(v, pivotLoc+1, last); // make the recursive call
    }
}
```
Quick Sort – Choosing the Pivot

• Other choices that were once popular included choosing the first element or the last element of the range to serve as the pivot.
  • Poor choices, as the worst case would then occur whenever the array was already sorted or was in exactly reverse order.

• Choosing the midpoint is somewhat better.

• The most commonly recommended choices seem to be:
  • Select a position in the range to be sorted at random.
  • Examine the first, last, and midpoint values in the range to be sorted. Choose for the pivot whichever of these three values lies between the other two (in value, not position). This is called the median-of-three rule.
Quick Sort – Final Note

- Quick sort is a sorting algorithm with the optimal $O(N \log N)$ average case, but a $O(N^2)$ worst case.

- Despite its slower worst case behavior, the Quick sort is generally preferred over Merge sort when sorting array-like structures (including vectors and deques).
  - This is because the constant multiplier on quick sort’s average case is much smaller
  - If we are careful, we can make the worst case input of quick sort a relatively obscure case that would seldom be seen in practice.
Heap Sort
Binary Heaps

- Heap sort uses the Binary Heap structure for sorting.
- A binary heap is a binary tree (not a binary search tree) with the properties:
  - The tree is complete (entirely filled, except possibly on the lowest level, which is filled from left to right).
  - Each non-root node in the tree has a smaller (or equal) value than its parent. This binary heap is called a max-heap
    - $key(\text{parent}) \geq key(\text{child})$
    - the largest value in the heap will be at the root.
  - We can also have a min-heap, in which every child has a value larger than its parent
  - In this course, we will always assume that a “heap” is a “max-heap” unless explicitly stated otherwise.
Binary Heaps

• The parent of node $i$ is in slot $\left\lfloor \frac{i-1}{2} \right\rfloor$.
• The children of node $i$ are in $2i+1$ and $2i+2$.

Let’s consider a situation in which we have a “damaged” heap with one node out of position.
• The out-of-place node is too large (i.e., larger than its parent).
• The out-of-place node is too small (i.e., smaller than one or both of its children).

• **How do we “fix” the heap?**
Binary Heaps – Bubble Up

- When we have a node that is larger than its parent, we bubble it up by swapping it with its parent until it has reached its proper position.

```cpp
template<typename T>
void bubbleUp(vector<T>& heap, unsigned nodeToBubble){
    unsigned parent = (nodeToBubble - 1) / 2;
    while (node > 0 && heap[nodeToBubble] > heap[parent]){
        swap(heap[nodeToBubble], heap[parent]);
        nodeToBubble = parent;
        parent = (nodeToBubble - 1) / 2;
    }
}
```
Binary Heaps – Percolate Down

- When we have a node that is smaller than one or both of its children, we percolate it down by swapping it with the larger of its children.

```cpp
template<typename T>
void percolateDown(vector<T>& heap, unsigned nodeToPerc)
{
    while (2*nodeToPerc+1 < heap.size())
    {
        unsigned child1 = 2*nodeToPerc+1;
        unsigned child2 = child1 +1;
        unsigned largerChild = child1;
        if (child2 < heap.size() && heap[child2] > heap[child1])
            largerChild = child2;
        if (heap[largerChild] > heap[nodeToPerc])
            swap (heap[nodeToPerc], heap[largerChild]);
        else
            nodeToPerc = heap.size();
    }
}
```
Binary Heaps – Insertion

- The heap is a complete tree, so **new nodes** will continue to **go into the bottom level**, **filling it from left to right**.

- **If** the new value is out of position, it must be because it is larger than its parent → **Bubble it up**

- Example: Insertion of 54

```cpp
template<typename T>
void add_to_heap(vector<T>& heap, const T& newValue)
{
    // add newValue to complete tree
    heap.push_back(newValue);
    // repair the heap
    bubbleUp(heap, heap.size() - 1);
}
```
Binary Heaps – Deletion

• Pop or remove, from a heap, will remove the value that is currently in the root.
• To preserve structure, the rightmost node in the bottom level will replace the root.
• That new root value will almost certainly be out of position, being smaller than one or both of its children → Precolate Down
• Deletion of root (66)

```cpp
template<typename T>
void remove_from_heap (vector<T>& heap) {
    heap[0] = heap[heap.size() - 1]; // replace root
    heap.pop_back(); // remove the duplicate
    percolateDown(heap, 0); // repair the heap
}
```
Binary Heaps – Building a Heap

• A binary heap of $N$ nodes has a height of $\lceil \log(n) \rceil$

• Single insertion or deletion is of order $O(\log N)$ at worst case

• The average case for single insertion is $O(1)$

• Building a heap is in order of $O(N \log N)$.

• Since each percolate Down takes, in worst case, a time proportional to the height of the node being percolated,
  • The total time for build_heap is proportional to the sum of the heights of all the nodes in a complete tree.
  • It is possible to show that this sum is itself $O(N)$, where $N$ is the number of nodes in the tree.
  • Therefore build_heap is $O(N)$.

```cpp
template<typename T>
void build_heap(vector <T>& heap)
{
    unsigned i = (heap.size() - 1) / 2;
    do {
        percolateDown (heap , i);
        --i;
    } while ( i >= 0 ) ;
}
```
Heapsort

- With \texttt{pop_heap()}, the position last-1 winds up holding the former root of the heap (the largest element that had been in the heap).
- The first call here would put the largest element of the heap \( c \) in position \( c.\text{end}()\)-1.
- The second call would put the second largest element in \( c \) in position \( c.\text{end}()\)-2.
- The third call would put the third largest element in \( c \) in position \( c.\text{end}()\)-3.
- Keeping this up, we will eventually wind up having sorted all the elements in \( c \).
- A heap sort is really pretty simple. First we \textit{form the array into a heap}. Then we \textit{repeatedly pop the heap, collecting the successive maximum values} at the end of the container.
Heapsort - Analysis

- pop_heap runs in time proportional to the log of the number of elements in the heap, so the worst-case complexity of the call to is $O(\log (\text{last-first}))$

- The value of last changes each time around the loop.

- Let $N$ stand for the value of last-first when we first entered the heapsort function.
  - Each time around the loop, last - first $\leq N$.

- Obtaining an overly loose complexity bound, we can treat the body as $O(\log N)$

- Then, the loop reduces to $O(N \log N)$

- make_heap runs in time proportional to the number of elements $N$, so the complexity of that call is $O(N)$.

$\Rightarrow$ the entire algorithm is $O(N \log N)$
Heapsort – Final Note

- Heapsort has an advantage over the merge sort (which also has an $O(N \log N)$ worst case)
- Heapsort has a negligible memory overhead, while merge sort has $O(N)$ overhead.
- Heapsort has a better worst case complexity than quick sort, but experiment has shown that heapsort tends to be slower on average because it moves more elements than does quick sort.
- The sorting algorithm used in most implementations of the C++ `std::sort` function is an algorithm called the **introspective sort**.
- Introspective sorts combine two algorithms:
  - Heapsort, which has a worst-case and average-case complexity of $O(N \log(N))$.
  - Quicksort, which also has an average-case complexity of $O(N \log(N))$.
- An introspective sort starts as an ordinary quicksort, but monitors the size of the stack used to control the quicksort recursion. If the stack grows much larger than $\log(N)$, the sort switches over to the heapsort.
Questions?
Assignment #7

• Due Sun Nov 24th, 11:59pm

• Written Assignment
  • Ford & Topp, Chapter #12 & #14:

<table>
<thead>
<tr>
<th>Chapter 12</th>
<th>Chapter 14</th>
</tr>
</thead>
<tbody>
<tr>
<td>Question #</td>
<td>Page #</td>
</tr>
<tr>
<td>Q.28</td>
<td>P.722</td>
</tr>
<tr>
<td>Q.30</td>
<td>P.723</td>
</tr>
<tr>
<td>Q.24</td>
<td>P.852</td>
</tr>
</tbody>
</table>

• Given the following array: {30, 60, 20, 50, 40, 10}.
  • Using the Shell Sort, show the array after each iteration of the outer loop corresponding to a new Gap value.
  • Using the Quick Sort, show the array after each call of pivotIndex(). Assume the pivot index is calculated as (last+first)/2
Assignment #7

• Programming Assignment
  
  • Q1: The Bubblesort is another simple sorting algorithm, whose pseudocode is as follows:

  1. Write a C++ Bubblesort(vector<T> &v) method that takes a vector v and sort the elements in this vector v using Bubble Sort scheme.

  2. Write a C++ program that:
     • Generates 20 random numbers and stores them in a std::vector structure;
     • Sorts them using the Bubblesort algorithm;
     • Outputs them

  3. What is the worst-case time complexity of Bubblesort algorithm? Show how you derive the complexity.
Assignment #7

• Due Sun Nov 24th, 11:59pm

• Submission Format:

  • **Written Assignment**
    • Create single PDF file with name:
      cs361_assignment_7_<firstName>_<_lastName>
    • Have a cover page with your name and your email
    • Submit through Blackboard.

  • **Programming Assignment**
    • Make sure your two programs compile and execute using `g++` on Dept’s Linux machines.
    • Create a “Readme.txt” file for each program that list how to compile and execute the program. Include your name and your email.
    • Your main file should be named “main.cpp”.
    • You should have a Makefile.
    • Zip all files (.h, .c, Makefile, etc.) for each program and name them assign7_progam respectively.
    • Submit through Blackboard.

• **Final Submission Materials (2 Files):**
  • One PDF for written assignment.
  • One ZIP file for the program