Advanced Data Structures and Algorithms

CS 361 – Fall 2013

Lec. #09: Sorting

Tamer Nadeem
Dept. of Computer Science
Class Objective/Overview

• Understand *Declaration of Associative Container*

• Familiarize with *The Hash Class*

• Familiarize with *Balanced Search Trees - AVL*

• Familiarize with *Balanced Search Trees – B-Tree*

• Understand *2-3-4 Trees*

• Understand *Red-Black Trees*

• Understand *Insertion Sort and Its Worst Case Analysis*

• Understand *Shell Sort and Its Worst Case Analysis*
Following Up with Sets and Maps
Overview of Sets and Maps

• In a set or map, a given key value may **appear only once**
  • Adding a key K to a set replaces any existing key equal to K.
  • Adding a key-data (K, D1) to a map that has (K, D2) replaces (K, D2) with (K, D1)

• In a multiset or multimap, the same key can **occur any number of times**.
  • For a multiset we can now ask “how many K’s are in this set?
  • For a multimap, adding a key-data pair (K, D1) to a multimap that already has (K, D2) results in multimap that has both (K, D1) and (K, D2).
Implementation of STL “set”

• STL uses a red-black search tree.

• A red-black tree is a binary search tree that maintains balance between left and right subtrees of a node.

• The corresponding running time for red-black tree is always $O(\log_2 n)$

• STL implements ordered associated containers.
• Let’s assume we have the following `Object3D` class:

```cpp
class Object3D{
    . . .
    private:
        float x;
        float y;
        float z;
    }
```

• We want to declare a set container of `Object3D` items:

```cpp
Set<Object3D> v3DSet;
```

• To support set operations, the `Object3D` class should **overloads** the operators `==` and `<` by comparing the key fields in the operands.

```cpp
class Object3D{
    public:
        . . .
        bool operator== (const Object3D &lhs, const Object3D &rhs)
            return (x==rhs.x && y==rhs.y && z==rhs.z);
        }
        bool operator< (const Object3D &lhs, const Object3D &rhs)
            . . .
            }
    private:
        float x;
        float y;
        float z;
    }
```
Following Up with Hashing and Hash Tables
Hashing

- **Hashing** is an important approach to set/map construction.
- We’ve seen sets and maps with $O(N)$ and $O(\log N)$ search and insert operations.
- **Hash tables** trade off space for speed, sometimes achieving an average case of $O(1)$ search and insert times.
- Hash tables use a **hashing function** to compute an element’s position within the array that holds the table.
template <typename T, typename HashFunc>
class hash {
public:
    hash(int nbuckets, const HashFunc& hfunc = HashFunc());
    // constructor specifying the number of buckets in the hash table and the hash function
    hash(T *first, T *last, int nbuckets, const HashFunc& hfunc = HashFunc());
    // constructor with arguments including a pointer range [first, last) of values to insert, …
    bool empty() const; // is the hash table empty?
    int size() const; // return number of elements in the hash table
    iterator find(const T& item);
    // return an iterator pointing at item if it is in the table; otherwise, return end()
    const_iterator find(const T& item) const;
    // return a pair whose iterator component points at item and whose bool component is true,
    // otherwise bool is false.
    pair<iterator, bool> insert(const T& item);
    // if item is not in the table, insert it and return a pair whose iterator component points
    // at item and whose bool component is true, otherwise bool is false.
    int erase(const T& item); // if item is in the table, erase it and return 1; otherwise, return 0
    void erase(iterator pos); // erase the item pointed to by pos.
    void erase(iterator first, iterator last); // erase all items in the range [first, last).
    iterator begin(); // return an iterator positioned at the start of the hash table
    const_iterator begin() const; // constant version
    iterator end(); // return an iterator positioned past the last element of the hash table
    const_iterator end() const; // constant version
private:
    int numBuckets; // number of buckets in the table
    vector<list<T> > bucket; // the hash table is a vector of lists
    HashFunc hf; // hash function
    int hashtablesSize; // number of elements in the hash table
};
The Hash Class- Example

- A hash table stores objects of type \textit{employee} (\textit{ssn} is the key).
- The hash function object type \textit{hFemp} compute the hash value of an employee using his \textit{ssn}.

```cpp
class employee{
    public:
    employee(const string &snum, double sal):
        ssn(snum), salary(sal) {}

    //hash function object type
    private:
    string ssn;
    double salary;
};

//hash function object type for employee
class hFemp{
    public:
    unsigned int operator() (const employee &item) const{
        //calculate the hash value hValue of the employee
        . . .
        return hValue;
    }
};

// Declare a hash table with 157 buckets
// to store employee objects
Hash<employee, hFemp> hEmp(157);
...
The Hash Class - Iterator

// points to the hash table container
hash<T,HashFunc> *hashTable;

// index of current bucket being traversed
int currentBucket;

// points to the current element in the current bucket
typename list<T>::iterator currentLoc;

// find next non-empty bucket and set currentLoc
// to point at its first element
void findNext(){
    int i;

    // search from the next bucket to end of
    // table for a non-empty bucket
    for(i=currentBucket+1; i < hashTable->numBuckets; i++){
        if (!hashTable->bucket[i].empty()){
            // found a non-empty bucket. Set currentBucket
            // index to i and currentLoc to point at the first
            // element of the list
            currentBucket = i;
            currentLoc = hashTable->bucket[i].begin();
            return;
        }
    }
    currentBucket = -1;  // we are at end()
}

hash<int, hFintID> ht;
hash<int, hFintID>::iterator hIter;

hIter currentBucket=2
currentLoc

hf(x) = x

Hash iterator hIter referencing element 22 in table ht.

iterator operator++ (int){
    // move to the next data value or the end of the list
    currentLoc++;
    if (currentLoc ==
        hashTable->bucket[currentBucket].end())
        findNext();
    return *this;
}
Balanced Search Trees

AVL Trees
Balanced Trees

- BST operation performance is bounded by the height of the tree, which can range from an ideal of $O(\log N)$ (balanced trees) to $O(N)$ (degenerate trees).

- Various algorithms have been developed for building search trees that remain balanced. We’ll look at 2:
  - AVL trees
  - B trees

![Complete Tree](image1)

![Full / Perfect Tree](image2)
AVL Trees

• An AVL tree (Adelson-Velskii and Landis) is a binary search tree for which each node’s children differ in height by at most 1.
  • Guarantees that the height of the tree is $O(\log N)$.
  • Need to maintain height info in each node.

• AVL insertion starts out identical to normal binary search tree insertion.
  • After the new node has been created and put in place, each of its ancestors must check to see if still balanced.
  • In a balanced tree, this difference must be -1, 0, or 1. 0 means that both subtrees have the same height. -1 means that the left tree is higher (by 1), and 1 means that the right tree is higher.
  • If any are unbalanced, the balance is restored by a process called rotation.

```cpp
template <class T>
class avlNode
{
    public:
        ...
        T value ;
        avlNode<T> * parent ;
        avlNode<T> * left ;
        avlNode<T> * right ;
        short balanceFactor ;
};
```
AVL Trees Example

- Insert operation may cause balance factor to become 2 or –2 for some node
  - only nodes on the path from insertion point to root node have possibly changed in height
  - So after the Insert, go back up to the root node by node, updating heights
  - If a new balance factor (the difference $h_{\text{left}} - h_{\text{right}}$) is 2 or –2, adjust tree by rotation around the node
Insertions in AVL Trees

• Let the node that needs rebalancing be \( \alpha \).

• There are 4 cases:
  
  *Outside Cases* (require single rotation):
  1. Insertion into left subtree of left child of \( \alpha \).
  2. Insertion into right subtree of right child of \( \alpha \).

  *Inside Cases* (require double rotation):
  3. Insertion into right subtree of left child of \( \alpha \).
  4. Insertion into left subtree of right child of \( \alpha \).

• The rebalancing is performed through four separate rotation algorithms.
AVL Insertion: Outside Case

Consider a valid AVL subtree

Inserting into X destroys the AVL property at node j

Do a “right rotation”

“Single Right rotation” done!

AVL property has been restored!
Example - Single Rotation

Before Rotation:

```
    1
   /|
  4 6
 / | \
5  1 8
```

After Rotation:

```
    1
   /|
  4 6
 / | \
8  9
```

Before Rotation:

```
    1
   /|
  4 6
 / | \
5  8
```

After Rotation:

```
    1
   /|
  4 6
 / | \
8  9
```

Implementing AVL Single Rotation

template <class T>
avlNode<T>* avlNode<T>::singleRotateLeft ()
// perform single rotation rooted at current node
{
    avlNode<T>* U = this;
    avlNode<T>* H = U->right;
    avlNode<T>* I = H->left;

    U->right = I;
    H->left = U;
    if ( I != 0 )
        I->parent = U;
    H->parent = U->parent;
    U->parent = H;
    // now update the balance factors
    int Ubf = U->balanceFactor;
    int Hbf = H->balanceFactor;
    if (Hbf <= 0) {
        if (Ubf >= 1)
            H->balanceFactor = Hbf - 1;
        else
            H->balanceFactor = Ubf + Hbf - 2;
    } else {
        if (Ubf <= Hbf)
            H->balanceFactor = Ubf - 2;
        else
            H->balanceFactor = Hbf - 1;
    }
    U->balanceFactor = (Ubf - Hbf) - 1;
    return H;
}
AVL Trees Complexity

• An AVL tree is balanced, so its height is $O(\log N)$ where $N$ is the number of nodes.

• The rotation routines are all themselves $O(1)$

• Insertion into an AVL tree has a worst case $O(\log N)$.

• Searching an AVL tree is completely unchanged from BST’s, and takes time proportional to the height of the tree, making $O(\log N)$.

• Removing nodes from a binary tree also requires rotations, but remains $O(\log N)$ as well.
Balanced Search Trees

B-Trees
B-Trees

• B-trees are a form of balanced search tree based upon general trees.
• A B-tree node can contain several data elements, rather than just one as in binary search trees.
• They are especially useful for search structures stored on disk. Disks have different retrieval characteristics than internal memory (RAM).
  • Obviously, disk access is much much slower.
  • Furthermore, data is arranged in concentric circles (called tracks) on each side of a disk “platter”.
• B-trees are a good match for on-disk storage and searching because we can choose the node size to match the cylinder (multiple parallel tracks) size.
• In doing so, we will store many data members in each node, making the tree flatter, so fewer node-to-node transitions will be needed.
B-Trees

For a B-tree of order $m$:

- All **data is in leaves.** **Keys** (only) can be **replicated** in interior nodes.

- The root is either
  - a leaf, or
  - an interior node with $2 \ldots m$ children

- All **interior nodes** other than the root have $\lfloor m/2 \rfloor \ldots m$ children

- All **leaves** are at the **same depth**.

Example of a B-tree of order 4.
B-Tree Insertion

- Inserting into a B-tree starts out by "find"ing the leaf in which to insert.
  - If there is room in the leaf for another data item, then we’re done.
  - If the leaf already has m items, then there’s no room.
  - **Split** the overfull node in half and pass the middle value up to the parent for insertion there.
  - If the value passed up to the parent causes the parent to be over-full, then it too splits and passes the middle value up to its parent.

- Deletion is usually lazy or semi-lazy (delete from leaf but **do not remove keys** within the interior nodes).

```cpp
BTreeNode<T>* find (const T& x, BTreeNode<T> *t)
{
    if ( t is a leaf )
        return t;
    else
    {
        i = 1;
        while ( (i < m) && (x >= t->key[i]) )
            ++i;
        return find (x, t->child[i]) ;
    }
}
```
Balanced Search Trees

2-3-4 Trees
2-3-4 Trees

- B-trees are generally used with a fairly high width (order).
- A closely related data structure arises when we take a B-tree of order 4 and relax just a few rules, including not storing all the data in the leaves but allowing some data to reside in the internal tree nodes.
- The result is called a 2-3-4 tree because each non-leaf node will, depending upon how full it is, have either 2, 3, or 4 children.
2-3-4Trees

- Example:

- Node Split:
2-3-4Trees Insertion

- Insertion Sequence: 2, 15, 12, 4, 8, 10, 25, 35, 55, 11, 9, 5, 7

Insert 2

Insert 15

Insert 12

Split 4-node (2, 12, 15)

Insert 4

Insert 8

Insert 10

Insert 25

Split 4-node (15, 25, 35)

Insert 35

Insert 55
2-3-4Trees Insertion

- Insertion Sequence:  2, 15, 12, 4, 8, 10, 25, 35, 55, 11, 9, 5, 7
Balanced Search Trees
Red-Black Trees
Red-Black Trees

• There is a fairly simple way to map **2-3-4 trees onto binary trees** to which a "color" has been added

```cpp
class RedBlackNode
{
    public:
        <[ ]> T value;
        RedBlackNode<T> * parent;
        RedBlackNode<T> * left;
        RedBlackNode<T> * right;
        bool color; // true=red, false=black
};
```
Red-Black Trees

- A **2-3-4 node with 2 children** (1 data value) is represented by a **black binary tree node** whose children are either leaves or black nodes.

- A **2-3-4 node with 3 children** (2 data values) is represented by a **black binary tree node with one red child**, the other child being a leaf or a black node. (Either child could be the red one, so the mirror image of the binary tree in this diagram is also legal.)

- A **2-3-4 node with 4 children** (3 data values) is represented by a **black binary tree node with two red children**.
Red-Black Trees

- Here is an example of the red-black equivalent to a 2-3-4 search tree.
Red-Black Trees - Example
Red-Black Trees

Some things to note:

• The root of a red-black tree is always black.

• No red node will ever have a red child.

• The red-black tree is a binary search tree and can be searched using the conventional binary search tree "find" algorithm.

• The height of a red-black tree is no more than twice the height of the equivalent 2-3-4 tree.
  • And we have already noted that the height of B-trees, including 2-3-4 trees, is $O(\log N)$ where $N$ is the number of data items in the tree.
  • We therefore know that the height of a red-black tree is also $O(\log N)$.
  • And that searches on a red-black tree have a $O(\log N)$ worst case.
Red-Black Trees

- The algorithms to insert nodes into a red-black tree add no more than a constant time for each node in the path from the root to the newly added leaf.
- Insertions into a red-black tree are worst case $O(\log N)$.
  - code for red-black trees re based on rotations very similar to those of AVL trees.
- Red-black trees are used in most implementations of `set`, `mset`, `map`, and `mmap` in the C++ std library.
Sorting
std Containers

• **Sorting**: given a sequence of data items in an unknown order, *re-arrange* the items to put them into *ascending* (descending) order by key.

• Sorting algorithms have been studied extensively.
  • *No one best* algorithm for all circumstances
  • The *big-O behavior is a key* to understanding where and when to use different algorithms.
Insertion Sort
Insertion Sort

• The insertion sort *divides* the list of items into a *sorted and an unsorted regions*, with the sorted items in the first part of the list.

• *Idea:* Repeatedly take the *first item from the unsorted region* and *insert it into the proper position in the sorted portion* of the list.
Insertion Sort - Algorithm

• At the beginning of each outer iteration, items 0 . . . i-1 are properly ordered.

• Each outer iteration seeks to insert item v[i] into the appropriate position within 0 . . . i.

```cpp
template <typename T>
void insertionSort ( vector<T>& v )
{
    int i, j, n=v.size();
    T target;
    // place v[i] into the sublist v[0] . . . v[i-1],
    // 1 <= i < n, so it is in the correct position
    for ( i=1; i<n; i++ )
    {
        // index j scans down list from v[i] looking for
        // correct position to locate target. assigns it to v[j]
        j = i;
        target = v[i];
        // locate insertion point by scanning downward as long
        // as target < v[j-1] and we have not encountered the
        // beginning of the list
        while ( j > 0 && target < v[j-1] )
        {
            // shift elements up list to make room for insertion
            v[j] = v[j-1];
            j--;
        }
        // the location is found; insert target
        v[j] = target;
    }
}
```
Insertion Sort – Worst Case Analysis

• Assume comparisons & copying are \( O(1) \).

```cpp
template <typename T>
void insertionSort ( vector<T>& v )
{
    int i, j, n=v.size( ) ;          // O(1)
    T target;                   // O(1)
    // place v[i] into the sublist v[0] . . . v[i-1],
    // 1 <= i < n, so it is in the correct position
    for ( i=1; i<n; i++ )
    {
        // index j scans down list from v[i] looking for
        // correct position to locate target. assigns it to v[j]
        j = i ;                      // O(1)
        target = v[i] ;              // O(1)
        // locate insertion point by scanning downward as long
        // as target < v[j-1] and we have not encountered the
        // beginning of the list
        while ( j > 0 && target < v[j-1] )
        {
            // shift elements up list to make room for insertion
            v[j] = v[j-1];            // O(1)
            j--;                     // O(1)
        }
        // the location is found; insert target
        v[j] = target;            // O(1)
    }
}
```
Insertion Sort – Worst Case Analysis

• Loop: work Inside to Outside

• Looking at the inner loop

• In the worst case, how many times do we go around the inner loop?

Answer: \( i \) times

• What is the complexity of the inner loop?

Answer: The body and condition are \( O(1) \), and the loop executes \( i \), \( \rightarrow \) entire loop is \( O(i) \)

template &lt;typename T&gt;
void insertionSort ( vector &lt;T&gt; &amp; v )
{
    int i, j, n=v.size( ) ;    // O(1)
    T target;  // O(1)
    // place v[ i ] into the sublist v[0] . . . v [i -1],
    // 1 <= i < n, so it is in the correct position
    for ( i=1; i<n; i++)
    {
        // index j scans down list from v[i] looking for
        // correct position to locate target. assigns it to v [j]
        j = i ;      // O(1)
        target = v[i] ;  // O(1)
        // locate insertion point by scanning downward as long
        // as target < v[j-1] and we have not encountered the
        // beginning of the list
        while ( j > 0 &amp;&amp; target < v[j-1])   // O(i)
            {
                // shift elements up list to make room for insertion
                v[j] = v[j-1];     // O(1)
                j--;               // O(1)
            }
        // the location is found; insert target
        v [j] = target;    // O(1)
    }
}
Insertion Sort – Worst Case Analysis

• Looking at the outer loop
  
  • The entire outer loop body is \( O(i) \).
  
  • The outer loop executes \( (n-1) \) times.

• What is the complexity of the entire outer loop?

  **Answer:** The general rule for loops is to sum up the cost of all loop iterations:

  \[
  \sum_{i=0}^{n-1} O(i) = O\left( \sum_{i=0}^{n-1} i \right) = O\left( \frac{n(n-1)}{2} \right) = O(n^2)
  \]
Insertion Sort – Worst Case Analysis

• Then, what is the complexity of the entire function?

A proper answer would be that this function is \( O(v.size()^2) \).

Or, we could say that:

*Insertion sort has a worst case of \( O(N^2) \) where \( N \) is the size of the input vector*.

```cpp
template <typename T>
void insertionSort ( vector <T>& v )
{
    int i, j, n=v.size( ) ;       // O(1)
    T target;                     // O(1)
    // place v[ i ] into the sublist v[0] . . . v [i-1],
    // 1 <= i < n, so it is in the correct position
    for ( i=1; i<n; i++)          // O(n^2)
    {
        // index j scans down list from v[i] looking for
        // correct position to locate target. assigns it to v [j]
        j = i ;                      // O(1)
        target = v[i] ;             // O(1)
        // locate insertion point by scanning downward as long
        // as target < v[j-1] and we have not encountered the
        // beginning of the list
        while ( j > 0 && target < v[j-1]) // O(i)
        {
            // shift elements up list to make room for insertion
            v[j] = v[j-1];         // O(1)
            j--;                  // O(1)
        }
        // the location is found; insert target
        v [j] = target;          // O(1)
    }
}
```
Insertion Sort – Special/Best Case

- Consider the behavior of this algorithm when applied to an array that is already sorted.
  - we never enter the body of the inner loop.
- The inner loop is then \( O(1) \).
- The insertionSort is \( O(v.size()) \) or \( O(N) \).

```cpp
template <typename T>
void insertionSort ( vector <T>& v )
{
    int i, j, n=v.size( ) ;       // O(1)
    T target;                     // O(1)
    // place v[ i ] into the sublist v[0] . . . v [ i -1],
    // 1 <= i < n, so it is in the correct position
    for ( i=1; i<n; i++)         // O(n)
    {
        // index j scans down list from v[i] looking for
        // correct position to locate target. assigns it to v [j]
        j = i ;                     // O(1)
        target = v[i] ;             // O(1)
        // locate insertion point by scanning downward as long
        // as target < v[j-1] and we have not encountered the
        // beginning of the list
        while ( j > 0 && target < v[j-1]) // O(1)
        {
            // shift elements up list to make room for insertion
            v[j] = v[j-1];              // O(1)
            j--;                       // O(1)
        }
        // the location is found; insert target
        v [j] = target;             // O(1)
    }
}
```
Shell Sort
Shell Sort

• Any sorting algorithm that only swaps adjacent elements has average time no faster than $O(n^2)$.

• The obvious way around this limitation is to compare and, when necessary, exchange distant objects.
  • The Shell sort (named for its inventor, Donald Shell) is an early attempt

• Shell sort uses a sequence $h_1, h_2, \ldots, h_t$ called the increment sequence.
  • Any increment sequence is fine as long as $h_1=1$

• The "middle" of a shell sort looks like insertion sort, but
  • Instead of comparing $i^{th}$ element to $i-1, i-2, i-3, \ldots$ we compare to $i-h_k, i-2h_k, i-3h_k, \ldots$

• The outer loop of Shell sort decreases $h_k$ (Gap), eventually to 1

• At end of any "phase", we have $a[i] \leq a[i+h_k]$
  • Elements spaced $h_k$ apart are sorted ($h_k - sorted$)
Shell Sort - Example

- Shell sort improves on the efficiency of insertion sort by quickly shifting values to their destination.

- The distance between comparisons decreases as the sorting algorithm runs until the last phase in which adjacent elements are compared.

Sort: 18 32 12 5 38 33 16 2

8 Numbers to be sorted, Shell’s increment will be \( \text{floor}(n/2) \)

* \( \text{floor}(8/2) \Rightarrow \text{floor}(4) = 4 \)

increment 4:

1 2 3 4

18 32 12 5 38 33 16 2

Step 1) Only look at 18 and 38 and sort in order; 18 and 38 stays at its current position because they are in order.

Step 2) Only look at 32 and 33 and sort in order; 32 and 33 stays at its current position because they are in order.

Step 3) Only look at 12 and 16 and sort in order; 12 and 16 stays at its current position because they are in order.

Step 4) Only look at 5 and 2 and sort in order; 2 and 5 need to be switched to be in order.
Shell Sort - Example

- Sort: 18  32  12  5  38  33  16  2

Resulting numbers after increment 4 pass:

18   32   12   2   38   33   16   5

* floor(4/2) \(\Rightarrow\) floor(2) = 2

increment 2:

1   2

18   32   12   2   38   33   16   5

Step 1) Look at 18, 12, 38, 16 and sort them in their appropriate location:

12   38   16   2   18   33   38   5

Step 2) Look at 32, 2, 33, 5 and sort them in their appropriate location:

12   2   16   5   18   32   38   33
Shell Sort - Example

- Sort: 18  32  12  5  38  33  16  2

Resulting numbers after increment 2 pass:

12  2  16  5  18  32  38  33

* floor(2/2) \[\Rightarrow\] floor(1) = 1

Increment 1:

1

12  2  16  5  18  32  38  33

Step 1) The last increment or phase of Shell sort is basically an Insertion sort algorithm.

2  5  12  16  18  32  33  38
Shell Sort - Algorithm

• Note that, if Gap==1,
  • the inner two loops of the Shell sort are simply an “ordinary” insertion sort.

• If Gap==2, have something very similar to an insertion sort, but:
  • array elements in even numbered positions are only compared to other elements in even numbered positions;
  • elements in odd-numbered positions are compared to other elements in odd-numbered positions.

```cpp
// Shellsort: sort first N items in array A
// T: must have copy constructor, operator=, and operator<
template <class T>
void shellsort(T a[], int n)
{
    for (int Gap = n / 2; Gap > 0; Gap = Gap/2)
    {
        //inv: for all i in Gap..n-1, a[i] >= a[i-Gap]
        for( int i = Gap; i < n; i++ )
        {
            T Tmp = a[i];
            int j = i;
            while (j >= Gap && Tmp < a[j - Gap])
            {
                a[j] = a[j - Gap];
                j -= Gap;
            }
            a[j] = Tmp;
        }
    }
}
```
Shell Sort - Algorithm

• The speed comes from the fact that:

• Most of the inner loops executions exit immediately, or after only a single swap

• Because the larger-Gapped phases have already moved the elements close to where they belong

```cpp
// Shellsort: sort first N items in array A
// T: must have copy constructor, operator=, and operator<
template <class T>
void shellsort(T a[], int n)
{
    for (int Gap = n / 2; Gap > 0;  Gap = Gap/2)
    {
        //inv: for all i in Gap..n-1, a[i] >= a[i-Gap]
        for( int i = Gap; i < n; i++ )
        {
            T Tmp = a[i];
            int j = i;
            while (j >= Gap && Tmp < a[j - Gap])
            {
                a[j] = a[j - Gap];
                j -= Gap;
            }
            a[j] = Tmp;
        }
    }
}
```
Shell Sort – Worst Case Analysis

- The two statements in the body of the inner loop are \( O(1) \). So is the condition of the innermost loop.

- How many times does the inner loop repeat?

  **Answer:** The innermost loop starts \( j \) at \( i \), decreases \( j \) by \( \text{Gap} \) each around, and continues until \( j \) has been reduced to \( \text{Gap} \).

  \[ \rightarrow \text{answer is } (i - \text{Gap})/\text{Gap}. \]
Shell Sort – Worst Case Analysis

- Looking at the middle loop
- The entire middle loop body is $O(i/\text{Gap})$.
- The outer loop executes $(n-\text{Gap})$ times.

What is the complexity of the entire middle loop?

Answer: The general rule for loops is to sum up the cost of all loop iterations:

$$O\left( \sum_{i=\text{Gap}}^{n-1} \frac{i}{\text{Gap}} \right)$$

$$= O\left( \frac{1}{\text{Gap}} \sum_{i=\text{Gap}}^{n-1} i \right) = O(n^2 / \text{Gap})$$

```cpp
// Shellsort: sort first N items in array A
// T: must have copy constructor, operator=, and operator<
template <class T>
void shellsort(T a[], int n)
{
    for (int Gap = n / 2; Gap > 0; Gap = Gap/2)
    {
        //inv: for all i in Gap..n-1, a[i] >= a[i-Gap]
        for( int i = Gap; i < n; i++ )  // O(n^2/Gap)
        {
            T Tmp = a[i];
            int j = i;
            while (j >= Gap && Tmp < a[j - Gap])  // O(i/Gap)
            {
                a[j] = a[j - Gap];
                j -= Gap;
            }
            a[j] = Tmp;
        }
    }
}
```
Shell Sort – Worst Case Analysis

• Looking at the Outer loop
• The entire middle loop body is $O(n^2/Gap)$.
• How many times the outer loop executes?
  • Assume, for simplicity, that $n$ is an exact power of 2.
  • the outer loop executes $\log_2(n)$ times.
• What is the complexity of the entire middle loop?
  • Gap will take on values 1, 2, 4, $\ldots$, $2^{\log(n)-1}$
  • Total effort is: $O\left(\sum_{i=0}^{\log n-1} \frac{n^2}{2^i}\right)$.
  $\Rightarrow O\left(n^2 \sum_{i=0}^{\log n-1} \frac{1}{2^i}\right)$.
  • Given: $\sum_{i=0}^{\log n-1} \frac{1}{2^i} < 2$
  $\Rightarrow O\left(n^2 \sum_{i=0}^{\log n-1} \frac{1}{2^i}\right) = O(2n^2) = O(n^2)$

```cpp
// Shellsort: sort first N items in array A
// T: must have copy constructor, operator=, and operator<
template <class T>
void shellsort(T a[], int n)
{
    for (int Gap = n / 2; Gap > 0;  Gap = Gap/2) // O(n^2)
    {
        //inv: for all i in Gap..n-1, a[i] >= a[i-Gap]
        for( int i = Gap; i < n; i++ ) // O(n^2/Gap)
        {
            T Tmp = a[i];
            int j = i;
            while (j >= Gap && Tmp < a[j - Gap]) // O(i/Gap)
            {
                a[j] = a[j - Gap];
                j -= Gap;
            } // O(1)
            a[j] = Tmp;
        } // O(1)
    }
}
```
Shell Sort – Other Increment Sequences

• Different increment sequences can yield better results.

• Looking at increment sequences proposed by some other designers:
  
  • **Hibbard**: 1, 3, 7, . . . , 2^{k-1}
    
    • Worst case: $O(N^{3/2})$
    • Average (unproven): $O(N^{5/4})$

  • **Sedgewick**: 1, 5, 19, 41, 109, . . .
    
    • Worst case: $O(N^{4/3})$
    • Average (unproven): $O(N^{7/6})$

• Many of these results have been obtained via experimentation because no one has been able to prove them.

• Oddly enough, the following appears to work about as well as anything: Divide by 2.2
Questions?