Internal Sorting

1. Introduction

- Given a set of $n$ records $r_1, r_2, \ldots, r_n$ with search keys $k_1, k_2, \ldots, k_n$.

  Goal: Rearrange records into $r_{j1}, r_{j2}, \ldots, r_{j_n}$ such that $k_{j1} \leq k_{j2} \leq \ldots \leq k_{j_n}$

- **Use internal sorting**: When all records can fit into the main memory at the same time

- **Use external sorting**: When all records cannot fit into the main memory at the same time. Usually, records are in large files, which are stored in the disk, need to consider internal and external issues.

- **Input**: Assume each record is a structure with a key field. There are $n$ records (structures) in an array $A$.

- **Output**: Array $A$ with sorted records (in non-decreasing order)
2. Selection sort

- Algorithm:

```c
// compare remaining keys in A[j..n] to select the next smallest key
// A[j] is in place after each pass

for (j=0; j<n-1; j++)
{
    smallest_index = j
    for (k=j+1; k<= n-1; k++)
        if (A[smallest_index].key > A[k].key) smallest_index = k
    SWAP(A[j], A[smallest_index])
}
```

- Example:

<table>
<thead>
<tr>
<th>Index</th>
<th>Initial</th>
<th>1st loop</th>
<th>2nd loop</th>
<th>3rd loop</th>
<th>4th loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>13</td>
<td>-10</td>
<td>-10</td>
<td>-10</td>
<td>-10</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-5</td>
<td>-5</td>
<td>-5</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>-5</td>
<td>-5</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>-10</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
</tr>
</tbody>
</table>

- Analysis

Best case running time :  \(O(n^2)\)
Worst Case Running time :  \(O(n^2)\)
Average Case Running time :  \(O(n^2)\)
3. Bubble sort

- Algorithm

```
// compare adjacent keys from the remaining keys in A[0, n-j-1]
// and exchange them if they are out of order
// one record, A[n-pass], is in place in every pass

sorted = false
for (pass=1; (pass < n) && !sorted; pass++)
{
    sorted = true
    for (k=0; k<n-pass; k++)
        if (A[k].key > A[k+1].key)
            SWAP(A[k], A[k+1])
            sorted = false
}
```

- Example:

<table>
<thead>
<tr>
<th>Index</th>
<th>Initial</th>
<th>1st loop</th>
<th>2nd loop</th>
<th>3rd loop</th>
<th>4th loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>13</td>
<td>1</td>
<td>1</td>
<td>-5</td>
<td>-10</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>-5</td>
<td>-10</td>
<td>-5</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>-5</td>
<td>-10</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>-5</td>
<td>-10</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>-10</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
</tr>
</tbody>
</table>

- Analysis

Best case running time: \(O(n)\) // input a sorted list
Worst Case Running time: \(O(n^2)\)
Average Case Running time: \(O(n^2)\) // sort half of the list
4. Insertion sort

- **Algorithm**

  // at the beginning of j\(^{th}\) pass, j elements in A[0...j-1] are already sorted,
  // insert new element (initially in A[j] position) into the proper location
  // in A[0…j] so that j+1 elements in A[0…j] are sorted at the end of j\(^{th}\) pass

  for (j=1; j<=n-1; j++)
  {
    currentRecord = A[j];

    // shift all elements in A[loc…j-1] that are > j\(^{th}\) element to
    // A[loc+1…j]. All elements in A[0…loc –1] are <= j\(^{th}\) element

    loc = j;
    for ( ; loc > 0 & & (currentRecord.key < A[loc-1].key); loc--)

    // set j\(^{th}\) element in new location
    A[loc] = currentRecord;
  }

- **Example** :

<table>
<thead>
<tr>
<th>Index</th>
<th>Initial</th>
<th>1(^{st}) loop</th>
<th>2(^{nd}) loop</th>
<th>3(^{rd}) loop</th>
<th>4(^{th}) loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>13</td>
<td>1</td>
<td>1</td>
<td>-5</td>
<td>-10</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>13</td>
<td>2</td>
<td>1</td>
<td>-5</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>13</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>-5</td>
<td>-5</td>
<td>-5</td>
<td>13</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>-10</td>
<td>-10</td>
<td>-10</td>
<td>-10</td>
<td>13</td>
</tr>
</tbody>
</table>

- **Analysis**

  Best case running time : O(n)  // input a sorted list
  Worst Case Running time : O(n^2)
  Average Case Running time : O(n^2)
5. Merge sort

- Algorithm // divide and conquer strategy

  // merge sort is a recursive function. it recursively divides a list of
  // elements into two equal size sub-lists until a list contains an element,
  // then use merge() to merge with adjacent list
  mergesort(inout A:itemArray, in first:integer, in last:integer)
  {
    if (first < last)
    {
      mid = (first+last)/2
      mergesort(A, first, mid);
      mergesort(A, mid+1, last)
      merge(A, first, mid, last)
    }
  }

  // Input two sorted lists, A[first…mid] and A[mid+1 .. last].
  // Output a sorted list A[first…last]. Need to use temporary
  // storage array B[first…last] to store the result, then copy B[] into A[]
  merge(inout A:itemArray,in  first:integer, in mid:integer, in last:integer)
  {
    i=first;     // index of 1st list
    j=mid+1;    // index of 2nd list
    k=first;    // index of B

    while( i <= mid && j <= last)
    {
      if (A[i].key <= A[j].key)
      {
        B[k] = A[i]; i++;
      } else {
        B[k] = A[j]; j++;
      }
      k++;
    }
// one of two lists is done.
// copy remaining elements in uncompleted list into B[]
// copy B[] into A[]

• Example:

(10 14 8 20 17 15 9 88)

(10 14 8 20) (17 15 9 88)

(10 14) (8 20) (17 15) (9 88)

(10) (14) (8) (20) (17) (15) (9) (88)

(8 10 14 20) (9 15 17 88)

(8 9 10 14 15 17 20 88)
8. Internal Sorting

- **Analysis**

  merge : \( O(\text{total size of two-sorted list}) \)

  mergesort : Always divides the list into half
  Total number of passes \( O(\log n) \)
  In each pass, merge total \( n \) keys (may merge() multiple times)
  Total running time : \( O(\log n) \times n = O(n \log n) \)

6. **Heap sort**

- heapsort() will be covered in the future.
- skip analysis (should cover in csc510), running time : \( O(n \log n) \)

7. **Quicksort**

- Use **divide and conquer** technique :
  - divide problem into subproblems,
  - conquer subproblems by solving them recursively and
  - combine subproblem solutions to obtain the desired solution

- Divide : partition array elements using a pivot element \( p \) into subarrays, \( S_1 \) and \( S_2 \).

  
<table>
<thead>
<tr>
<th>S1</th>
<th>( p )</th>
<th>S2</th>
</tr>
</thead>
</table>

  All elements in \( S_1 < p \), all elements in \( S_2 \geq p \), and \( p \) is in correct position.

- Conquer : recursively sort two subarrays \( S_1 \) and \( S_2 \)
- Combine : don’t have to combine the solution
- Given \( k \) elements in array, partition takes \( O(k) \)
• Partition Algorithm

Partition(inout A: itemArray, in first:integer,in last:integer,
    out pivotIndex:integer) // different from the text


Output: partition A[first…last] into regions S1 : A[first…pivotIndex-1] and
    S2 : A[pivotIndex+1…last] that satisfies the above condition.
    A[pivotIndex] element is in-place.

p = A[last] // pivot element
i = first-1  // last element in S1

for (j = first to last-1)
{
    if A[j] < p then
        { i = i + 1
        swap(A[i],A[j])
        }
    swap(A[i+1],A[last])
pivotIndex = i+1

• Example :

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>first = 0, last = 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>A values:</td>
<td>7</td>
<td>8</td>
<td>2</td>
<td>6</td>
<td>5</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>p = 4, i = -1</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>8</td>
<td>2</td>
<td>6</td>
<td>5</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>After j = 0</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>8</td>
<td>2</td>
<td>6</td>
<td>5</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>After j = 1</td>
</tr>
<tr>
<td>swap()</td>
<td>2</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>After j = 2, i = 0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>After j = 3</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>After j = 4</td>
</tr>
<tr>
<td>swap()</td>
<td>2</td>
<td>1</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>8</td>
<td>3</td>
<td>4</td>
<td>After j = 5, i = 1</td>
</tr>
<tr>
<td>swap()</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>5</td>
<td>8</td>
<td>7</td>
<td>4</td>
<td>After j = 6, i = 2</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>swap(A[3],A[7]), pivotIndex = 3</td>
</tr>
</tbody>
</table>
• Analysis Partition: Running time O(last-first+1) // Size of the input array A

• Quicksort Algorithm

QuickSort(inout A:itemArray,in first:integer,in last:integer)
{
    if (first < last)
    {
        partition(A, first, last, pivotIndex)
        QuickSort(A, first, pivotIndex-1);
        QuickSort(A, pivotIndex+1, last)
    }
}

• Analysis Quicksort

Worst case: Partition() always splits the array into 0 and n-1 elements in subarrays. (e.g. original array is sorted). We have

\[ T(n) = O(n) + T(n-1) + T(0) \]  // one element in place after 1st partition
\[ = O(n) + O(n-1) + T(n-2) \]  // T(0) is constant, can be eliminate
\[ ... \]
\[ = O(n) + O(n-1) + ... + O(1) \]
\[ = O(n^2) \]

Best case: if we are lucky, Partition() always splits the array evenly. We have

\[ T(n) = O(n) + 2T(n/2) \]
\[ = O(n) + O(n) + 4T(n/4) \]  // total running time of 2 partitions
\[ = O(n \log n) \]  // of size n/2 is O(n)
\[ = O(n \log n) \]  // \( i = \log n \) for \( (n/2^i) = 1 \)

Average case: also O(n log n)  // analyze this in csc510
8. General issues

- Comparison sorting: using comparisons of elements to obtain the sorted order.

- So far, we have covered several comparison sorting algorithms.

- The worst case performance: $O(n \log n)$ or $O(n^2)$

- Can we do better?

  What is the lower bound for comparison sorting methods? i.e. Given the best comparison sorting method, what is the worst case running time?

  Clearly: at least $O(n)$ since we need comparisons for $n$ elements.

  Can we get lower bound of $O(n)$?

- Theorem: The best performance of a comparison sorting method is $O(n \log n)$

- Therefore, heapsort, mergesort and quicksort are asymptotically optimal comparison sorts.

- So, if there any **linear time**, i.e. $O(n)$, non-comparison sorting method? Answer: Yes.
9. Counting Sort

- depends on assumption about the keys to be sorted (i.e. integers in the range of 1 to k)

- Algorithm

CountingSort (in A:itemArray, out B:itemArray, in k:integer)

Input : array A with n elements, where each A[i] is a number in \{1,2,...,k\}
Output : sorted list in array B

/* need additional temporary storage : array C[1..k] */

step 1 : for i = 1 to k    set C[i] = 0  O(k)

step 2 : for j = 1 to n   /* count # elements equal to i using array C */  O(n)
    C[A[j]] = C[A[j]]+1

step 3 : for j = 2 to k     /* count # of elements <= j */  O(k)
    C[j] = C[j]+ C[j-1]

step 4 : for j = n downto 1 /* compute sorted list in array B */  O(n)
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]]-1

- Example :

<table>
<thead>
<tr>
<th>Index</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3</td>
<td>6</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>k = 6</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>after step 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>after step 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>7</td>
<td>7</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>after step 3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>
• Counting sort is a **stable algorithm**, i.e. input order maintained among elements with same number.

• Running time $T(n) = O(k+n) = O(n)$ if $k = O(n)$

• If $k$ is very large $k >> n$ then counting sort is not useful. Use Radix sort

10. Radix sort

• Punched card readers for census tabulation in the early 1900’s (by IBM).

  Goal : To sort the cards in order

  Each punched card has 80 columns, each column has 12 places (*0 1 3 4 etc)

  Punched card reader : machine that can sort one column of cards at a time. i.e. divide cards into 12 bins.

  So, for a specific column, use punched card reader to divide cards into bins, combine cards by using cards in bin 0 preceding the cards in bin 1 etc

  For all 80 columns, need to design good algorithm to sort all cards

• Let assume there are only 10 places (0 1 2 ...9) and $d$ columns

  /* may think : integer numbers with at most $d$ digits */

  Radix Sort Algorithm :

  ```
  for i = d to 1
      use a stable algorithm to sort array on $i^{th}$ digit
  /* if it is not a stable algorithm then we may get wrong output */
  ```
• Example:

```
329  720  720  329
457  355  329  355
657 3^{th} 436 2^{nd} 436 1^{st} 436
839 => 457 => 839 => 457
436 657 355 657
720 329 457 720
355 839 657 839
```

• Running time: each pass $O(n+k)$ /* $k = 10$, small number */
  total: $O(dn + dk)$
  when $d$ is a constant and $k$ is small, we have $O(n)$

• Is Radix sort better than Counting sort?

  for example: $k = n^2$ then running time for counting sort = $O(k) = O(n^2)$

  for radix sort, each number can be stored using $2 \log n$ bits (binary numbers upto $n^2$ in computer)

  If we divide a number into $[\log n \text{ bits} | \log n \text{ bits}]$
  // each $\log n$ bits = $n$ bins

  Using 2 pass, each need $O(n)$, total = $2*O(n) = O(n)$

  Suggestion: use Radix sort when $n \geq 2000$
11. STL Sorting algorithms

- #include <algorithm>

- // may define “<” operation using extra last parameter in sorting algorithms

- Sorting

  // This is not a stable algorithm
  void sort(RandomAccessIterator first, RandomAccessIterator last);

  Note: RandomAccessIterator is defined for array, vector and deque.

- Mergesort

  // two mergesort functions
  OutputIterator merge( InputIterator1 first1, InputIterator1 last1,
                        InputIterator2 first2, InputIterator2 last2,
                        OutputIterator result);

  inline void inplace_merge( BidirectionalIterator first,
                            BidirectionalIterator middle,
                            BidirectionalIterator last);

- InputIterator & OutputIterator permit "single pass" algorithms. InputIterator allows only read access, and OutputIterator allows only write access

- BidirectionalIterator permits “multi pass” algorithms. It supports motion in both directions & allows read access and write access

- See http://www.sgi.com/tech/stl/Iterators.html
• Lexicographical comparison

// [1 2 3 4] < [1 2 3 4] → false; [1 2 3 4] < [1 2 4] → true
// [1 2 4] < [1 2 3 4] → false

bool lexicographical_compare(InputIterator1 first1, InputIterator1 last1,
                            InputIterator2 first2, InputIterator2 last2);

• copy()

  • format :

    OutputIterator copy( InputIterator first, InputIterator last,
                       OutputIterator result);

    Copy copies elements from the range [first, last) to the range [result,
    result + (last - first)). That is, it performs the assignments *result = *first,
    *(result + 1) = *(first + 1), and so on. The return value is result + (last -
    first).

• Example :

  vector<int> V;
  V.push_back(5); V.push_back(8); V.push_back(7); V.push_back(9);
  list<int> L(5); // need to make sure L size is enough!
  copy(V.begin(), V.end(), L.begin());

• istream_iterator/ostream_iterator

  • istream_iterator (ostream_iterator) is an inputIterator (outputIterator) that
    performs formatted input (output) of objects from a particular istream
    (ostream)

  • See http://www.sgi.com/tech/stl/ostream_iterator.html

• Example : Copy the elements of a vector to the standard output, one per
  line, i.e. copy(V.begin(), V.end(),ostream_iterator<int>(cout, "\n");
• Other sorting algorithms: nth_element(), min_element(), max_element(), binary_search(), heapsort operations, stable_sort(), heap sort operations and etc.

• Example 1:

```c++
libra% cat -n sort.C
    1 #include <iostream>
    2 #include <vector>
    3 #include <deque>
    4 #include <list>
    5 #include <algorithm>
    6 #include <cassert>
    7 using namespace std;
    8
    9 inline bool ignoreCase(char c1, char c2)
   10 {  
   11     return tolower(c1) < tolower(c2);
   12   }
   13
   14 main()
   15 {
   16     vector<int> v,u;
   17     deque<char> w;
   18     v.push_back(5); v.push_back(8); v.push_back(7); v.push_back(9);
   19     cout << "Use ostream iterator to list elements, before sort() : ";
   20     copy(v.begin(), v.end(), ostream_iterator<int>(cout, "  "));
   21     cout << endl;
   22     sort(v.begin(), v.end());
   23     cout << "After sort() : ";
   24     copy(v.begin(), v.end(), ostream_iterator<int>(cout, "  "));
   25     cout << endl;
   26     cout << "-----------------------------\n";
   27   }
```
w.push_back('G'); w.push_back('K'); w.push_back('V'); w.push_back('a');
cout << "Before sort(): ";
copy(w.begin(), w.end(), ostream_iterator<char>(cout, " "));
cout << endl;

sort(w.begin(), w.end());
cout << "After sort(): ";
copy(w.begin(), w.end(), ostream_iterator<char>(cout, " "));
cout << endl;

copy(w.begin(), w.end(), ignoreCase);
cout << "After sort() using ignoreCase(): ";
copy(w.begin(), w.end(), ostream_iterator<char>(cout, " "));
cout << endl;
cout << "---
"

u.push_back(1); u.push_back(3); u.push_back(10); u.push_back(19);
cout << "Two sorted lists in two vectors, before merge():
		";
copy(v.begin(), v.end(), ostream_iterator<int>(cout, " "));
copy(u.begin(), u.end(), ostream_iterator<int>(cout, " "));
cout << "After merge(): ";
merge(v.begin(), v.end(), u.begin(), u.end(),

        ostream_iterator<int>(cout, " "));
cout << endl;

cout << "---
"
u.push_back(5); u.push_back(11); u.push_back(20); u.push_back(88);
cout << "Two sorted lists in a vector, before inplace_merge():
		";
copy(u.begin(), u.end(), ostream_iterator<int>(cout, " "));
cout << "After inplace_merge(): ";
inplace_merge(u.begin(), u.begin()+4, u.end());
cout << "After inplace_merge(): ";
copy(u.begin(), u.end(), ostream_iterator<int>(cout, " "));
cout << endl;
8. Internal Sorting

cout << "---------------------------------------------------\n";

int A1[] = {3, 1, 4, 1, 5, 9, 3};
int A2[] = {3, 1, 4, 2, 3, 2, 7};

int A3[] = {1, 2, 3, 4, 5};
int A4[] = {1, 2, 3, 4};

const int N1 = sizeof(A1) / sizeof(int);
const int N2 = sizeof(A2) / sizeof(int);
const int N3 = sizeof(A3) / sizeof(int);
const int N4 = sizeof(A4) / sizeof(int);

bool C12 = lexicographical_compare(A1, A1 + N1, A2, A2 + N2);
bool C34 = lexicographical_compare(A3, A3 + N3, A4, A4 + N4);

cout << "A1[] < A2[]: " << (C12 ? "true" : "false") << endl;
cout << "A3[] < A4[]: " << (C34 ? "true" : "false") << endl;

cout << "---------------------------------------------------\n";

list<int> z;
z.push_back(5); z.push_back(11); z.push_back(20); z.push_back(88);
bool t = lexicographical_compare(z.begin(), z.end(), z.begin(), z.end());
cout << "z[] < z[]: " << (t ? "true" : "false") << endl;

cout << "---------------------------------------------------\n";
libra%
libra% g++ sort.C
libra% a.out

Use ostream iterator to list elements, before sort(): 5 8 7 9
After sort(): 5 7 8 9

Before sort(): G K V a
After sort(): G K V a
After sort() using ignoreCase(): a G K V

Two sorted lists in two vectors, before merge():
  5 7 8 9
  1 3 10 19
After merge(): 1 3 5 7 8 9 10 19

Two sorted lists in a vector, before inplace_merge():
  1 3 10 19 5 11 20 88
After inplace_merge(): 1 3 5 10 11 19 20 88

A1[] < A2[]: true
A3[] < A4[]: false

z[] < z[]: false

libra%
• Example 2 :

```cpp
#include <iostream>
#include <vector>
#include <deque>
#include <algorithm>
#include <cctype>
using namespace std;

class comClass {
public:
    bool operator()(const int c1, const int c2) const
    { return c1 > c2; }
};

struct comStruct {
    bool operator()(const int c1, const int c2) const
    { return c1 > c2; }
};

bool compare(int c1, int c2)
{
    return c1 > c2;
}

main()
{
    vector<int> v,u,z;
    comClass xx;
    comStruct yy;

    v.push_back(25); v.push_back(38); v.push_back(70); v.push_back(98);
    u.push_back(18); u.push_back(23); u.push_back(40); u.push_back(55);
    for (int a=1; a<=15; a++) z.push_back(a);

    vector<int>::iterator i,j;

    i = z.begin(); j = z.end();
    i++; i++; j--;
    copy(u.begin(), u.end(), ostream_iterator<int>(cout, " "));
    cout << "---------------------------------------------------"
;```
```cpp
8. Internal Sorting

copy(v.begin(), v.end(), ostream_iterator<int>(cout, " "));
cout << "n---------------------------------------------------
";
cout << "j= " << *j << "|i= " << *i << endl;
copy(z.begin(), z.end(), ostream_iterator<int>(cout, " "));
cout << "n---------------------------------------------------
";

j = merge(v.begin(), v.end(), u.begin(), u.end(), i);
cout << "j= " << *j << "|i= " << *i << endl;
copy(z.begin(), z.end(), ostream_iterator<int>(cout, " "));
cout << "n---------------------------------------------------
";

i = z.begin();

// merge(v.begin(), v.end(), u.begin(), u.end(), i, yy);
merge(v.begin(), v.end(), u.begin(), u.end(), i, xx);
// merge(v.begin(), v.end(), u.begin(), u.end(), i, compare);
copy(z.begin(), z.end(), ostream_iterator<int>(cout, " "));
cout << "n---------------------------------------------------
";

// sort(z.begin(), z.end(), yy);
sort(z.begin(), z.end(), xx);
// sort(z.begin(), z.end(), compare);
copy(z.begin(), z.end(), ostream_iterator<int>(cout, " "));
cout << "n---------------------------------------------------
";
}

libra% g++ sort2.c
libra% a.out
18  23  40  55
---------------------------------------------------
25  38  70  98
---------------------------------------------------
j=15|i=3
1  2  3  4  5  6  7  8  9  10  11  12  13  14  15
---------------------------------------------------

j=11|i=18
1  2  18  23  25  38  40  55  70  98  11  12  13  14  15
---------------------------------------------------
25  38  70  98  18  23  40  55  70  98  11  12  13  14  15
---------------------------------------------------
98  98  70  70  55  40  38  25  23  18  15  14  13  12  11
---------------------------------------------------
```