Sorting Algorithms

A sorting algorithm can be classified as being **internal** if the records that it is sorting are in main memory, or **external** if some of the records that it is sorting are in secondary storage (harddisk, SSD, or tape drive).

In this course, we shall only discuss internal sorting algorithms.
To simplify discussion, we shall consider the sorting of array of integers in most of the examples.

Efficiency of a sorting method is usually measured by the number of comparisons and data movements required.

Bubble sort

- Scan the array from left to right, exchange pairs of elements that are out-of-order.
- Repeat the above process for up to N–1 times where N is the number of records in the array.

Example:

```
pass 1: 25  57  48  37  92  60
        25  57  48  37  92  60  ;swap 57 with 48
        25  48  57  37  92  60  ;swap 57with 37
        25  48  37  57  92  60
        25  48  37  57  92  60  ;swap 92 with 60
        25  48  37  57  60  92  ;92 is excluded in the next pass

pass 2: 25  48  37  57  60  92
        25  48  37  57  60  92  ;swap 48 with 37
        25  37  48  57  60  92
        25  37  48  57  60  92
        25  37  48  57  60  92  ;60 is excluded in the next pass

pass 3: 25  37  48  57  60  92
        25  37  48  57  60  92
        25  37  48  57  60  92  ;no swapping takes place in a pass
                                ;sorting is done
```

```
void bubbleSort(int x[], int N)
{
   int switched = 1;

   for (int pass = 1; pass < N && switched; pass++)
   {
      switched = 0;

      for (int j = 0; j < N-pass; j++)
         if (x[j] > x[j+1]) // assert: j+1 is a valid index
         {
            int temp = x[j];
            x[j] = x[j+1];
            x[j+1] = temp;
            switched = 1;
         }
   }
}
```

Complexity of bubble sort

- If the sorting takes k passes, the total number of comparisons is

  $$\sum_{i=1}^{k}(N-i) = (2kN - k^2 - k)/2$$

- The number of data movement is up to 3 times the number of comparisons

- Time complexity

Best case	$k = 1$	$O(N)$
Average case	$k = N/2$	$O(N^2)$
Worst case	$k = N-1$	$O(N^2)$

- Bubble sort is a simple sorting method, but its <u>efficiency is poor</u> (because of the large number of data movement) when implemented on a conventional sequential machine.

Insertion sort

Successively insert a new element into a sorted sublist.

Example:

input array: [25 57 48 37 92 60]

sorted sublist: [25] 57 48 37 92 60 ; initial condition
sorted sublist: [25 57] 48 37 92 60 ; insert 57
sorted sublist: [25 48 57] 37 92 60 ; insert 48
sorted sublist: [25 37 48 57] 92 60 ; insert 37
sorted sublist: [25 37 48 57 92] 60 ; insert 92
sorted sublist: [25 37 48 57 60 92] ; insert 60

```
void insertionSort(int x[], int N)
{
   for (int i = 1; i < N; i++)
   {
      int t = x[i];
      int j;
      for (j = i-1; j >= 0 && x[j] > t; j--)
         x[j+1] = x[j];

      x[j+1] = t;
   }
}
```

Complexity of insertion sort

- Let c be the number of comparisons required to insert an element into a sorted sublist of size k.

  Best case: $c = 1$
  Average case: $c = k/2$
  Worst case: $c = k$

- The total number of comparisons = $\sum_{k=1}^{N-1} c$
- Number of data movement ≈ number of comparisons
- Time complexity

Best case	$c = 1$	$O(N)$
Average case	$c = k/2$	$O(N^2)$
Worst case	$c = k$	$O(N^2)$

- Because of the simplicity of insertion sort, it is often used to sort arrays with small number of elements, e.g. $N < 20$.

User defined generic insertion sort function

```
template<class Type>
void insertionSort(Type *x, unsigned n,
                   int (*compare)(const Type&, const Type&))
{

// compare(const Type&, const Type&) is a function parameter
// int is the return type of function compare()

// The function parameter is passed by address, i.e.
// pass the address of a function to another function.

   for (int i = 1; i < n; i++)
   {
      Type t = x[i];
      int j;
      for (j = i-1; j >= 0 && compare(x[j], t) > 0; j--)
         x[j+1] = x[j];

      x[j+1] = t;
   }
}
```

```
Return value of compare(a, b)
  - return a positive value if a > b
  - return the value zero   if a == b
  - return a negative value if a < b
```

Example codes

```
struct date
{
   int year, month, day;
};

int compareDate(const date& a, const date& b)
{
   if (a.year != b.year)
      return a.year – b.year;

   if (a.month != b.month)
      return a.month – b.month;

   return a.day – b.day;  // can use subtraction operation
                          // to compare int values
}

int main()
{
   int n = 100;
   date *a = new date[n];

   // codes to generate values in a[]


   // sort the array a[] in ascending order
   insertionSort(a, n, compareDate);

}
```

Example compare functions for the user defined generic `insertionSort` function.

Function to compare floating point numbers

```
int compareDouble(const double& a, const double& b)
{
   return a – b;  // Incorrect comparison, why ?
}
```

A correct implementation to compare floating point numbers

```
int compareDouble(const double& a, const double& b)
{
   if (a < b)
      return -1;
   else if (a > b)
      return 1;
   else
      return 0;
}
```

Function to compare C++ string objects

```
int compareString(const string& a, const string& b)
{
   return a.compare(b); // alphabetical order
}
```

Function to compare cstring

```
typedef char* charptr;

int compare_cstring(const charptr& a, const charptr& b)
{
   return strcmp(a, b);  // strcmp is a C library function
                         // alphabetical order
}
```

Quicksort

Consider an array segment $x[s..e]$

1. Choose a pivot element a from a specific position, say $a = x[s]$
2. Partition $x[s..e]$ using a, i.e. rearrange the elements in $x[s..e]$ and a is placed into position j (*i.e.* $x[j] = a$) such that $x[i] \le a$ for $i = s, s+1, \ldots, (j-1)$, and $x[k] > a$ for $k = j+1, \ldots, e$
3. The two subarrays $x[s..(j-1)]$ and $x[(j+1)..e]$ are sorted recursively using the same method.

Partitioning operation:

Step 1: scan the array from left to right to look for an element x[i] > x[s]

Step 2: scan the array from right to left to look for an element x[j] ≤ x[s]

Step 3: if ($i < j$) swap x[i] with x[j] and go to step 1;
otherwise swap x[s] with x[j] (partitioning finished).

Example: Partitioning process

```
     |→                      ←|
50  33  64  48  37  92  25  57       // i < j, swap x[i] with x[j]
        ↑i                  ↑j

50  33  25  48  37  92  64  57       // i > j, swap x[s] with x[j]
                ↑j  ↑i

37  33  25  48  50  92  64  57       // partitioning finished
[←   ≤ 50   →]      [← > 50 →]       // sort the left and right sublists recursively
  left sublist      right sublist
```

Recursive partitioning of the left sublist

```
25  33  37  48  50  92  64  57
[←  →]
```

Recursive partition of the right sublist

```
25  33  37  48  50  57  64  92
                    [←    →]
```

```
void swap(int x[], int i, int j)
{
   int t = x[i];
   x[i] = x[j];
   x[j] = t;
}


int partition(int x[], int s, int e)
{
   // precondition: s < e
   int i = s + 1;
   int j = e;
   bool done = false;

   while (!done)
   {
      while (i < j && x[i] <= x[s])
         i++;

      while (x[j] > x[s])  // j will NOT go out of bound
         j--;

      if (i < j)
         swap(x, i, j);
      else
         done = true;
   }
   swap(x, s, j); // swap x[s] and x[j]

   retrun j;
}


void simpleQuicksort(int x[], int start, int end)
{
   if (start < end)
   {
      int j = partition(x, start, end);
      simpleQuicksort(x, start, j-1);
      simpleQuicksort(x, j+1, end);
   }
}
```

Complexity of quicksort

Let $T(N)$ be the time to sort an array of size N using quicksort, and $T(1) = b$.
The time to partition the array = cN.
b and c are some constants.

In the best case, each time the array segment is partitioned into 2 subarrays of roughly equal size.

$$T(N) = 2T(N/2) + cN$$

We can expand the recurrent equation:

$$\begin{aligned}
T(N) &= 2(2T(N/4) + cN/2) + cN \\
&= 4T(N/4) + cN + cN \\
&= \ldots \\
&= NT(1) + cN + \ldots + cN \\
&= bN + (\log_2 N) \times cN \\
&= O(N \log_2 N)
\end{aligned}$$

In the worst case, each time the array segment is partitioned into an empty subarray and a subarray of size $N–1$.

$$\begin{aligned}
T(N) &= T(N-1) + cN \\
&= T(N-2) + c(N-1) + cN \\
&= \ldots \\
&= T(1) + 2c + \ldots + cN \\
&= b + c \sum_{i=2}^{N} i \\
&= O(N^2)
\end{aligned}$$

It can also be shown that the average case complexity of quicksort is approximately equal to $1.38\, N \log_2 N$.

Remark: If the size of the array is large, quicksort is one of the most efficient sorting methods known today.

Improvements to the basic quicksort algorithm

1. Choose the median of the first, last, and middle elements as the pivot in the partitioning process.
2. If the length of the sublist is less than some threshold, e.g. 10, use insertion sort to sort the sublist instead.

```
#define Threshold 10

void quicksort(int x[], int start, int end)
{
   if (end - start > Threshold)
   {
      int mid = (start + end) / 2;

      if (x[start] >= x[mid])
      {
         if (x[mid] >= x[end])
            swap(x, start, mid);  // order: s >= m >= e
         else if (x[start] >= x[end])
            swap(x, start, end);  // order: s >= e >= m
         // else no swapping required, order: e >= s >= m
      }
      else
      {
         if (x[end] >= x[mid])
            swap(x, start, mid);  // order: e >= m >= s
         else if (x[end] >= x[start])
            swap(x, start, end);  // order: m >= e >= s
          // else no swapping required, order: m >= s >= e
      }

      //assert: x[start] is the median of x[start], x[mid],
      //        and x[end]

      int j = partition(x, start, end);
      quicksort(x, start, j-1);
      quicksort(x, j+1, end);
   }
   else
      insertionSort(x, start, end); //slight modification
                                    //required
}
```

The `qsort()` and `bsearch()` functions in the C library **<cstdlib>**

Template function is NOT supported in C.

```
//function interface

void qsort(void *base, unsigned num, unsigned objSize,
           int (*compare)(const void *, const void *));

// base = starting address of the array
// num = number of elements in the array
// objSize = size of an element (no. of bytes)
// compare is a function parameter
// void * is a pure address, with no data type information
// associated to it.



void* bsearch(const void *key, void *base,
              unsigned num, unsigned objSize,
              int (*compare)(const void *, const void *));

// key = address of the key record
// Return value:
//   - A pointer to an element in the array that matches the
//     search key.
//   - If there are more than 1 matching elements, this may
//     point to any of them.
//   - If key is not found, a null pointer is returned.
```

```
//Example: use qsort()to sort an array of date

struct date
{
   int year, month, day;
};

int compareDate(const void *a, const void *b)
{
   date *d1 = (date *)a;  // typecast the pointer
   date *d2 = (date *)b;  // before using it to reference
                          // the date object
   if (d1->year != d2->year)
      return d1->year – d2->year;

   if (d1->month != d2->month)
      return d1->month - d2->month;

   return d1->day - d2->day;
}

int main()
{
   int len = 100;
   date *list = new date[len];
   // codes to assign values to list[]

   qsort(list, len, sizeof(date), compareDate);

   // sizeof(dataType) is a compiler directive

   // list is a pointer (4 bytes),
   // hence sizeof(list) is equal to 4.
   // Prefer to use sizeof(date *) rather than list.

   // sizeof(list) is NOT equal to the length of list.
}
```

Use the `qsort` function to sort an array of **cstring**, i.e. `char[]`.

```
#include <cstring>  // C library

int compare_cstring(const void *a, const void *b)
{
   char **c1 = (char **)a;
   char **c2 = (char **)b;

   // cstring is a char[], i.e. char *
   // b is a pointer to a cstring
   // hence, data type of b is char **

   return strcmp(*c1, *c2);  //compare cstring
}

int main()
{
   char *months[] = {“January”, “February”, “March”,
                     “April”, “May”, “June”, “July”,
                     “August”, “September”, “October”,
                     “November”, “December”};
   int n = 12;

   qsort(months, n, sizeof(char *), compare_cstring);

   // remark: sizeof(char *) = 4

}
```

Use the qsort function to sort an array of **string**, i.e. C++ string.

```
#include <string>  // C++ string class

int compareString(const void *a, const void *b)
{
   string *s1 = (string *)a;
   string *s2 = (string *)b;

   // use member function compare() in string class to
   // compare string objects

   return s1->compare(*s2);  // or (*s1).compare(*s2)
}

int main()
{
   string months[] = {“January”, “February”, “March”,
                      “April”, “May”, “June”, “July”,
                      “August”, “September”, “October”,
                      “November”, “December”};
   int n = 12;

   qsort(months, n, sizeof(string), compareString);

   // remark: sizeof(string) = 28 (in Visual Studio)

   // Example on the uses of bsearch

   string key = “June”;

   string *p = (string *)
               bsearch(&key, months, n, sizeof(string),
                       compareString);

   if (p != nullptr)
   {
      int loc = p – months;  // convert to index
      cout << key << “ is found at index ” << loc << endl;
   }
   else
      cout << key << “ is not found” << endl;
}
```

Use insertion sort to illustrate the internal details of the sort function in the C library `<cstdlib>`

Function `memcpy` (memory copy) in `<cstring>`

```
void* memcpy(void *destination, const void *source, unsigned n);
// copy n bytes from source to destination
// return the destination address


void insertionSort(void *base, unsigned n, unsigned objSize,
                   int (*compare)(const void *, const void *))
{
   char *t = new char[objSize];  // t[] to store 1 array element

   char *b = (char *)base;
   // byte address, sizeof(char) = 1

   for (int i = 0; i < n; i++)
   {
      memcpy(t, b+i*objSize, objSize); // t = base[i]
      // b+i*objSize is the address of base[i]

      int j;
      for (j = i-1; j >= 0 && compare(b+j*objSize, t) > 0; j--)
         memcpy(b+(j+1)*objSize, b+j*objSize, objSize);
         // base[j+1] = base[j]

      memcpy(b+(j+1)*objSize, t, objSize);  // base[j+1] = t
   }

   delete[] t; // Return memory resource to system.
}
```

Merge sort

- Initially the input file is divided into N subfiles of size 1.
- Adjacent pairs of files are merged to form larger subfiles.
- The merging process is repeated until there is only one file remaining.

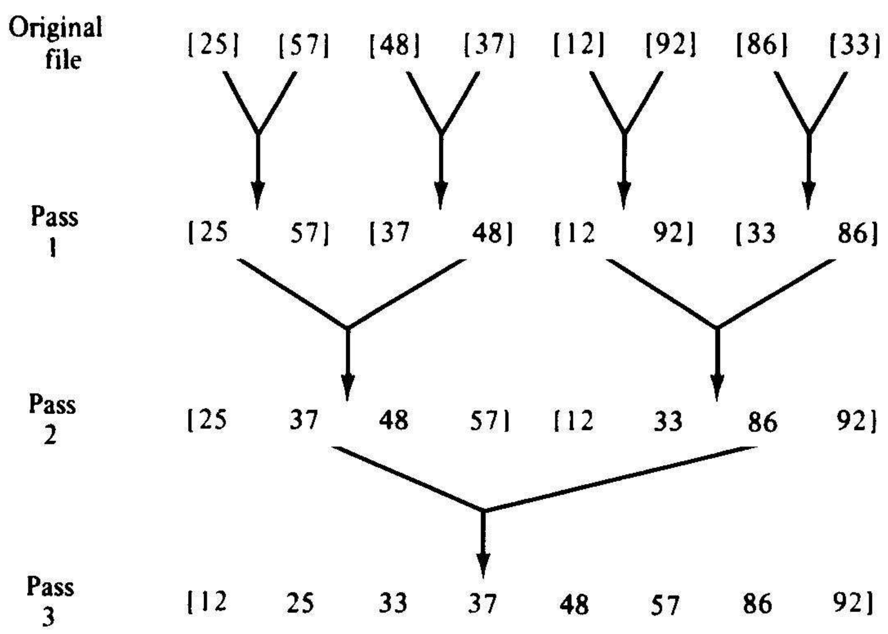

Figure 6.5.1 Successive passes of the merge sort.

pseudo code of the mergesort algorithm

```
//x[] is the input array, aux[] is the temporary storage

size = 1;  //initial size of subfiles
while (size < n) //still have 2 or more subfiles
{
   for (i = 0; i < no. of subfiles – 1; i += 2)
      merge subfiles i and i+1 from x[] to aux[];

   copy any remaining single subfile from x[] to aux[];

   size *= 2;

   copy aux[] to x[] for preparation of the next merge-pass;

}
```

```
void mergesort(int x[], int n)
{
   int *aux, i, j, k, L1, L2, u1, u2, size;

   aux = new int[n];

   size = 1; //size of subfiles
   while (size < n)
   {
      L1 = 0; k = 0;
      while (L1 + size < n) // 2 or more files to merge
      {
         L2 = L1 + size;
         u1 = L2 – 1;
         u2 = (L2+size-1 < n) ? L2+size-1 : n-1;

         // merge subfiles x[L1..u1] and x[L2..u2]
         for (i = L1, j = L2; i <= u1 && j <= u2; k++)
            if (x[i] <= x[j])
               aux[k] = x[i++];
            else
               aux[k] = x[j++];

         while (i <= u1)
            aux[k++] = x[i++];
         while (j <= u2)
            aux[k++] = x[j++];

         // advance L1 to the start of the next pair of
         // files
         L1 = u2+1;
      }

      // copy any remaining single file
      for (i = L1; i < n; i++)
        aux[k++] = x[i];

      // copy aux[] back to x[] and adjust size
      for (i = 0; i < n; i++)
         x[i] = aux[i];

      size *= 2;
   }
   delete[] aux;
}
```

Improvement to the mergesort algorithm:

- Instead of merging each set of files from `x[]` to `aux[]` and then copy `aux[]` back to `x[]`, alternate merge passes can be performed from `x[]` to `aux[]` and from `aux[]` to `x[]`.

Complexity of mergesort

- mergesort requires $O(N)$ additional space for the auxiliary array
- The time to do one merge pass is $O(N)$.
- In one merge pass, the size of the sorted subfiles is doubled. Hence, $\log_2 N$ merge passes are required.
- The overall time complexity is $O(N \log_2 N)$.

Stable property of sorting algorithms

- Let *x* and *y* be 2 records with equal key value, and *x* appears before *y* in the input list.
- A sorting method is said to be **stable** if the relative order of *x* and *y* is preserved in the output, i.e. *x* appears before *y* in the sorted list.
- Bubble sort, insertion sort, and mergesort are stable.
- quicksort is not stable.

Example

- Suppose we have an array of student records ordered by student name.
- We want to sort the array by program code (e.g. BECE, BSCS, BBA, etc.) and students in the same program are ordered by student name.
- If you use qsort to sort the array by program code (i.e. only compare the program code), students in the same program may not be ordered by student name.
- If you use insertion sort to sort the array by program code, then students in the same program are ordered by student name.