Sorting Algorithms

- Internal sorting: the whole dataset is stored in main memory.
- External sorting: the dataset is too large and cannot be stored in main memory, i.e. some of the data records 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.

In this section, we shall discuss 4 sorting algorithms

- Bubble sort
- Insertion sort
- Quicksort
- Merge sort

Heapsort and Tree sort will be discussed after we have introduced binary tree.

Built-in sort functions in C/C++ (also for Java and Python) are based on variants of quicksort and merge sort algorithms.

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: $\underline{25\ \ 57}\ \ 48\ \ 37\ \ 92\ \ 60$
$25\ \ \underline{57\ \ 48}\ \ 37\ \ 92\ \ 60$;swap 57 with 48
$25\ \ 48\ \ \underline{57\ \ 37}\ \ 92\ \ 60$;swap 57with 37
$25\ \ 48\ \ 37\ \ \underline{57\ \ 92}\ \ 60$
$25\ \ 48\ \ 37\ \ 57\ \ \underline{92\ \ 60}$;swap 92 with 60
25 48 37 57 60 **92** ;92 is excluded in the next pass

pass 2: $\underline{25\ \ 48}\ \ 37\ \ 57\ \ 60\ \ 92$
$25\ \ \underline{48\ \ 37}\ \ 57\ \ 60\ \ 92$;swap 48 with 37
$25\ \ 37\ \ \underline{48\ \ 57}\ \ 60\ \ 92$
$25\ \ 37\ \ 48\ \ \underline{57\ \ 60}\ \ 92$
25 37 48 57 **60** 92 ;60 is excluded in the next pass

pass 3: $\underline{25\ \ 37}\ \ 48\ \ 57\ \ 60\ \ 92$
$25\ \ \underline{37\ \ 48}\ \ 57\ \ 60\ \ 92$
$25\ \ 37\ \ \underline{48\ \ 57}\ \ 60\ \ 92$;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 < N, with pass > 0
         {
            int temp = x[j];
            x[j] = x[j+1];
            x[j+1] = temp;
            switched = 1;
         }
   }
}
```

Complexity of bubble sort

- Number of comparisons required in pass i is $N - i$.
- 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 efficiency is poor (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       ; 57 inserted
sorted sublist:  [25  48  57] 37  92  60       ; 48 inserted
sorted sublist:  [25  37  48  57] 92  60       ; 37 inserted
sorted sublist:  [25  37  48  57  92] 60       ; 92 inserted
sorted sublist:  [25  37  48  57  60  92]      ; 60 inserted
```

```
void insertionSort(int x[], int N)
{
   // initially x[0..0] is already sorted

   for (int i = 1; i < N; i++)   // insert x[i] to x[0..i-1]
   {
      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. $N < 20$.

Possible improvement to insertion sort:
- User binary search to determine the insertion point, i.e. reduce the number of key comparison.
  * This is helpful if key comparison is expansive, i.e. comparison is not a constant time operation, e.g. string comparison.
- Data movement still requires linear time.
- Overall time complexity is still $O(N^2)$

User defined generic insertion sort function

- Sort function that can be used to sort an array of any data type.

```
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 the called function.

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

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

Convention of the compare function (full ordering):

```
Return value of compare(a, b)
  - return a positive int   if a > b
  - return the value zero   if a == b
  - return a negative int   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 up to 32-bit 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);

}
```

Compare functions for other data types.

Function to compare floating point numbers

```
int compareDouble(const double& a, const double& b)
{
   return a - b;  // Incorrect comparison result, 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 comp_string(const string& a, const string& b)
{
   return a.compare(b); //alphabetical order, case sensitive
}
```

Function to compare cstring

```
typedef char* charptr;

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

Function to compare cstring ignore cases (case-insensitive).
C/C++ does not have built-in case-insensitive string comparison function.
If you want to do case-insensitive comparison, you need to write your own compare function.

```
int comp_cstring_NoCase(const charptr& a, const charptr& b)
{
   int diff = 0;
   for (int i = 0; diff == 0 &&
                  (a[i] != '\0' || b[i] != '\0'); i++)
      diff = tolower(a[i]) - tolower(b[i]);

   return diff;
}
```

Remarks about the Visual Studio compiler

```
int comp_cstring(const char*& a, const char*& b)
{
   return strcmp(a, b);
}

void otherFn()
{
   char u[] = “abcd”;
   char v[] = “cdef”;

   int r = comp_cstring(u, v);  // compiler signals an error

// a reference of type “const char*&” (not const-qualified)
// cannot be initialized with a value of type “char[]”

   char* a = u;
   char* b = v;
   int k = comp_cstring(a, b); // compiler signals an error
}
```

```
// without the const-qualifier
int comp_cstring(char*& a, char*& b)
{
   return strcmp(a, b);
}

void otherFn()
{
   char u[] = “abcd”;
   char v[] = “cdef”;

   char* a = u;
   char* b = v;
   int k = comp_cstring(a, b); // accepted by compiler

   int r = comp_cstring(u, v); // compiler signals an error!

}
```

Quicksort

To sort 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 $x[j] \le x[s]$

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)
   {
      // require index bound check
      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 compare-exchange based 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.

```
void qsort(void* base, size_t num, size_t size,
           int (*compar)(const void*, const void*));
// void* is a pure address, with no data type information
// associated to it.
// size_t is equivalent to unsigned int

// base = starting address of the array
// num = number of elements in the array
// size = size of an element (no. of bytes)
// compar is a function parameter


void* bsearch(const void* key, const void* base,
              size_t num, size_t size,
              int (*compar)(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.

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

Remarks:

1. **In general, the same compare function is used in sorting and searching. (But there may be exceptions based on the application requirements).**
2. **The actual data objects at addresses *base and *key should be of the same data type. (User does not know the exact details of the internal implementation of the lib functions. If data types of objects at *base and *key are not the same, it may lead to errors not easy to diagnose.)**

```
//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), sizeof(list) = 4.

   // More explicit to use sizeof(date *) rather than
   // sizeof(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 comp_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
}
```

Alternative way to define the compare function

```
typedef char* charptr;

int comp_cstring(const void* a, const void* b)
{
   charptr *c1 = (charptr *)a;
   charptr *c2 = (charptr *)b;
   return strcmp(*c1, *c2);  //compare cstring
}
```

```
int main()
{
   char *month[13];  // charptr month[13];
   month[0] = "Illegal month";
   month[1] = "January";
   ...
   month[12] = "December";

   int n = 13;

   qsort(months, n, sizeof(char *), comp_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 comp_string(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[] = {"Illegal month", "January",
                      "February", "March", "April",
                      "May", "June", "July", "August",
                      "September", "October",
                      "November", "December"};
   int n = 13;

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

   // remark: sizeof(string) = 28

   // 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;
}
```

A possible design of `bsearch`

```
void* bsearch(const void* key, const void* base,
              size_t num, size_t size,
              int (*compar)(const void*, const void*))
{
   char *b = (char*)base;  // sizeof(char) = 1
   int low = 0;
   int high = num - 1;
   while (low <= high)
   {
      int mid = (low + high) / 2;
      int r = compar(key, b + mid * size);
      if (r == 0)
         return b + mid * size;
      if (r < 0)
         high = mid - 1;
      else
         low = mid + 1;
   }
   return nullptr;
}
```

Remark:
It is also possible to perform the key comparison by **`compar(b + mid * size, key)`**, and adjust the if-else statements accordingly.

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

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

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

```
void insertionSort(void* base, size_t n, size_t size,
                   int (*compar)(const void*, const void*))
{
   char* b = (char *)base; // sizeof(char) = 1

   char* t = new char[size];  // t[] to store an array element

   for (int i = 1; i < n; i++)
   {
      memcpy(t, b+i*size, size); // t = base[i]
      // b+i*size is the address of base[i]

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

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

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

Comparison of bubble sort, insertion sort and qsort

	Bubble sort	Insertion sort	qsort (quicksort)
Computation complexity	$O(n^2)$	$O(n^2)$	$O(n \log n)$

Execution time to sort an array of double (i7-3770 CPU):

n	Bubble sort	Insertion sort	qsort
10,000	0.297 second	0.062 second	< 0.001 second
100,000	26.832 seconds	5.475 seconds	0.062 second
1,000,000	44.7 minutes (estimated)	9.1 minutes (estimated)	0.697 second

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.

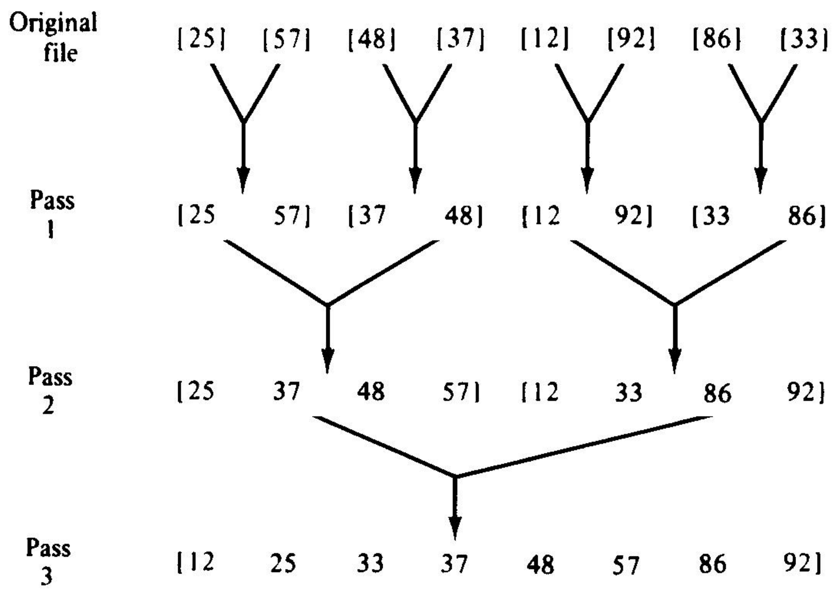

Figure 6.5.1 Successive passes of the merge sort.

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)
{
   while (there are 2 or more subfiles not yet merged)
      left subfile is from L1 to u1
      right subfile is from L2 to u2
      merge left and right subfiles 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;
}
```

Implementing merge sort using recursion

```
void msort(int x[], int s, int e)
{
   // Recursive merge sort. Sort x[s..e] inclusive.
   if (s >= e)
      return;

   int mid = (s + e) / 2;
   msort(x, s, mid);    // sort left sublist x[s..mid]
   msort(x, mid+1, e); // sort right sublist x[mid+1..e]

   // Merge x[s..mid] with x[mid+1..e]
   int* aux = new int[e-s+1];
   int i = s;
   int j = mid + 1;
   int k = 0;
   while (i <= mid && j <= e)
   {
      if (x[i] <= x[j])
         aux[k++] = a[i++];
      else
         aux[k++] = a[j++];
   }
   while (i <= mid)
      aux[k++] = x[i++];
   while (j <= e)
      aux[k++] = x[j++];

   for (i = 0; i < k; i++) // copy sorted list back to x[]
      x[s+i] = aux[i];

   delete[] aux;  // return memory space to OS
}


void mergeSort(int x[], int n)  // driver function
{
   msort(x, 0, n-1);
}
```

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.