Chapter 7
Sorting

Terminology

- **Internal** – sort done totally in main memory.
- **External** – uses auxiliary storage (disk).
- **Stable** – retains original order if keys are the same. In asking if a sort is stable, we are asking can it be reasonably coded to be stable.
- **Adaptive (Non-Oblivious)** – takes advantage of existing order to do less work.
- **Sort by address** – uses indirect addressing so the record (structure) doesn’t have to be moved.
- **Inversion** – a pair of elements that is out of order. Important in determining a lower bound. The number of pairs of elements is \( \frac{n(n-1)}{2} \) (\( n \) ways to pick first, \( n-1 \) ways to pick the second, divide by two as order isn’t important). On average, only half of these pairs are out of order, so number of inversions is \( \frac{n(n-1)}{4} \). \( n^2 \) swaps of adjacent elements are required.

### Selection Sort

- Select elements one at a time and place in proper final position.
- Repeatedly find the largest (or smallest).

```cpp
template<class T>
void SelectionSort(T a[], int n)
{
    // Sort the n elements a[0:n-1].
    for (int size = n; size > 1; size--)
    {
        int j = Max(a, size); // loop
        Swap(a[j], a[size - 1]); // loop
    } // for
} // SelectionSort
```

**Figure 1 Selection Sort**

- Analysis: \( n + n - 1 + n - 2 + \ldots + 1 = n(n - 1)/2 \)
- Only \( n \) moves – use when records (structure) are long.
- Requires \( n^2 / 2 \) compares even when almost already sorted – non-adaptive (Oblivious).
- **Unstable** – because it may swap the element in the last location past other instances of itself.

In this static diagram to show sort running: The y axis is location in array. The magnitude of a number is indicated by shading - higher numbers are darker, and lower numbers are lighter. We begin on the left hand side with the numbers in a random order, and the sorting progression plays out until we reach the right hand side with a sorted sequence. Time, in this particular case, is measured by the number of "swaps" performed.
Bubble Sort

- Compares adjacent elements – exchanges if out of order.
- Lists get smaller each time – at least one is placed in final order.
- Place of last swap is as much as you have to look at. Quit if you do no swaps.
- Code is shown in Figure 2.
- Stable
- adaptive – if the hasSwapped flag is included
- Analysis: \[ n + n - 1 + n - 2 + ... + 1 = n(n - 1)/2 \]

```cpp
void bubble(Data[] a, int size)
{
    bool hasSwapped = false;
    for (int i = 0; i < size - 1; i++)
    {
        for (int j = 0; j < size - 1 - i; j++)
            if (a[j].value > a[j+1].value)
            {
                hasSwapped = true;
                swap(a[j], a[j+1]);
            }
    }
    if (!hasSwapped) return;
}
```

Figure 2 Bubble Sort
**Insertion Sort**

- Sorts by inserting records into an already sorted portion of the array.
- Two groups of keys - sorted and unsorted.
- Code is shown in Figure 3.
- Insert \( n \) times - each time have to move about \( \frac{1}{2} \) elements to insert.
- The number of elements in the list changes so \( \frac{1}{2}(1 + 2 + 3 + \ldots + n) = \frac{1}{2} \times \frac{1}{2} n(n + 1) = \frac{1}{4} n^2 \)
- Performs better when the degree of unsortedness is low – recommended for data that is nearly sorted. Adaptive (Non-Oblivious) as if new element to be added is greater than last element of sorted part, we do not have to look at anything else.
- **Stable** as never swap with other than an adjacent element
- Improvements
  1. Use binary search \( O(n \log n) \) compares, but number of moves doesn’t change so no real gain.
  2. Linked list storage – can’t use binary search – still \( O(n^2) \).
- Could use sentinel containing the key search until \( p->\text{info.key} \geq q->\text{info.key} \)
- **Sentinel** is extra item added to one end of the list so ensure that the loop terminates without having to include a separate check.

```cpp
template<class T>
void InsertionSort(T a[], int n)
{ // Sort a[0:n-1].
  for (int i = 1; i < n; i++)
  {   T t = a[i];
      for (int j = i; j > 0 && t < a[j-1]; j--)
        a[j] = a[j - 1];
      a[j] = t;
  } // for
} // InsertionSort
```

**Figure 3**  Insertion Sort
Shell Sort  (devised by Donald Shell in 1959)

- A subquadratic algorithm whose code is only slightly longer than insertion sort, making it the simplest of the faster algorithms.
- Avoid large amounts of data movement by:
  - Comparing elements that are far apart.
  - Comparing elements that are less far apart.
  - Gradually shrinks toward insertion sort.
- Consider Figure 4, which shows an example of shell sort with the increment sequence of \{5, 3, 1\}.

<table>
<thead>
<tr>
<th>Original</th>
<th>81</th>
<th>94</th>
<th>11</th>
<th>96</th>
<th>12</th>
<th>35</th>
<th>17</th>
<th>95</th>
<th>28</th>
<th>58</th>
<th>41</th>
<th>75</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>After 5-sort</td>
<td>35</td>
<td>17</td>
<td>11</td>
<td>28</td>
<td>12</td>
<td>41</td>
<td>75</td>
<td>15</td>
<td>96</td>
<td>58</td>
<td>81</td>
<td>94</td>
<td>95</td>
</tr>
<tr>
<td>After 3-sort</td>
<td>28</td>
<td>12</td>
<td>11</td>
<td>35</td>
<td>15</td>
<td>41</td>
<td>58</td>
<td>17</td>
<td>94</td>
<td>75</td>
<td>81</td>
<td>96</td>
<td>95</td>
</tr>
<tr>
<td>After 1-sort</td>
<td>11</td>
<td>12</td>
<td>15</td>
<td>17</td>
<td>28</td>
<td>35</td>
<td>41</td>
<td>58</td>
<td>75</td>
<td>81</td>
<td>94</td>
<td>95</td>
<td>96</td>
</tr>
</tbody>
</table>

- Shell sort is known as a **diminishing gap sort**.
- Any sequence of increments will, do as long as it ends in 1 – but some are better than others. In the figure below, the increment sequence is a **geometric sequence** in which every term is roughly 2.2 times smaller than the previous one:

- The code for Shell sort is shown in Figure 5.
- Worst case running time is \( O(n^2) \).
- When the gap goes to 1, the sort is essentially an insertion sort.
- Can get bad running times if all big numbers are at even positions and all small numbers are at odd positions AND the gaps don’t rearrange them as they are always even.
- The average running time appears to be between \( O(n^{5/4}) \) – \( O(n^{7/6}) \) if we use a gap of 2.2.
- Reasonable sort for large inputs. Complicated analysis.
- **Unstable** – as when your gap is five, you swap elements five apart – ignoring duplicates that lie between.
- **Adaptive** – as builds off insertion sort, which is adaptive.

```cpp
template<class T>
void ShellSort (T a[], int length)
{
    int j;
    for (int gap = length / 2; gap > 0;
         gap = gap == 2 ? 1 : static_cast<int>(gap / 2.2))
        for (int i = gap; i < length; i++)
            {
            T tmp = a[i];
            for (j = i; j >= gap && tmp < a[j - gap]; j -= gap)
                a[j] = a[j - gap];
            a[j] = tmp;
        } // for
} // ShellSort
```

Figure 4 Shell Sort

Figure 5 Shell Sort
Heap Sort

- The process goes as follows:
  - Build a max heap (using your almost complete binary tree)
  - While the heap isn’t empty, deleteMax.
- Complexity – $O(n \log n)$
- **Unstable**
Divide and Conquer

- Divide and Conquer is a method of algorithm design that has created such efficient algorithms as Merge Sort.
- In terms of algorithms, this method has three distinct steps:
  - **Divide**: If the input size is too large to deal with in a straightforward manner, divide the data into two or more disjoint parts.
  - **Recur**: Use divide and conquer to solve the sub-problems associated with the data parts.
  - **Conquer**: Take the solutions to the sub-problems and “combine” these solutions into a solution for the original problem.
- Note, the work in recursive solutions happens either in dividing the problem in two or in combining the results. The following algorithms are different in what causes the work.
- The next two sorts are divide and conquer algorithms: MergeSort and QuickSort.

Merge Sort

- Chop the lists into two sublists. Sort the two pieces. Combine by merging. (Some techniques just get sublists of larger and larger powers of two.)
- The pieces may also be sorted via MergeSort, so it is recursive.
- What would the code look like for MergeSort?
- The code for Merge of two sorted segments (NOT the complete mergesort) is shown in Figure 6.
- In merging sublists of length \(n\), clearly no more than \(2n\) compares are required. Actually it is less than this, but it is easy to count this way.
- Each level takes exactly \(n\) compares and there are \(\log n\) levels, the complexity is \(O(n \log n)\).
- A closer count is \(n \log n - 1.25n + 1\) (by running test cases)
- If merging using files (reading and writing sorted pieces to a file), no problem with

```c
void m_sort(int numbers[], int temp[], int left, int right)
{
    if (right <= left) return;
    if (right-left < 64)
    {
        insertionSort(numbers,left,right);
        return;
    }
    int mid = (right + left) / 2;
    m_sort(numbers, temp, left, mid);
    m_sort(numbers, temp, (mid+1), right);
    merge(numbers, temp, left, (mid+1), right);
}

void Merge(int c[], int d[] int l, int m, int r)
{ // Merge c[l:m] and c[m:r] to d[l:r].
    int i = l,    // cursor for first segment
        j = m+1,  // cursor for second
        k = 0;    // cursor for result
    while ((i <= m) && (j <= r))
    if (c[i] <= c[j]) d[k++] = c[i++];
    else d[k++] = c[j++];
    while (j<=r)
    d[k++] = c[j++];
    while (i<=m)
    d[k++] = c[i++];
    memcpy (&c[l], &d[0], (r-l+1)*intSize);
}
```

Figure 6 Merge
storage space. If we have an array of items to be stored, the MergeSort requires an auxiliary storage.

- Works well for external sorts (not all data is in memory at the same time) as doesn’t need to see entire data (like a quicksort does).
- **Stable**, as control order when merging
- **Non-adaptive (Oblivious)** as it does the same amount of work regardless of initial order.

Suppose we want to use less space. Assume the left section is smaller
template<class T>
void Merge_low(T c[],int l, int m, int r)
{ assert(m-l <=r-m);
  int i = 0, // cursor for first segment
      j = m+1, // cursor for second
      k =l; // cursor for result
  //copy c[l..m] to d[] using memcpy for speed
  memcpy (&d[0], &c[l], (m-l+1)*intSize);

  while ((i <= m-l) && (j <= r))
    if (d[i] <= c[j]) c[k++] = d[i++];
    else c[k++] = c[j++];

  while ( i <= m-l)
    c[k++] = d[i++];
}

If the right section is smaller, copy it to temporary memory and start merging from the right ends of both segments.

**Quick Sort**

- In practice, considered to be the fastest sorting algorithm known. timsort is stable and faster on data with order. quicksort is unstable and faster on random data.
- Better than merge sort as partitioning is faster than merging.
- Partition the set into two sets: those elements less than a value in the array, pivot, and those elements greater than pivot.
- Often done as: Use two pointers – top pointer looks for a value smaller than $j$, bottom pointer looks for a value larger than $j$. Then interchange. This does only a third as many swaps.
• Apply recursively.
• The code for QuickSort is shown in Figure 7.
• Analysis: At each level, all elements of array are examined. The number of levels depends on how equally the pieces are divided. Best case: \( \log n \) levels yielding \( O(n \log n) \).
• Worst case: Divide unequally. let \( n \) be the size of the array to be sorted:
  \[ C(n) = n - 1 + C(n-1) = n(n-1)/2 \]
• Space requirement: depends on recursive stacking.
• **Unstable** – as you swap elements far apart during partition (without considering equal element between)
• Not only can quicksort NOT take advantage of partial ordering, it actually gets worse when the file is in order. This is because it doesn’t divide the array in half, so the depth of recursion is \( n \) instead of \( \log n \). A bad thing for complexity, right? This is worse than being oblivious. We could call it **dangerously-oblivious** – it not only fails to take advantage of existing order, it gets worse!!!
• Improvements
  1. Use median of three (or random pivot) so don’t get a bad pivot. (if sort in place, get sentinels)
  2. Use sentinel at each end so don’t have to check (avoid left and right crossing)
  3. Switch to insertion sort when size gets small - can do one big insertion sort on all, but it is recommended to sort the pieces separately because of caching.
  4. Swap elements equal to pivot as partitioning tends to divide chunks of equal nodes in half.

```cpp
template<class T>
void QuickSort( T a[], int l, int r )
{  if ( l >= r ) return;
   int i = l,      // left-to-right cursor
       j = r + 1;  // right-to-left cursor
   T pivot = a[l];
   while ( true ) {
      do { // find >= element on left side
        i = i + 1;
      } while (a[i] < pivot);
      do { // find <= element on right side
        j = j - 1;
      } while ( a[j] > pivot );
      if ( i >= j ) break;  // swap pair not found
      Swap( a[i], a[j] );
   } // while
   a[l] = a[j];
   a[j] = pivot;
   QuickSort( a, l, j-1 ); // sort left segment
   QuickSort( a, j+1, r ); // sort right segment
 } // QuickSort
```

Figure 7  Quick Sort
Bentley-McIlroy 3-way partitioning

Partitioning invariant

<table>
<thead>
<tr>
<th>equal</th>
<th>less</th>
<th>greater</th>
<th>equal</th>
</tr>
</thead>
</table>

- move from left to find an element that is not less
- move from right to find an element that is not greater
- stop if pointers have crossed
- exchange
- if left element equal, exchange to left end
- if right element equal, exchange to right end

Swap equals to center after partition

<table>
<thead>
<tr>
<th>less</th>
<th>equal</th>
<th>greater</th>
</tr>
</thead>
</table>

KEY FEATURES
- always uses N-1 (three-way) compares
- no extra overhead if no equal keys
- only one “extra” exchange per equal key
**Complexity**

- In Figure 8, notice that QuickSort and MergeSort have a similar program structure.
- Both have $O(n)$ work to either
  1. Divide into chunks or
  2. Put the chunks back together.
- The pictures we draw (for expected case) look the same.
- The formula analysis looks the same.
- We see it doesn’t matter whether we do the work before the recursion or after. The work is the same.

**Quickselect**

- Find the $k^{th}$ smallest element in an array of $N$ items.
- We can do better than sorting the array first.
- The solution is to have an algorithm that is similar to QuickSort but with only one recursive call.
- The steps of the algorithm are as follows:
  1. If the number of elements in $S$ is 1, return the single element in $S$.
  2. Pick any element $v$ in $S$. It is the pivot.
  3. Partition $S - \{v\}$ into $L$ and $R$, exactly as was done with quicksort.
  4. If $k \leq$ number of elements in $L$, the item we are searching for must be in $L$. Call $QuickSelect(L, k)$ recursively. Otherwise, if $k$ is exactly equal to 1 more than the number of items in $L$, the pivot is the $k^{th}$ smallest element, and we can return it as the answer. Otherwise, the $k^{th}$ smallest element lies in $R$, and it is the $(k - |L| - 1)^{th}$ smallest element in $R$. Again, we can make a recursive call and return the result.
- Notice that only one recursive call is made.
- If the pivot is chosen correctly, it can be shown, that even in the worst case, the running time is linear.

```c
int QuickSelect(a[], low, high, k) {
    if (low > high) return ERROR
    pivot = partition(a, low, high)
    if (pivot-low == k) return a[pivot];
    if (k, pivot-low) return QuickSelect(a, low, pivot-1, k)
    return QuickSelect(a, pivot+1, high, k-pivot-low-1)
} // QuickSelect
```

```c
QuickSort(a[], low, high) {
    pivot = partition(a, low, high)
    QuickSort(a, low, pivot-1)
    QuickSort(a, pivot+1, high)
} // QuickSort
```

```c
MergeSort(a[], low, high) {
    mid=(low+high)/2
    MergeSort(a, low, mid)
    MergeSort(a, mid+1, high)
    Merge(a, low, mid, high)
} // MergeSort
```

Figure 8  QuickSort and MergeSort
**Timsort**

invented by Tim Peters 2002 for use in Python

Takes advantage of natural order. Less than $O(n \log n)$ on some kinds of inputs. Less than $\log(n!)$

$$\log n! \approx n \log n - n + \frac{\log(n(1 + 4n(1 + 2n)))}{6} + \frac{\log(\pi)}{2}.$$

1. adaptive, stable, natural mergesort
2. Uses insertion sort

Important features:

1. Goes through array, finding runs that are in order, and merging them with previous runs intelligently.
2. Goal is to have runs of the same size – as merge time depends on the length of the longer run. If runs are too small, extends to length MINRUN using insertion sort to add following elements to the run. Using natural runlengths can become wildly unbalanced.
3. Allows descending runs – and just swaps in place.

The desire for sort to be stable constrains permissible merging patterns. For example, if we have 3 consecutive runs (A, B and C) of lengths

A:10000  B:20000  C:10000

we dare not merge A with C first, because if A, B and C happen to contain a common element, it would get out of order with respect to its occurrence(s) in B. The merging must be done as (A+B)+C or A+(B+C) instead.

When a run is identified, its base address and length are pushed on a stack in the MergeState struct. merge_collapse() is then called to see whether it should merge it with preceding run(s). We would like to delay merging as long as possible in order to exploit patterns that may come up later, but we like even more to do merging as soon as possible to exploit that the run just found is still high in the memory hierarchy. We also can't delay merging "too long" because it consumes memory to remember the runs that are still unmerged, and the stack has a fixed size.

In order to balance run lengths (while keeping a low bound on the number of runs) we maintain two invariants on the stack entries, where A, B and C are the lengths of the three rightmost not-yet merged slices:

1. $|A| > |B|+|C|$ (if not, merge something)
2. $|B| > |C|$ (if not, merge something)
Merge Memory
Merging adjacent runs of lengths A and B in-place is very difficult, but if we have temp memory
equal to min(|A|, |B|), it's easy.

If A is the smaller-sized chunk (function merge_low), copy A to a temp array, leave B alone, and
then we can do the obvious merge algorithm left to right.
There's always a free area in the original area equal to the number not yet merged from the temp array.

If B is the smaller chunk(merge_high), much the same, except (in order to minimize the amount
of extra space we need) that we need to merge right to left, copying B into a temp array and
starting the result at the right end of where B used to live.

Refinement: When we're about to merge adjacent runs A and B, we first do a form of binary
search to see where B[0] should end up in A. Elements in A preceding that point are already in
their final positions, effectively shrinking the size of A. Likewise we also search to see where
A[last] should end up in B, and elements of B after that point can also be ignored as they are in
the right place. This cuts the amount of temp memory also.

Merge Algorithm

merge_low() and merge_high() are where the bulk of the time is spent.

Merging (for merge_low) begins in the usual, obvious way, comparing the first element of A to
the first of B, and moving B[0] to the merge area if it's less than A[0], else moving A[0] to the
merge area. The only twist here is keeping track of how many times in a row "the winner"
comes from the same run.

If that count reaches MIN_GALLOP, we switch to "galloping mode". Here we search B for
where A[0] belongs, and move over all the B's before that point in one chunk to the merge area
(moving a chunk is much cheaper than moving separately), then move A[0] to the merge area.
Then we search A for where B[0] belongs, and similarly move a slice of A in one chunk. Then
back to searching B for where the new A[0] belongs, etc. We stay in galloping mode until both
searches find slices to copy less than MIN_GALLOP elements long, at which point we go back
to one-pair-at-a-time mode.

Galloping
Assume A is the shorter run. In galloping mode, we first look for A[0] in B. We do this via
"galloping by powers of two", comparing A[0] in turn to B[0], B[1], B[3], B[7], ..., B[2**j - 1],
..., until finding the k such that B[2**(k-1) - 1] < A[0] <= B[2**k - 1]. This takes at most
roughly \( \lg(|B|) \) comparisons, and, unlike a straight binary search, favors finding the right spot (which is likely early in \( B \)).

In picture below:
Now, looking at the marked progression from left to right:
- 1) timsort finds a descending run, and reverses the run in-place. This is done directly on the array of pointers, so seems "instant" from our vantage point.
- 2) The run is now boosted to length minrun using insertion sort.
- 3) No run is detected at the beginning of the next block, and insertion sort is used to sort the entire block. Note that the sorted elements at the bottom of this block are not treated specially - timsort doesn't detect runs that start in the middle of blocks being boosted to minrun.
- 4) Finally, mergesort is used to merge the runs.
Indirect Sorting

- If we are sorting an array whose elements are large, copying the items can be very expensive.
- Not all of our sorts did the same amount of moving of data. Which ones were particularly good about limiting the amount of moving?
- We can get around this by doing indirect sorting.
  - We have an additional array of pointers where each element of the pointer array points to an element in the original array.
  - When sorting, we compare keys in the original array but we swap the element in the pointer array.
- This saves a lot of copying at the expense of indirect references to the elements of the original array.
- We must still rearrange the final array. The text has a clever way of doing that. It involves pointer arithmetic and shuffles cycles of shifts. Basically,
  - Find a \( \text{ptr}[i] \) which is not pointing to the \( i^{th} \) element of \( a \).
  - Save the thing in the \( i^{th} \) element of \( a \).
  - Move the correct a element into the \( i^{th} \) location freeing up the \( \text{nextj} \) location
  - Now move the thing that wants to be in the \( \text{nextj} \) location to it. We know what that element is because it is \( \text{p[nextj]} \).
  - Repeat until we reach something that is in its final position. This shuffling could rearrange every element, but likely a smaller cycle of elements is shifted.