7 Sorting Algorithms

7.1 \(O(n^2)\) sorting algorithms

Reading: MAW 7.1 and 7.2

- Insertion sort:

\[
\begin{array}{cccc}
4 & 1 & 3 & 2 \\
1 & 4 & 3 & 2 \\
1 & 3 & 4 & 2 \\
1 & 2 & 3 & 4 \\
\end{array}
\]

Worst-case time: \(O(n^2)\).

- Selection sort:

\[
\begin{array}{cccc}
4 & 1 & 3 & 2 \\
1 & 4 & 3 & 2 \\
1 & 2 & 3 & 4 \\
1 & 2 & 3 & 4 \\
\end{array}
\]

Worst-case time: \(O(n^2)\).

- Bubble sort:

\[
\begin{array}{cccc}
4 & 1 & 3 & 2 \\
1 & 3 & 2 & 4 \\
1 & 2 & 3 & 4 \\
1 & 2 & 3 & 4 \\
\end{array}
\]

Worst-case time: \(O(n^2)\).

7.2 Shell sort

Reading: MAW 7.4

- Introduction:

Shell sort, also called diminishing increment sort, was designed by Donald Shell (1959). It is the first sorting algorithm to break the \(n^2\) barrier. Subquadratic time complexity has been proved recently.

- Idea:

Define a sequence: \(h_t > h_{t-1} > \cdots > h_2 > h_1 = 1\).

Pass 1: \(h_t\)-sort the array, i.e., \(A[i] \leq A[i + h_t]\) for any \(i\).

Pass 2: \(h_{t-1}\)-sort the array, i.e., \(a A[i] \leq A[i + h_{t-1}]\) for any \(i\).

\[\cdots\]

Pass \(t\): \(h_1\)-sort the array, i.e., \(A[i] \leq A[i + 1]\) for any \(i\).

In each pass, use a modified insertion sort.

- Example: Use sequence 5, 3, 1.

\[
\begin{array}{cccccccccccccccc}
81 & 94 & 11 & 96 & 12 & 35 & 17 & 95 & 28 & 58 & 41 & 75 & 15 \\
35 & 17 & 11 & 28 & 12 & 41 & 75 & 15 & 96 & 58 & 81 & 94 & 95 \\
28 & 12 & 11 & 35 & 15 & 41 & 58 & 17 & 94 & 75 & 81 & 96 & 95 \\
11 & 12 & 15 & 17 & 28 & 35 & 41 & 58 & 75 & 81 & 94 & 95 & 96 \\
\end{array}
\]
• How to define the $h$-sequence:
  
  – Shell’s sequence: $h_t = \left\lfloor \frac{t}{2} \right\rfloor$ and $h_k = \left\lfloor \frac{h_{k-1}}{2} \right\rfloor$ for $k = t - 1, \ldots, 1$. For example, when $n = 13$, the sequence is 6, 3, 1.
  – Hibbard’s sequence: $2^t - 1, \ldots, 15, 7, 3, 1$.

• Time complexity: Determined by the sequence used.
  
  – Shell’s sequence: $O(n^2)$.
  – Hibbard’s sequence: $O(n^{1.5})$.
  – No sequence gives $O(n \log n)$ time complexity.

7.3 $O(n \log n)$ sorting algorithms

Reading: MAW 7.5 and 7.6

• Heap sort: Use a max heap.

  4 1 3 2 16 9 10 14 8 7
  16 14 10 8 7 9 3 2 4 1
  1 14 10 8 7 9 3 2 4 (16)
  14 8 10 4 7 9 3 2 1 (16)
  1 8 10 4 7 9 3 2 (14 16)
  10 8 9 4 7 1 3 2 (14 16)
  2 8 9 4 7 1 3 (10 14 16)
  ……

HeapSort(A)
  initialize array A into a heap
  for i=n to 2
    swap A[1] and A[i]
    make A[1..i-1] into a heap by calling RebuildHeap

Time complexity:
$O(n) + nO(\log n) = O(n \log n)$.

• Merge sort:
  
  – Recursive implementation: Top-down

    (8 3 5 7)
    (8 3) (5 7)
    (8) (3) (5) (7)
    (3 8) (5 7)
    (3 5 7 8)

MergeSort(A, 1, u)
  if 1 < u
    mid = (l + u) / 2
    MergeSort(A, 1, mid)
    MergeSort(A, mid + 1, u)
    merge the sorted A[1..mid] and the sorted A[mid+1..u]
      into one sorted array
How to merge two sorted lists into one?

Time complexity:
\[ T(n) = 2T\left(\frac{n}{2}\right) + n = O(n \log n) . \]

- Nonrecursive implementation: Bottom-up

\[
(3 \ 5 \ 7 \ 8) \\
(3 \ 8) (5 \ 7) \\
(8) (3) (5) (7)
\]

## 7.4 Quick sort

*Reading: MAW 7.7*


  \( A \) is divided into \( A_1 \) and \( A_2 \) such that for any \( x \) in \( A_1 \) and any \( y \) in \( A_2 \), we have \( x \leq y \).

  Then \( A_1 \) and \( A_2 \) are sorted recursively with the same method.

- Algorithm:

  ```plaintext
  QuickSort(A, l, u)
  if l < u
    partition A[l..u] into
    A[l..k] and A[k+1..u]
    QuickSort(A, l, k)
    QuickSort(A, k+1, u)
  ```

- How to partition \( A \):

  Method 1: Pick a number from \( A \), called a pivot \( p \). Create two new arrays \( A_1 \) and \( A_2 \). For numbers in \( A \) less than \( p \), add to \( A_1 \). For numbers in \( A \) greater than or equal to \( p \), add to \( A_2 \). Write \( A_1 \) back to \( A[l..k] \) and \( A_2 \) back to \( A[k+1..u] \).

  Note that \( k \) is determined by the number of items in \( A_1 \).

  Method 2: Without using auxiliary memory.

  ```plaintext
  Partition(A, l, u)
  pivot = A[l]
  i = l - 1
  j = u + 1
  while true
    repeat
      i ++
      until A[i] >= pivot
    repeat
      j --
      until A[j] <= pivot
    if i < j swap A[i] and A[j]
    else return j as k
  ```

  Time: \( O(n) \).

  Space: \( O(1) \).

  Example: \( 5 \ 3 \ 2 \ 6 \ 4 \ 1 \ 3 \ 7 \rightarrow (3 \ 3 \ 2 \ 1 \ 4) \ (6 \ 5 \ 7) \)

- Analysis of quick sort:

  - Worst-case: When \( A \) is already sorted and \( pivot \) is always the first number, \( O(n^2) \).
– Best-case: When \textit{pivot} is chosen such that the partition always yields two sublists of roughly the same size, \( T(n) = 2T(n/2) + O(n) \), giving \( O(n \log n) \).
– Average-case: \( O(n \log n) \).

- Choosing the pivot:
  – Random choice among all in the array, or
  – Median-of-three random choices.

7.5 \( O(n) \) sorting algorithms

\textit{Reading:} MAW 7.10 and 3.2

- Count sort (called Bucket sort by MAW):
  Let \( A \) be the input array with \( A[i] \) being an integer in \([1, k]\) for some known \( k \).
  Let \( B \) be the output array (sorted).
  For each \( i \) in \([1, k]\), determine \( C[i] \), the number of copies of \( i \) in \( A \).

\begin{verbatim}
CountSort(A)
    for i = 1 to k
        C[i] = 0
    for j = 1 to n
        C[A[j]] = C[A[j]] + 1
    for i = 1 to k
        for l = 1 to C[i]
            B[j] = i
            j ++
\end{verbatim}

Time complexity:
\( O(k + n) = O(n) \) if \( k = O(n) \).

- Radix sort:
  Let \( A \) be the input array, where \( A[i] \) is an integer with \( d \) digits in base-\( k \) representation, i.e., \( A[i] = b_{d}b_{d-1} \cdots b_{1}b_{0} \).

\begin{verbatim}
RadixSort(A)
    for j = 0 to k-1
        initialize queue Q[j]
    for j = 1 to d
        for i = 1 to n
            insert A[i] to Q[b_{j}(i)]
        for i = 0 to k-1
            delete from Q[i]
            write the numbers back to A
\end{verbatim}

Example:
329 427 839 436 720 355
720 355 436 427 329 839
720 427 329 436 839 355
329 355 427 436 720 839

Time complexity:
\( O(d(n + k)) = O(n) \) if \( d = O(1) \) and \( k = O(n) \).
Bucket sort:
Let $A[1..n]$ be the input array with $A[i]$ in $[0, 1)$ for all $i$. Let $B[0..n-1]$ be an array of pointers with $B[i]$ pointing to a list of numbers in the range of $[\frac{i}{n}, \frac{i+1}{n})$. The algorithm contains three steps:

- Distribute the numbers in $A$ into the linked lists in $B$.
- Sort each linked list.
- Concatenate the linked lists into one sorted array for output.

\[\text{BucketSort}(A)\]
\[\text{for } i = 1 \text{ to } n\]
\[\quad \text{insert } A[i] \text{ into the list pointed by } B[j], \text{ where } j \text{ is the integer part of } nA[i]\]
\[\text{for } i = 0 \text{ to } n-1\]
\[\quad \text{sort list pointed by } B[i]\]
\[\text{concatenate lists } B[0], ..., B[n-1]\]

Example: $A$ is $0.78, 0.17, 0.39, 0.26, 0.72, 0.94, 0.61$ with $n = 7$.

Time: On average, when all the numbers in $A$ are randomly generated and distributed uniformly in $[0, 1)$, each linked list will have length 1, yielding $O(n)$ for the average case.

What is the worst-case time complexity?