Sorting

Contents

- The Sorting Problem
- The Structure of Data
- Sorting Algorithms
  - Bubble Sort, Insertion Sort, Quick Sort, Merge Sort, Radix Sort, Bucket Sort

The Sorting Problem

- The sorting problem:
  - Input: a sequence of numbers $\{a_1, a_2, \ldots, a_n)$
  - Output: a reordered sequence such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$
- Input sequence usually an *n*-element array

The Structure of Data

- Our numbers to be sorted are rarely isolated values
  - Usually part of a collection of data called a *record*
  - Each *record* contains a *key* (the value to be sorted)
  - Remaining data in the *record* called *satellite data*
- If records include large amounts of satellite data, our array may hold pointers to records
- The data structure is irrelevant to the sorting algorithm

Bubble Sort

- Popular sorting algorithm
- Repeatedly swap adjacent elements that are out of order
- Too slow for most problems
- Worst-case $O(n^2)$

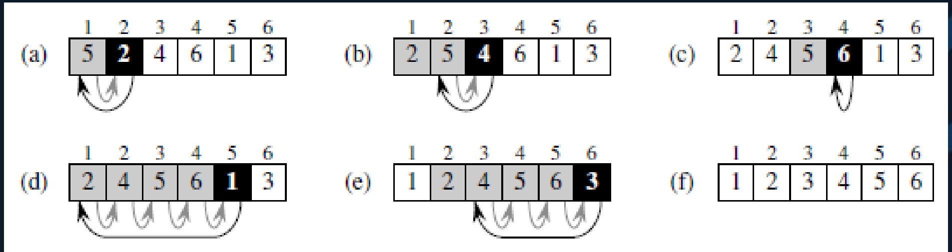
```
def BubbleSort(A)
  for i = 1 to length[A]
     for j = length[A] down to i + 1
        if A[j] < A[j-i]
           swap A[j] with A[j-i]
```

Canberra Institute of Technology

aie
SPECIALIST EDUCATORS IN GAMES, ANIMATION & FILM VFX

Insertion Sort

```
def InsertionSort(A)
  for i = 2 to length[A]
     key = A[i]
     j = i-1
     while j > 0 and A[j] > key
        A[j+1] = A[j]
        j = j-1
     A[j+1] = key
```



Insertion Sort: Analysis

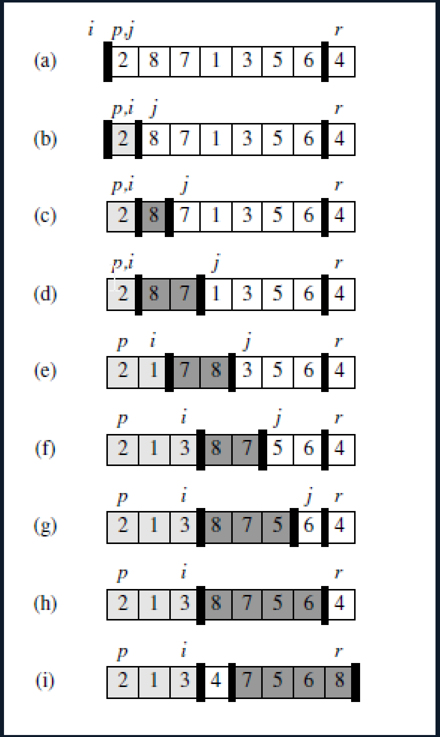
- Uses an *incremental approach*
- Time to sort increases as the array grows
- Time also differs (for two arrays of the same size) depending on how nearly sorted they are
- In-place sorting
- Worst-case running time of $O(n^2)$
  - But typically better than Bubble Sort, and with less swaps

Quick Sort

```
def QuickSort(A, p, r)
   if p < r then
      q = Partition(A, p, r)
      QuickSort(A, p, q-1)
      QuickSort(A, q + 1, r)

def Partition(A, p, r)
   x = A[r]
   i = p – 1
   for j = p to r – 1
      if A[j] ≤ x then
         i = i + 1
         swap values A[i] and A[j]
   swap values A[i+1] and A[r]
   return i + 1
```

- x = A[r] used as a *pivot* element around which to partition the subarray A[p..r]



Quick Sort

(a) The initial array and variable settings. None of the elements have been placed in either of the first two partitions.

(b) The value 2 is “swapped with itself” and put in the partition of smaller values.

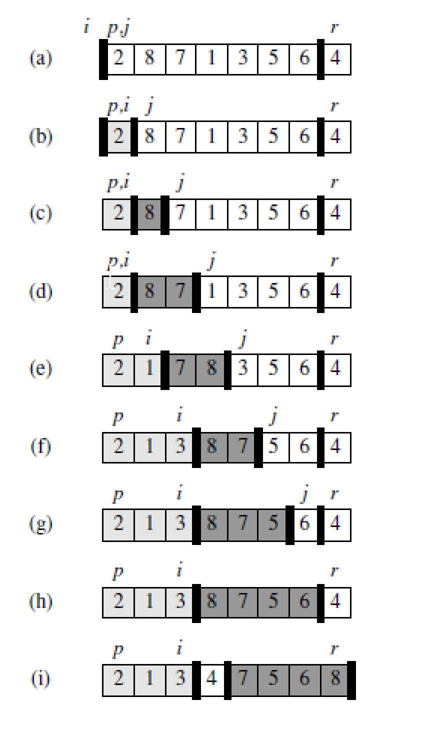
(c)–(d) The values 8 and 7 are added to the partition of larger values.

(e) The values 1 and 8 are swapped, and the smaller partition grows.

(f) The values 3 and 7 are swapped, and the smaller partition grows.

(g)–(h) The larger partition grows to include 5 and 6 and the loop terminates.

(i) In the last two lines, the pivot element is swapped so that it lies between the two partitions.

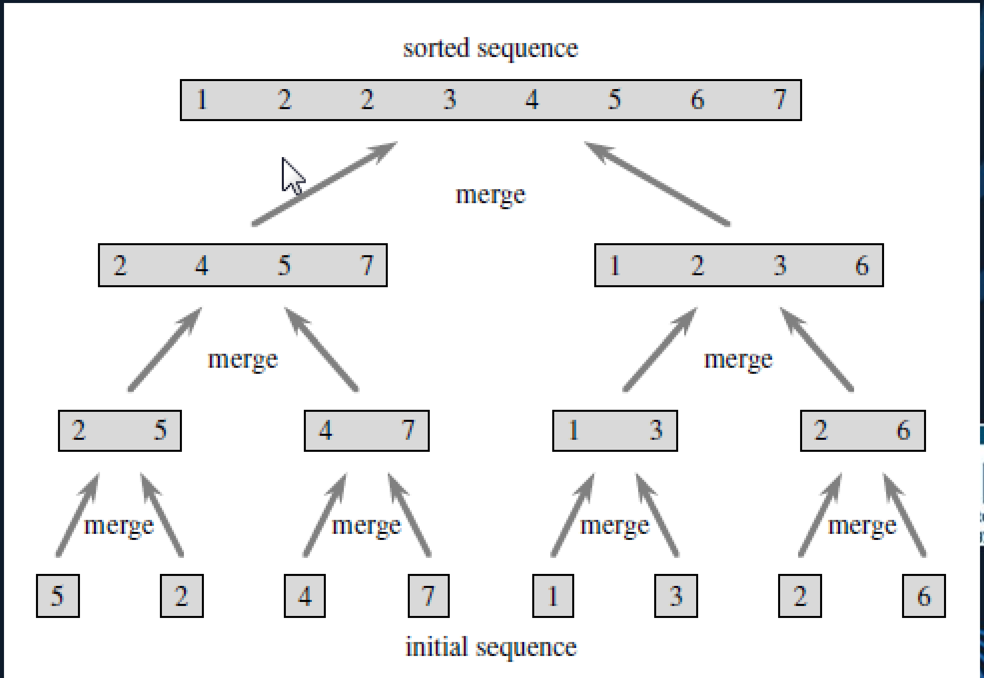


Quick Sort: Analysis

- Worst-case running time of $O(n^2)$
- Expected running time is $O(n \log n)$
- In-place sorting
- A divide and conquer algorithm
  - Problem divided into sub-problems
  - Conquer sub-problems by solving recursively
  - Combine solutions into final solution for original problem

Merge Sort

- Merge pairs of 1-item arrays to form a sorted array of length 2,
- Merge pairs of arrays of length 2 to form a sorted array of length 4,
- …and so on until
- 2 arrays of length n/2 are merged to form the final sorted array



Merge Sort

- A is an array
- p, q, and r are indices numbering elements such that p ≤ q < r
- Merge assumes subarrays A[p..q] and A[q+1..r] are in sorted order
- Subarrays merged to form current sub array A[p..r]

```
def Merge (A, p, q, r)
   leftEnd = q – p + 1
   rightEnd = r – q
   create array L[1..leftEnd + 1]
   create array R[1..leftEnd + 1]

   for i = 1 to leftEnd
      L[i] = A[p + i – 1]
   for j = 1 to rightEnd
      R[j] = A[q + j]

   L[leftEnd + 1] = empty
   R[rightEnd + 1] = empty
   i = 1
   j = 1

   for k = p to r
      if (R[j] is empty) or (L[i] not empty and L[i] ≤ R[j]) then
         A[k] = L[i]
         i = i + 1
      else
         A[k] = R[j]
         j = j + 1


def MergeSort(A, p, r)
   if p < r
      then q = (p+r)/2
         MergeSort(A, p, q)
         MergeSort(A, q+1, r)
         Merge(A, p, q, r)
```

Merge Sort: Analysis

- O(n log n)
- Better running time than Insertion Sort and Quick Sort
- Requires creating temporary arrays
- A divide and conquer algorithm

Radix Sort

```
def RadixSort(A, d)
   for i = 1 to d
      us a stable sort to sort array A on digit i
```

1. Take the least significant digit of each key.
2. Group the keys based on that digit, but otherwise keep the original order of keys.
3. Repeat the grouping process with each more significant digit.

Radix Sort: Analysis

- Given *n* *d*-digit numbers, in which each digit can take up to *k* possible values, Radix Sort sorts in O(d(n+k)) time

- May make fewer passes than Quick Sort over *n* keys, but each pass of Radix Sort may take longer
  - Depends on characteristics of implementation

Bucket Sort

- Partition an array into a number of buckets
- Each bucked sorted individually
- O(n) when input drawn from a uniform distribution

```
def BucketSort(A)
   n = length[A]
   for i = 1 to n
      k = most significant number for A[i]
      insert A[i] into list B[k]
   for i = 0 to n-1
      sort list B[i] with insertion sort
   concatenate lists B[0], B[1], .. , B[n-1]
```

Canberra Institute of Technology

aie SPECIALIST EDUCATORS IN GAMES, ANIMATION & FILM VFX

Summary

- When sorting, our input sequence is typically an *n*-element array
- Our array may hold key values, or references to records
- The best sorting algorithm to use depends largely on the application
- Several algorithms were presented here, however many more exist

References

- Thomas H. Cormen, 2001. *Introduction to Algorithms, Second Edition*. 2nd Edition. The MIT Press.

Canberra Institute of Technology

aie

SPECIALIST EDUCATORS IN GAMES, ANIMATION & FILM VFX