Algorithms and Data Structures

Advanced sorting algorithms

dr Szymon Murawski

Comarch SA

March 14, 2019

Table of contents I

Sorting so far

Algorithms

- Selection sort
- Insertion sort
- Bubble sort
- Cocktail sort

- All the algorithms we learned so far are quite bad, they have time complexity of $O(n^2)$, which makes them unsuitable for sorting large amounts of data
- They are however very easy to implement
- They also require little overhead, which makes them usable for very small arrays
- Next we will learn about advanced sorting algorithms, which all perform much better then the previous ones!

Course plan

Divide and conquer sort

Divide and Conquer approach

- **Divide** the problem into smaller subproblems
- **Conquer** the subproblems by solving them recursively
- **Combine** the solutions to the subproblems into the solution to the original problem

Merge sort

- **Divide** the n-element sequence to be sorted into two subsequences of $n/2$ elements each.
- **Conquer** by sorting the subsequences recursively using merge sort.
- **Merge** the two sorted subsequences to produce the sorted answer

Merge sort

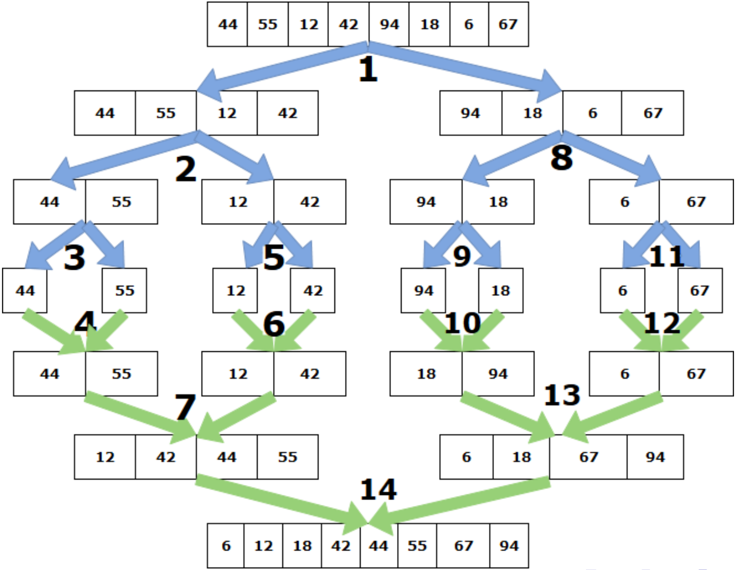
Pseudocode

- Input: Array A, start index p, end index r
- Output: Sorted array A

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

- Merge sort applies divide and conquer approach to the problem of sorting
- It is recursive algorithm with base case being $p = r$, which means an array of just one number.
- It should be obvious that an array with only one number is already sorted:)
- Pseudocode is quite simple, but the real magic happens inside *Merge* procedure

Merge sort graph



Merge sort call stack

```
MergeSort(A, 0, 7)
MergeSort(A, 0, 3)
MergeSort(A, 0, 1)
MergeSort(A, 0, 0)
MergeSort(A, 1, 1)
Merge(A, 0, 0, 1)
MergeSort(A, 2, 3)
MergeSort(A, 2, 2)
MergeSort(A, 3, 3)
Merge(A, 2, 2, 3)
Merge(A, 0, 1, 3)
MergeSort(A, 4, 7)
MergeSort(A, 4, 5)
MergeSort(A, 4, 4)
MergeSort(A, 5, 5)
Merge(A, 4, 4, 5)
MergeSort(A, 6, 7)
MergeSort(A, 6, 6)
MergeSort(A, 7, 7)
Merge(A, 6, 6, 7)
Merge(A, 4, 5, 7)
Merge(A, 0, 4, 7)
```

Merge procedure

- Real magic in merge sort happens in merge procedure
- Merge combines the two sorted sequences into one sorted sequence
- To do that it must create two temporary tables
- The procedure compares the first element in each of those arrays and pulls the smallest in to the output array
- If one array is empty, then procedure just copies all the elements from the second array

Merge procedure pseudocode

```
Merge(A, p, q, r):
  n1 = q - p + 1;
  n2 =  r - q;
  L[n1] = A[p..q]
  R[n2] = A[q+1..r] //temp array
  /* Merge the temp arrays back into A[p..r]*/
  k = p;
  while (i < n1 && j < n2)
      if (L[i] <= R[j])
          A[k] = L[i];
          i++;
  else
          A[k] = R[j];
          j++;
      k++;
  //Copy the remaining elements of L[], if there are any
  while (i < n1)
    A[k] = L[i]; i++; k++;
  //Copy the remaining elements of R[], if there are any
  while (j < n2)
    A[k] = R[j]; j++; k++;
```

Merge sort analysis

- First of the advanced algorithms
- Complexity $O(nlgn)$
- Complexity is the same in all cases - best, worst or average
- Stable sorting
- Out of place sorting, requires about $O(n)$ additional space
- Perfect, when the data can only be accessed sequentially (linked lists)
- We can use sentinel to skip checking whether a subarray is empty

Course plan

Quicksort

- Developed in 1959 by Tony Hoare, while he was a visiting student in Moscow
- Similar to merge sort in that it also divides original array into two subarrays
- Array is divided based on a pivot element - all elements lesser than the pivot form one subsequence, all larger form second subsequnce
- Choosing the right pivot has dramatic effect on the whole algorithm!

Pseudocode

- Input: Array A, start index p, end index r
- Output: Sorted array A

```
QuickSort(A, p, r):
 if p < r
    pivot = Partition(A, p, r)
    QuickSort(A, p, pivot)
    QuickSort(A, pivot+1, r)
```

Partition procedure

Pseudocode

```
Partiton(A, p, r)
  pivot = A[r]
  i = p - 1
    for( j = p; j < r -1; j-- )
      if A[j] <= x
        i++
        swap A[i] with A[j]
    swap A[i+1] with A[r]
  return i + 1
```

- i is the last index of array of smaller than pivot numbers
- j is the last index of array of larger than pivot numbers
- $A[j+1...r-1]$ is the undivided yet part of array

Partition example

$i = -1$	42	55	12	67	94	18	6	44

Partition example

$i = -1$	42	55	12	67	94	18	6	44
$i = 0$	42	55	12	67	94	18	6	44

Partition example

$i = -1$	42	55	12	67	94	18	6	44
$i = 0$	42	55	12	67	94	18	6	44
$i = 0$	42	55	12	67	94	18	6	44

Partition example

$i = -1$	42	55	12	67	94	18	6	44
$i = 0$	42	55	12	67	94	18	6	44
$i = 0$	42	55	12	67	94	18	6	44
$i = 1$	42	12	55	67	94	18	6	44

Partition example

$i = -1$	42	55	12	67	94	18	6	44
$i = 0$	42	55	12	67	94	18	6	44
$i = 0$	42	55	12	67	94	18	6	44
$i = 1$	42	12	55	67	94	18	6	44
$i = 1$	42	12	55	67	94	18	6	44

Partition example

$i = -1$	42	55	12	67	94	18	6	44
$i = 0$	42	55	12	67	94	18	6	44
$i = 0$	42	55	12	67	94	18	6	44
$i = 1$	42	12	55	67	94	18	6	44
$i = 1$	42	12	55	67	94	18	6	44
$i = 1$	42	12	55	67	94	18	6	44

Partition example

$i = -1$	42	55	12	67	94	18	6	44
$i = 0$	42	55	12	67	94	18	6	44
$i = 0$	42	55	12	67	94	18	6	44
$i = 1$	42	12	55	67	94	18	6	44
$i = 1$	42	12	55	67	94	18	6	44
$i = 1$	42	12	55	67	94	18	6	44
$i = 2$	42	12	18	67	94	55	6	44

Partition example

$i = -1$	42	55	12	67	94	18	6	44
$i = 0$	42	55	12	67	94	18	6	44
$i = 0$	42	55	12	67	94	18	6	44
$i = 1$	42	12	55	67	94	18	6	44
$i = 1$	42	12	55	67	94	18	6	44
$i = 1$	42	12	55	67	94	18	6	44
$i = 2$	42	12	18	67	94	55	6	44
$i = 3$	42	12	18	6	94	55	67	44

Partition example

$i = -1$	42	55	12	67	94	18	6	44
$i = 0$	42	55	12	67	94	18	6	44
$i = 0$	42	55	12	67	94	18	6	44
$i = 1$	42	12	55	67	94	18	6	44
$i = 1$	42	12	55	67	94	18	6	44
$i = 1$	42	12	55	67	94	18	6	44
$i = 2$	42	12	18	67	94	55	6	44
$i = 3$	42	12	18	6	94	55	67	44
$i = 3$	42	12	18	6	44	55	67	94

Quicksort example

42	55	12	67	94	18	6	44
42	12	18	6	44	55	67	94
6	12	18	42	44	55	67	94
6	12	18	42	44	55	67	94
6	12	18	42	44	55	67	94
6	12	18	42	44	55	67	94
6	12	18	42	44	55	67	94

Pivot problem

- Depending on the chosen pivot we can get either balanced or imbalanced split:
  - Balanced split - both subarrays are roughly the same size
  - Imbalanced split - one subarray is much larger than the other
- Worst case scenario is one subarray of size zero
- Choosing the right pivot has dramatic impact on the complexity of the algorithm - in worst case scenario time complexity is $O(n^2)$, compared to $O(nlgn)$ for average case

Choosing pivot

- First/last element of the array - low cost of computing, but is vulnerable to data distribution
- Random element from the array - quite good, but good random number generator is needed
- Median of three elements - good, also quite fast to compute
- Median of five or more elements - not much better than median-of-3
- BFPRT (Blum-Floyd-Pratt-Rivest-Trajan) - guarantees logarithmic worst case, while also making average case slightly worse. Sometimes called median of medians

Quicksort analysis

- Complexity $O(nlgn)$ in average case
- In worst case complexity $O(n^2)$
- Worst case is quite rare, also can be mitigated by good pivot select strategy
- Unstable sort
- In-place sort
- Best algorithm for standard sorting, used widely in programming
- About two times faster than merge sort
- Uses tail recursion

Course plan

Lower bound for sorting

- All the algorithms described to this point operate on comparing two numbers, to determine the sort order
- It can be shown, that any comparison sort algorithm requires $O(nlgn)$ comparisons in the worst case.
- Asymptotically, merge and heapsort (will be introduced later) are optimal sorts (quicksort has $O(n^2)$ in the worst case)
- It is however possible to sort in linear time(!), but it requires some knowledge of the data we are about to sort

Counting sort

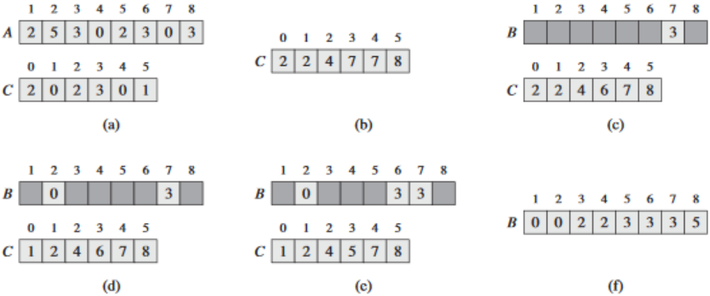
- Let's assume that we are given an array of numbers to sort. We also know that there are only k unique values among the number we are sorting
- We calculate count c_k how many each of the unique numbers appear in the input array
- Output array is constructed by just placing the k number c_k times, starting from the smallest k with non-zero c_k
- Notice, there are no comparisons here!

Counting sort

Pseudocode

```
CountingSort(A, k) :
int count[0..k] = 0
int output[]
  for (int i=0; i < length(A); i++)
    count[A[i]]++;
//count[x] now contains the number of elements equal to x
for (i=1; i<=k; i++)
  count[i] += count[i-1]
// count[x] now contains the number of elements less than or equal to x
for (i=length(A) - 1; i >= 0; i--)
  output[count[A[i]]-1] = A[i]
  count[A[i]]--
// We do it in reverse order for stability
return output
```

Counting sort simple example



Counting sort analysis

- Time complexity is $O(k + n)$.
- Usually k is of the order of n, we have linear time complexity!
- Does not use any comparison operators
- Stable sort
- Out of place sort - requires addition $O(n + k)$ space
- If $k \gg n$ then it is unpractical to use it
- Can only work on positive numbers - algorithm described before does not work if the numbers can be negative!

Counting sort with negative numbers

Pseudocode

```
CountingSortWithNegativeNumbers(A):
  max = max_element(A);
  min = min_element(A);
  int range = max - min + 1;
  int count[range];
  int output[length(A)];
  for(i = 0; i < length(A); i++)
    count[A[i]-min]++
  for(i = 1; i < count.size(); i++)
    count[i] += count[i-1];
  for(i = length(A); i >= 0; i--)
    output[ count[A[i]-min] -1 ] = A[i];
    count[A[i]-min]--;
return output
```

Course plan

Radix sort

- Counting sort is not usable when $k \gg n$.
- Still, we do not need to use comparison sorting algorithms!
- Assume we have to sort n numbers, each with at most d digits
- We can then use counting sort and sort digit by digit, starting with the least significant one!
- The counting sort needs to be stable!

```
RadixSort(A,d):
  for i=1; i<=d; i++
    CountingSort on digit d in A
```

Radix Sort example

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

Radix sort analysis

- Linear time, when input array is sufficiently larger than number of digits
- Out of place sorting
- Stable sort
- Radix sort can be faster than quicksort if the array is huge, or the keys small
- Still, radix sort is only usable on integer values
- Because of that two points it is often omitted in libraries

Course plan

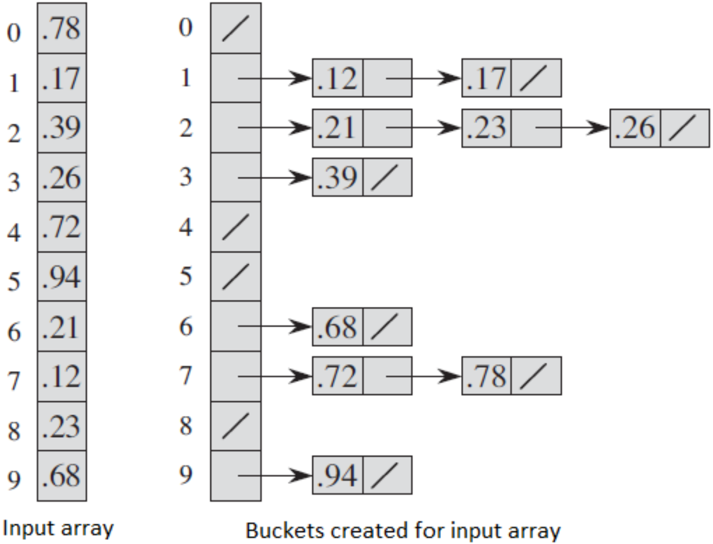
Bucket Sort

- Counting sort assumed that input is an array of small integers
- Bucket sort assumes that input data is generated by a random process, that distributes elements uniformly and independently over the interval $[0, 1)$
- Bucket sort divides the interval $[0, 1)$ into n egual-sized subintervals, called buckets, and then distributes the n input numbers into the buckets
- Since the inputs are uniformly and independently distributed over $[0, 1)$ we expect each bucket roughly the same size
- We sort the numbers in buckets using any other simple algorithm (like insertion sort), as each bucket should be fairly small

Bucket sort pseudocode

```
BucketSort(A):
  n=length(A)
  int[][] B[n-1] //This is array of arrays - our buckets
  for (i=0; i < n; i++)
    B[i] = empty array //initially every bucket is empty
  for (i=1; i <= n; i++)
    index = floor(nA[i])
    B[index].push(A[i])
  for (i=0; i<n; i++)
    sort(B[i])
  //array concatenation
  output = B[0] + B[1] + ... B[n-1]
```

Bucket sort example



Bucket sort analysis

- Complexity $O(n + \frac{n^2}{k} + k)$, where k is number of buckets
- Out of place sorting
- Stability of sorting depends on algorithm used to sort individual buckets
- Bucket sort is heavily dependent on the uniform distribution of data. In best case scenario it runs in linear time, but worst case is $O(n^2)$

Course plan

Summary of sorting algorithms

Name	Best	Average	Worst	Memory	Stable	In-place
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	Yes
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	No	Yes
Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	Yes
Cocktail sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	Yes
Merge sort	$O(nlg\ n)$	$O(nlg\ n)$	$O(nlg\ n)$	$O(n)$	Yes	No
Quicksort	$O(nlg\ n)$	$O(nlg\ n)$	$O(n^2$	$O(lg\ n)$	No	No
Heap sort	$O(nlg\ n)$	$O(nlg\ n)$	$O(nlg\ n)$	$O(1)$	No	Yes
Counting sort	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(k)$	Yes	No
Radix sort	$O(nk)$	$O(nk)$	$O(nk)$	$O(n+k)$	Yes	No
Bucket sort	$O(n+k)$	$O(n+k)$	$O(n^2)$	$O(n)$	Yes	No

https://www.youtube.com/watch?v=QOYcpGnHH0g

Choosing the right algorithm

- Choose insertion sort if the input is small, or elements are roughly sorted, or you need very simple code
- Choose heapsort if worst-case scenario is your main concern
- Choose quicksort in general case

General remarks

- In typical applications quicksort behaves better than heapsort
- Simple methods are simple, but not very effective, they should be used only in very specific cases
- For practical use it is best to combine two methods - quickosort for large input (with monitoring and switching to heapsort in worst-case scenario) and insertion sort for smaller subproblems (threshold chosen empirically for a given platform).