Contents

Chapter 1

Interpolation search vs Binary search

Interpolation search works better than Binary Search for a sorted and uniformly distributed array.

On average the interpolation search makes about log(log(n)) comparisons (if the elements are uniformly distributed), where n is the number of elements to be searched. In the worst case (for instance where the numerical values of the keys increase exponentially) it can make up to O(n) comparisons.

Sources:
http://en.wikipedia.org/wiki/Interpolation_search

Source

http://www.geeksforgeeks.org/g-fact-84/

Category: Arrays

Chapter 2

Why is Binary Search preferred over Ternary Search?

The following is a simple recursive **Binary Search** function in C++ taken from here.

```
// A recursive binary search function. It returns location of x in
// given array arr[l..r] is present, otherwise -1
int binarySearch(int arr[], int l, int r, int x)
{
   if (r >= l)
   {
        int mid = l + (r - l)/2;

        // If the element is present at the middle itself
        if (arr[mid] == x)  return mid;

        // If element is smaller than mid, then it can only be present
        // in left subarray
        if (arr[mid] > x) return binarySearch(arr, l, mid-1, x);

        // Else the element can only be present in right subarray
        return binarySearch(arr, mid+1, r, x);
   }

   // We reach here when element is not present in array
   return -1;
}
```

The following is a simple recursive **Ternary Search** function in C++.

```
// A recursive ternary search function. It returns location of x in
// given array arr[l..r] is present, otherwise -1
int ternarySearch(int arr[], int l, int r, int x)
{
   if (r >= l)
```

```
    {
         int mid1 = l + (r - l)/3;
         int mid2 = mid1 + (r - l)/3;

         // If x is present at the mid1
         if (arr[mid1] == x)  return mid1;

         // If x is present at the mid2
         if (arr[mid2] == x)  return mid2;

         // If x is present in left one-third
         if (arr[mid1] > x) return ternarySearch(arr, l, mid1-1, x);

         // If x is present in right one-third
         if (arr[mid2] < x) return ternarySearch(arr, mid2+1, r, x);

         // If x is present in middle one-third
         return ternarySearch(arr, mid1+1, mid2-1, x);
    }
    // We reach here when element is not present in array
    return -1;
}
```

Which of the above two does less comparisons in worst case?
From the first look, it seems the ternary search does less number of comparisons as it makes Log_3n recursive calls, but binary search makes Log_2n recursive calls. Let us take a closer look.
The following is recursive formula for counting comparisons in worst case of Binary Search.

```
T(n) = T(n/2) + 2,  T(1) = 1
```

The following is recursive formula for counting comparisons in worst case of Ternary Search.

```
T(n) = T(n/3) + 4, T(1) = 1
```

In binary search, there are $2Log_2n + 1$ comparisons in worst case. In ternary search, there are $4Log_3n + 1$ comparisons in worst case.

Therefore, the comparison of Ternary and Binary Searches boils down the comparison of expressions $2Log_3n$ and Log_2n . The value of $2Log_3n$ can be written as $(2 / Log_23) * Log_2n$. Since the value of $(2 / Log_23)$ is more than one, Ternary Search does more comparisons than Binary Search in worst case.

Exercise:
Why Merge Sort divides input array in two halves, why not in three or more parts?

This article is contributed by **Anmol**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/binary-search-preferred-ternary-search/

Category: Arrays

Chapter 3

A Problem in Many Binary Search Implementations

Consider the following C implementation of Binary Search function, is there anything wrong in this?

```
// A iterative binary search function. It returns location of x in
// given array arr[l..r] if present, otherwise -1
int binarySearch(int arr[], int l, int r, int x)
{
    while (l <= r)
    {
        // find index of middle element
        int m = (l+r)/2;

        // Check if x is present at mid
        if (arr[m] == x) return m;

        // If x greater, ignore left half
        if (arr[m] < x) l = m + 1;

        // If x is smaller, ignore right half
        else r = m - 1;
    }

    // if we reach here, then element was not present
    return -1;
}
```

The above looks fine except one subtle thing, the expression "m = (l+r)/2 . It fails for large values of l and r. Specifically, it fails if the sum of low and high is greater than the maximum positive int value ($2^{31} - 1$). The sum overflows to a negative value, and the value stays negative when divided by two. In C this causes an array index out of bounds with unpredictable results.

What is the way to resolve this problem?
Following is one way:

```
        int mid = low + ((high - low) / 2);
```

Probably faster, and arguably as clear is (works only in Java, refer this):

```
int mid = (low + high) >>> 1;
```

In C and C++ (where you don't have the >>> operator), you can do this:

```
mid = ((unsigned int)low + (unsigned int)high)) >> 1
```

The similar problem appears in Merge Sort as well.

The above content is taken from google reasearch blog.

Please refer thisas well, it points out that the above solutions may not always work.

The above problem occurs when array length is 2^{30} or greater and the search repeatedly moves to second half of the array. This much size of array is not likely to appear most of the time. For example, when we try the below program with 32 bit Code Blocks compiler, we get compiler error.

```
int main()
{
    int arr[1<<30];
    return 0;
}
```

Output:

```
 error: size of array 'arr' is too large
```

Even when we try boolean array, the program compiles fine, but crashes when run in Windows 7.0 and Code Blocks 32 bit compiler

```
#include <stdbool.h>
int main()
{
    bool arr[1<<30];
    return 0;
}
```

Output: No compiler error, but crashes at run time.

Sources:

http://googleresearch.blogspot.in/2006/06/extra-extra-read-all-about-it-nearly.html
http://locklessinc.com/articles/binary_search/

This article is contributed by **Abhay Rathi**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/problem-binary-search-implementations/

Chapter 4

Stability in sorting algorithms

A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input unsorted array. Some sorting algorithms are stable by nature like Insertion sort, Merge Sort, Bubble Sort, etc. And some sorting algorithms are not, like Heap Sort, Quick Sort, etc.

However, any given sorting algo which is not stable can be modified to be stable. There can be sorting algo specific ways to make it stable, but in general, any comparison based sorting algorithm which is not stable by nature can be modified to be stable by changing the key comparison operation so that the comparison of two keys considers position as a factor for objects with equal keys.

References:

http://www.math.uic.edu/~leon/cs-mcs401-s08/handouts/stability.pdf
http://en.wikipedia.org/wiki/Sorting_algorithm#Stability

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/stability-in-sorting-algorithms/

Category: Misc

Chapter 5

Why Quick Sort preferred for Arrays and Merge Sort for Linked Lists?

Why is Quick Sortpreferred for arrays?
Below are recursive and iterative implementations of Quick Sort and Merge Sort for arrays.

Recursive Quick Sort for array.
Iterative Quick Sort for arrays.
Recursive Merge Sort for arrays
Iterative Merge Sort for arrays

Quick Sort in its general form is an in-place sort (i.e. it doesn't require any extra storage) whereas merge sort requires O(N) extra storage, N denoting the array size which may be quite expensive. Allocating and de-allocating the extra space used for merge sort increases the running time of the algorithm. Comparing average complexity we find that both type of sorts have O(NlogN) average complexity but the constants differ. For arrays, merge sort loses due to the use of extra O(N) storage space.

Most practical implementations of Quick Sort use randomized version. The randomized version has expected time complexity of O(nLogn). The worst case is possible in randomized version also, but worst case doesn't occur for a particular pattern (like sorted array) and randomized Quick Sort works well in practice.

Quick Sort is also a cache friendly sorting algorithm as it has good locality of reference when used for arrays.

Quick Sort is also tail recursive, therefore tail call optimizations is done.

Why is Merge Sortpreferred for Linked Lists?
Below are implementations of Quicksort and Mergesort for singly and doubly linked lists.

Quick Sort for Doubly Linked List

Quick Sort for Singly Linked List
Merge Sort for Singly Linked List

Merge Sort for Doubly Linked List

In case of linked lists the case is different mainly due to difference in memory allocation of arrays and linked lists. Unlike arrays, linked list nodes may not be adjacent in memory. Unlike array, in linked list, we can insert items in the middle in O(1) extra space and O(1) time. Therefore merge operation of merge sort can be implemented without extra space for linked lists.

In arrays, we can do random access as elements are continuous in memory. Let us say we have an integer (4-byte) array A and let the address of A[0] be x then to access A[i], we can directly access the memory at (x + i*4). Unlike arrays, we can not do random access in linked list. Quick Sort requires a lot of this kind of access. In linked list to access i'th index, we have to travel each and every node from the head to i'th node as we don't have continuous block of memory. Therefore, the overhead increases for quick sort. Merge sort accesses data sequentially and the need of random access is low.

Thanks to **Sayan Mukhopadhyay** for providing initial draft for above article. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/why-quick-sort-preferred-for-arrays-and-merge-sort-for-linked-lists/

Chapter 6

When does the worst case of Quicksort occur?

The answer depends on strategy for choosing pivot. In early versions of Quick Sort where leftmost (or rightmost) element is chosen as pivot, the worst occurs in following cases.

1) Array is already sorted in same order.
2) Array is already sorted in reverse order.
3) All elements are same (special case of case 1 and 2)

Since these cases are very common use cases, the problem was easily solved by choosing either a random index for the pivot, choosing the middle index of the partition or (especially for longer partitions) choosing the median of the first, middle and last element of the partition for the pivot. With these modifications, the worst case of Quick sort has less chances to occur, but worst case can still occur if the input array is such that the maximum (or minimum) element is always chosen as pivot.

References:
http://en.wikipedia.org/wiki/Quicksort

Source

http://www.geeksforgeeks.org/when-does-the-worst-case-of-quicksort-occur/

Category: Misc

Chapter 7

Lower bound for comparison based sorting algorithms

The problem of sorting can be viewed as following.

Input: A sequence of n numbers $<a_1, a_2, \ldots, a_n>$.
Output: A permutation (reordering) $<a'_1, a'_2, \ldots, a'_n>$ of the input sequence such that $a'_1 \; a'_2 \ldots\ldots \; '_n$.

A sorting algorithm is comparison based if it uses comparison operators to find the order between two numbers. Comparison sorts can be viewed abstractly in terms of decision trees. A decision tree is afull binary tree that represents the comparisons between elements that are performed by a particular sorting algorithm operating on an input of a given size. The execution of the sorting algorithm corresponds to tracing a path from the root of the decision tree to a leaf. At each internal node, a comparison $a_i \; a_j$ is made. The left subtree then dictates subsequent comparisons for $a_i \; a_j$, and the right subtree dictates subsequent comparisons for $a_i > a_j$. When we come to a leaf, the sorting algorithm has established the ordering. So we can say following about the decison tree.

1) Each of the $n!$ permutations on n elements must appear as one of the leaves of the decision tree for the sorting algorithm to sort properly.

2) Let x be the maximum number of comparisons in a sorting algorithm. The maximum height of the decison tree would be x. A tree with maximum height x has at most 2^x leaves.

After combining the above two facts, we get following relation.

```
     n!  <= 2^x

Taking Log on both sides.
     log2(n!)  <= x

Since log2(n!)  = Θ(nLogn),  we can say
     x = Ω(nLog2n)
```

Therefore, any comparison based sorting algorithm must make at least $nLog_2n$ comparisons to sort the input array, and Heapsort and merge sort are asymptotically optimal comparison sorts.

References:
Introduction to Algorithms, by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein

Source

http://www.geeksforgeeks.org/lower-bound-on-comparison-based-sorting-algorithms/

Chapter 8

Which sorting algorithm makes minimum number of memory writes?

Minimizing the number of writes is useful when making writes to some huge data set is very expensive, such as with EEPROMs or Flash memory, where each write reduces the lifespan of the memory.

Among the sorting algorithms that we generally study in our data structure and algorithm courses, Selection Sort makes least number of writes (it makes O(n) swaps). But, Cycle Sort almost always makes less number of writes compared to Selection Sort. In Cycle Sort, each value is either written zero times, if it's already in its correct position, or written one time to its correct position. This matches the minimal number of overwrites required for a completed in-place sort.

Sources:
http://en.wikipedia.org/wiki/Cycle_sort
http://en.wikipedia.org/wiki/Selection_sort

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/which-sorting-algorithm-makes-minimum-number-of-writes/

Category: Arrays

Chapter 9

Sort an array in wave form

Given an unsorted array of integers, sort the array into a wave like array. An array 'arr[0..n-1]' is sorted in wave form if arr[0] >= arr[1] = arr[3] = ...

Examples:

```
Input:  arr[] = {10, 5, 6, 3, 2, 20, 100, 80}
Output: arr[] = {10, 5, 6, 2, 20, 3, 100, 80} OR
                {20, 5, 10, 2, 80, 6, 100, 3} OR
                any other array that is in wave form

Input:  arr[] = {20, 10, 8, 6, 4, 2}
Output: arr[] = {20, 8, 10, 4, 6, 2} OR
                {10, 8, 20, 2, 6, 4} OR
                any other array that is in wave form

Input:  arr[] = {2, 4, 6, 8, 10, 20}
Output: arr[] = {4, 2, 8, 6, 20, 10} OR
                any other array that is in wave form

Input:  arr[] = {3, 6, 5, 10, 7, 20}
Output: arr[] = {6, 3, 10, 5, 20, 7} OR
                any other array that is in wave form
```

We strongly recommend to minimize your browser and try this yourself first.

A **Simple Solution** is to use sorting. First sort the input array, then swap all adjacent elements.

For example, let the input array be {3, 6, 5, 10, 7, 20}. After sorting, we get {3, 5, 6, 7, 10, 20}. After swapping adjacent elements, we get {5, 3, 7, 6, 20, 10}.

Below are implementations of this simple approach.

Chapter 10

C++

```
// A C++ program to sort an array in wave form using a sorting function
#include<iostream>
#include<algorithm>
using namespace std;

// A utility method to swap two numbers.
void swap(int *x, int *y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}

// This function sorts arr[0..n-1] in wave form, i.e.,
// arr[0] >= arr[1] <= arr[2] >= arr[3] <= arr[4] >= arr[5]..
void sortInWave(int arr[], int n)
{
    // Sort the input array
    sort(arr, arr+n);

    // Swap adjacent elements
    for (int i=0; i<n-1; i += 2)
        swap(&arr[i], &arr[i+1]);
}

// Driver program to test above function
int main()
{
    int arr[] = {10, 90, 49, 2, 1, 5, 23};
    int n = sizeof(arr)/sizeof(arr[0]);
    sortInWave(arr, n);
    for (int i=0; i<n; i++)
       cout << arr[i] << " ";
    return 0;
}
```

Chapter 11

Python

```
# Python function to sort the array arr[0..n-1] in wave form,
# i.e., arr[0] >= arr[1] <= arr[2] >= arr[3] <= arr[4] >= arr[5]
def sortInWave(arr, n):

    #sort the array
    arr.sort()

    # Swap adjacent elements
    for i in range(0,n-1,2):
        arr[i], arr[i+1] = arr[i+1], arr[i]

# Driver progrM
arr = [10, 90, 49, 2, 1, 5, 23]
sortInWave(arr, len(arr))
for i in range(0,len(arr)):
    print arr[i],

# This code is contributed by __Devesh Agrawal__
```

Output:

```
 2 1 10 5 49 23 90
```

The time complexity of the above solution is O(nLogn) if a O(nLogn) sorting algorithm like Merge Sort, Heap Sort, .. etc is used.

This can be done in **O(n) time by doing a single traversal** of given array. The idea is based on the fact that if we make sure that all even positioned (at index 0, 2, 4, ..) elements are greater than their adjacent odd elements, we don't need to worry about odd positioned element. Following are simple steps.
1) Traverse all even positioned elements of input array, and do following.
….a) If current element is smaller than previous odd element, swap previous and current.
….b) If current element is smaller than next odd element, swap next and current.

Below are implementations of above simple algorithm.

Chapter 12

C++

```
// A O(n) program to sort an input array in wave form
#include<iostream>
using namespace std;

// A utility method to swap two numbers.
void swap(int *x, int *y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}

// This function sorts arr[0..n-1] in wave form, i.e., arr[0] >=
// arr[1] <= arr[2] >= arr[3] <= arr[4] >= arr[5] ....
void sortInWave(int arr[], int n)
{
    // Traverse all even elements
    for (int i = 0; i < n; i+=2)
    {
        // If current even element is smaller than previous
        if (i>0 && arr[i-1] > arr[i] )
            swap(&arr[i], &arr[i-1]);

        // If current even element is smaller than next
        if (i<n-1 && arr[i] < arr[i+1] )
            swap(&arr[i], &arr[i + 1]);
    }
}

// Driver program to test above function
int main()
{
    int arr[] = {10, 90, 49, 2, 1, 5, 23};
    int n = sizeof(arr)/sizeof(arr[0]);
    sortInWave(arr, n);
    for (int i=0; i<n; i++)
       cout << arr[i] << " ";
```

```
    return 0;
}
```

Chapter 13

Python

```
# Python function to sort the array arr[0..n-1] in wave form,
# i.e., arr[0] >= arr[1] <= arr[2] >= arr[3] <= arr[4] >= arr[5]
def sortInWave(arr, n):

    # Traverse all even elements
    for i in range(0, n, 2):

        # If current even element is smaller than previous
        if (i> 0 and arr[i] < arr[i-1]):
            arr[i],arr[i-1]=arr[i-1],arr[i]

        # If current even element is smaller than next
        if (i < n-1 and arr[i] < arr[i+1]):
            arr[i],arr[i+1]=arr[i+1],arr[i]

# Driver program
arr = [10, 90, 49, 2, 1, 5, 23]
sortInWave(arr, len(arr))
for i in range(0,len(arr)):
    print arr[i],

# This code is contributed by __Devesh Agrawal__
```

Output:

```
 90 10 49 1 5 2 23
```

This article is contributed by **Shivam**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/sort-array-wave-form-2/

Chapter 14

Sort a nearly sorted (or K sorted) array

Given an array of n elements, where each element is at most k away from its target position, devise an algorithm that sorts in O(n log k) time.
For example, let us consider k is 2, an element at index 7 in the sorted array, can be at indexes 5, 6, 7, 8, 9 in the given array.

Source: Nearly sorted algorithm

We can **use Insertion Sort** to sort the elements efficiently. Following is the C code for standard Insertion Sort.

```
/* Function to sort an array using insertion sort*/
void insertionSort(int A[], int size)
{
   int i, key, j;
   for (i = 1; i < size; i++)
   {
       key = A[i];
       j = i-1;

       /* Move elements of A[0..i-1], that are greater than key, to one
          position ahead of their current position.
          This loop will run at most k times */
       while (j >= 0 && A[j] > key)
       {
           A[j+1] = A[j];
           j = j-1;
       }
       A[j+1] = key;
   }
}
```

The inner loop will run at most k times. To move every element to its correct place, at most k elements need to be moved. So overall *complexity will be O(nk)*

We can sort such arrays **more efficiently with the help of Heap data structure**. Following is the detailed process that uses Heap.
1) Create a Min Heap of size k+1 with first k+1 elements. This will take O(k) time (See this GFact)
2) One by one remove min element from heap, put it in result array, and add a new element to heap from remaining elements.

Removing an element and adding a new element to min heap will take Logk time. So overall complexity will be O(k) + O((n-k)*logK)

```
#include<iostream>
using namespace std;

// Prototype of a utility function to swap two integers
void swap(int *x, int *y);

// A class for Min Heap
class MinHeap
{
    int *harr; // pointer to array of elements in heap
    int heap_size; // size of min heap
public:
    // Constructor
    MinHeap(int a[], int size);

    // to heapify a subtree with root at given index
    void MinHeapify(int );

    // to get index of left child of node at index i
    int left(int i) { return (2*i + 1); }

    // to get index of right child of node at index i
    int right(int i) { return (2*i + 2); }

    // to remove min (or root), add a new value x, and return old root
    int replaceMin(int x);

    // to extract the root which is the minimum element
    int extractMin();
};

// Given an array of size n, where every element is k away from its target
// position, sorts the array in O(nLogk) time.
int sortK(int arr[], int n, int k)
{
    // Create a Min Heap of first (k+1) elements from
    // input array
    int *harr = new int[k+1];
    for (int i = 0; i<=k && i<n; i++) // i < n condition is needed when k > n
        harr[i] = arr[i];
    MinHeap hp(harr, k+1);

    // i is index for remaining elements in arr[] and ti
    // is target index of for cuurent minimum element in
    // Min Heapm 'hp'.
```

```
    for(int i = k+1, ti = 0; ti < n; i++, ti++)
    {
        // If there are remaining elements, then place
        // root of heap at target index and add arr[i]
        // to Min Heap
        if (i < n)
            arr[ti] = hp.replaceMin(arr[i]);

        // Otherwise place root at its target index and
        // reduce heap size
        else
            arr[ti] = hp.extractMin();
    }
}

// FOLLOWING ARE IMPLEMENTATIONS OF STANDARD MIN HEAP METHODS FROM CORMEN BOOK
// Constructor: Builds a heap from a given array a[] of given size
MinHeap::MinHeap(int a[], int size)
{
    heap_size = size;
    harr = a;  // store address of array
    int i = (heap_size - 1)/2;
    while (i >= 0)
    {
        MinHeapify(i);
        i--;
    }
}

// Method to remove minimum element (or root) from min heap
int MinHeap::extractMin()
{
    int root = harr[0];
    if (heap_size > 1)
    {
        harr[0] = harr[heap_size-1];
        heap_size--;
        MinHeapify(0);
    }
    return root;
}

// Method to change root with given value x, and return the old root
int MinHeap::replaceMin(int x)
{
    int root = harr[0];
    harr[0] = x;
    if (root < x)
        MinHeapify(0);
    return root;
}

// A recursive method to heapify a subtree with root at given index
// This method assumes that the subtrees are already heapified
```

```
void MinHeap::MinHeapify(int i)
{
    int l = left(i);
    int r = right(i);
    int smallest = i;
    if (l < heap_size && harr[l] < harr[i])
        smallest = l;
    if (r < heap_size && harr[r] < harr[smallest])
        smallest = r;
    if (smallest != i)
    {
        swap(&harr[i], &harr[smallest]);
        MinHeapify(smallest);
    }
}

// A utility function to swap two elements
void swap(int *x, int *y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}

// A utility function to print array elements
void printArray(int arr[], int size)
{
   for (int i=0; i < size; i++)
       cout << arr[i] << " ";
   cout << endl;
}

// Driver program to test above functions
int main()
{
    int k = 3;
    int arr[] = {2, 6, 3, 12, 56, 8};
    int n = sizeof(arr)/sizeof(arr[0]);
    sortK(arr, n, k);

    cout << "Following is sorted array\n";
    printArray (arr, n);

    return 0;
}
```

Output:

```
 Following is sorted array
2 3 6 8 12 56
```

The Min Heap based method takes O(nLogk) time and uses O(k) auxiliary space.

We can also **use a Balanced Binary Search Tree** instead of Heap to store K+1 elements. The insertand deleteoperations on Balanced BST also take O(Logk) time. So Balanced BST based method will also take O(nLogk) time, but the Heap bassed method seems to be more efficient as the minimum element will always be at root. Also, Heap doesn't need extra space for left and right pointers.

Please write comments if you find any of the above codes/algorithms incorrect, or find other ways to solve the same problem.

Source

http://www.geeksforgeeks.org/nearly-sorted-algorithm/

Category: Arrays

Post navigation

← Adobe Interview | Set 1 Microsoft Interview | Set 4 →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 15

Sort n numbers in range from 0 to n^2 - 1 in linear time

Given an array of numbers of size n. It is also given that the array elements are in range from 0 to $n^2 - 1$. Sort the given array in linear time.

```
Examples:
Since there are 5 elements, the elements can be from 0 to 24.
Input: arr[] = {0, 23, 14, 12, 9}
Output: arr[] = {0, 9, 12, 14, 23}

Since there are 3 elements, the elements can be from 0 to 8.
Input: arr[] = {7, 0, 2}
Output: arr[] = {0, 2, 7}
```

We strongly recommend to minimize the browser and try this yourself first.

Solution: If we use Counting Sort, it would take O(n^2) time as the given range is of size n^2. Using any comparison based sorting likeMerge Sort, Heap Sort, .. etc would take O(nLogn) time.
Now question arises how to do this in 0(n)? Firstly, is it possible? Can we use data given in question? n numbers in range from 0 to $n^2 - 1$?
The idea is to use Radix Sort. Following is standard Radix Sort algorithm.

```
1) Do following for each digit i where i varies from least
   significant digit to the most significant digit.
……..a) Sort input array using counting sort (or any stable
        sort) according to the i'th digit
```

Let there be d digits in input integers. Radix Sort takes O(d*(n+b)) time where b is the base for representing numbers, for example, for decimal system, b is 10. Since n^2-1 is the maximum possible value, the value of d would be $O(\log_b(n))$. So overall time complexity is $O((n+b)*O(\log_b(n))$. Which looks more than the time complexity of comparison based sorting algorithms for a large k. The idea is to change base b. If we set b as n, the value of $O(\log_b(n))$ becomes O(1) and overall time complexity becomes O(n).

```
arr[] = {0, 10, 13, 12, 7}

Let us consider the elements in base 5. For example 13 in
base 5 is 23, and 7 in base 5 is 12.
arr[] = {00(0), 20(10), 23(13), 22(12), 12(7)}

After first iteration (Sorting according to the last digit in
base 5),  we get.
arr[] = {00(0), 20(10), 12(7), 22(12), 23(13)}

After second iteration, we get
arr[] = {00(0), 12(7), 20(10), 22(12), 23(13)}
```

Following is C++ implementation to sort an array of size n where elements are in range from 0 to $n^2 - 1$.

```
#include<iostream>
using namespace std;

// A function to do counting sort of arr[] according to
// the digit represented by exp.
int countSort(int arr[], int n, int exp)
{

    int output[n]; // output array
    int i, count[n] ;
    for (int i=0; i < n; i++)
       count[i] = 0;

    // Store count of occurrences in count[]
    for (i = 0; i < n; i++)
        count[ (arr[i]/exp)%n ]++;

    // Change count[i] so that count[i] now contains actual
    // position of this digit in output[]
    for (i = 1; i < n; i++)
        count[i] += count[i - 1];

    // Build the output array
    for (i = n - 1; i >= 0; i--)
    {
        output[count[ (arr[i]/exp)%n] - 1] = arr[i];
        count[(arr[i]/exp)%n]--;
    }

    // Copy the output array to arr[], so that arr[] now
    // contains sorted numbers according to curent digit
    for (i = 0; i < n; i++)
        arr[i] = output[i];
}
```

```
// The main function to that sorts arr[] of size n using Radix Sort
void sort(int arr[], int n)
{
    // Do counting sort for first digit in base n. Note that
    // instead of passing digit number, exp (n^0 = 0) is passed.
    countSort(arr, n, 1);

    // Do counting sort for second digit in base n. Note that
    // instead of passing digit number, exp (n^1 = n) is passed.
    countSort(arr, n, n);
}

// A utility function to print an array
void printArr(int arr[], int n)
{
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
}

// Driver program to test above functions
int main()
{
    // Since array size is 7, elements should be from 0 to 48
    int arr[] = {40, 12, 45, 32, 33, 1, 22};
    int n = sizeof(arr)/sizeof(arr[0]);
    cout << "Given array is \n";
    printArr(arr, n);

    sort(arr, n);

    cout << "\nSorted array is \n";
    printArr(arr, n);
    return 0;
}
```

Output:

```
 Given array is
40 12 45 32 33 1 22
Sorted array is
1 12 22 32 33 40 45
```

How to sort if range is from 1 to n^2?
If range is from 1 to n n^2, the above process can not be directly applied, it must be changed. Consider n = 100 and range from 1 to 10000. Since the base is 100, a digit must be from 0 to 99 and there should be 2 digits in the numbers. But the number 10000 has more than 2 digits. So to sort numbers in a range from 1 to n^2, we can use following process.
1) Subtract all numbers by 1.
2) Since the range is now 0 to n^2, do counting sort twice as done in the above implementation.
3) After the elements are sorted, add 1 to all numbers to obtain the original numbers.

How to sort if range is from 0 to n^3 -1?
Since there can be 3 digits in base n, we need to call counting sort 3 times.

This article is contributed by **Bateesh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/sort-n-numbers-range-0-n2-1-linear-time/

Category: Arrays

Post navigation

← Rearrange an array so that arr[i] becomes arr[arr[i]] with O(1) extra space Rearrange a string so that all same characters become d distance away →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 16

Search in an almost sorted array

Given an array which is sorted, but after sorting some elements are moved to either of the adjacent positions, i.e., arr[i] may be present at arr[i+1] or arr[i-1]. Write an efficient function to search an element in this array. Basically the element arr[i] can only be swapped with either arr[i+1] or arr[i-1].

For example consider the array {2, 3, 10, 4, 40}, 4 is moved to next position and 10 is moved to previous position.

Example:

```
Input: arr[] =  {10, 3, 40, 20, 50, 80, 70}, key = 40
Output: 2
Output is index of 40 in given array

Input: arr[] =  {10, 3, 40, 20, 50, 80, 70}, key = 90
Output: -1
-1 is returned to indicate element is not present
```

A simple solution is to linearly search the given key in given array. Time complexity of this solution is O(n). We cab modify binary search to do it in O(Logn) time.

The idea is to compare the key with middle 3 elements, if present then return the index. If not present, then compare the key with middle element to decide whether to go in left half or right half. Comparing with middle element is enough as all the elements after mid+2 must be greater than element mid and all elements before mid-2 must be smaller than mid element.

Following is C++ implementation of this approach.

```
// C++ program to find an element in an almost sorted array
#include <stdio.h>

// A recursive binary search based function. It returns index of x in
// given array arr[l..r] is present, otherwise -1
int binarySearch(int arr[], int l, int r, int x)
{
   if (r >= l)
   {
        int mid = l + (r - l)/2;
```

```
        // If the element is present at one of the middle 3 positions
        if (arr[mid] == x)  return mid;
        if (mid > l && arr[mid-1] == x) return (mid - 1);
        if (mid < r && arr[mid+1] == x) return (mid + 1);

        // If element is smaller than mid, then it can only be present
        // in left subarray
        if (arr[mid] > x) return binarySearch(arr, l, mid-2, x);

        // Else the element can only be present in right subarray
        return binarySearch(arr, mid+2, r, x);
    }

    // We reach here when element is not present in array
    return -1;
}

// Driver program to test above function
int main(void)
{
    int arr[] = {3, 2, 10, 4, 40};
    int n = sizeof(arr)/ sizeof(arr[0]);
    int x = 4;
    int result = binarySearch(arr, 0, n-1, x);
    (result == -1)? printf("Element is not present in array")
                  : printf("Element is present at index %d", result);
    return 0;
}
```

Output:

```
 Element is present at index 3
```

Time complexity of the above function is O(Logn).

This article is contributed by **Abhishek**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/search-almost-sorted-array/

Category: Arrays

Chapter 17

Iterative Quick Sort

Following is a typical recursive implementation of Quick Sort that uses last element as pivot.

```
/* A typical recursive implementation of quick sort */

/* This function takes last element as pivot, places the pivot element at its
   correct position in sorted array, and places all smaller (smaller than pivot)
   to left of pivot and all greater elements to right of pivot */
int partition (int arr[], int l, int h)
{
    int x = arr[h];
    int i = (l - 1);

    for (int j = l; j <= h- 1; j++)
    {
        if (arr[j] <= x)
        {
            i++;
            swap (&arr[i], &arr[j]);
        }
    }
    swap (&arr[i + 1], &arr[h]);
    return (i + 1);
}

/* A[] --> Array to be sorted, l  --> Starting index, h  --> Ending index */
void quickSort(int A[], int l, int h)
{
    if (l < h)
    {
        int p = partition(A, l, h); /* Partitioning index */
        quickSort(A, l, p - 1);
        quickSort(A, p + 1, h);
    }
}
```

The above implementation can be optimized in many ways

1) The above implementation uses last index as pivot. This causes worst-case behavior on already sorted arrays, which is a commonly occurring case. The problem can be solved by choosing either a random index for the pivot, or choosing the middle index of the partition or choosing the median of the first, middle and last element of the partition for the pivot. (See thisfor details)

2) To reduce the recursion depth, recur first for the smaller half of the array, and use a tail call to recurse into the other.

3) Insertion sort works better for small subarrays. Insertion sort can be used for invocations on such small arrays (i.e. where the length is less than a threshold t determined experimentally). For example, this library implementation of qsort uses insertion sort below size 7.

Despite above optimizations, the function remains recursive and uses function call stack to store intermediate values of l and h. The function call stack stores other bookkeeping information together with parameters. Also, function calls involve overheads like storing activation record of the caller function and then resuming execution.

The above function can be easily converted to iterative version with the help of an auxiliary stack. Following is an iterative implementation of the above recursive code.

```
// An iterative implementation of quick sort
#include <stdio.h>

// A utility function to swap two elements
void swap ( int* a, int* b )
{
    int t = *a;
    *a = *b;
    *b = t;
}

/* This function is same in both iterative and recursive*/
int partition (int arr[], int l, int h)
{
    int x = arr[h];
    int i = (l - 1);

    for (int j = l; j <= h- 1; j++)
    {
        if (arr[j] <= x)
        {
            i++;
            swap (&arr[i], &arr[j]);
        }
    }
    swap (&arr[i + 1], &arr[h]);
    return (i + 1);
}

/* A[] --> Array to be sorted, l  --> Starting index, h  --> Ending index */
void quickSortIterative (int arr[], int l, int h)
{
    // Create an auxiliary stack
    int stack[ h - l + 1 ];
```

```
    // initialize top of stack
    int top = -1;

    // push initial values of l and h to stack
    stack[ ++top ] = l;
    stack[ ++top ] = h;

    // Keep popping from stack while is not empty
    while ( top >= 0 )
    {
        // Pop h and l
        h = stack[ top-- ];
        l = stack[ top-- ];

        // Set pivot element at its correct position in sorted array
        int p = partition( arr, l, h );

        // If there are elements on left side of pivot, then push left
        // side to stack
        if ( p-1 > l )
        {
            stack[ ++top ] = l;
            stack[ ++top ] = p - 1;
        }

        // If there are elements on right side of pivot, then push right
        // side to stack
        if ( p+1 < h )
        {
            stack[ ++top ] = p + 1;
            stack[ ++top ] = h;
        }
    }
}

// A utility function to print contents of arr
void printArr( int arr[], int n )
{
    int i;
    for ( i = 0; i < n; ++i )
        printf( "%d ", arr[i] );
}

// Driver program to test above functions
int main()
{
    int arr[] = {4, 3, 5, 2, 1, 3, 2, 3};
    int n = sizeof( arr ) / sizeof( *arr );
    quickSortIterative( arr, 0, n - 1 );
    printArr( arr, n );
    return 0;
}
```

Output:

```
1 2 2 3 3 3 4 5
```

The above mentioned optimizations for recursive quick sort can also be applied to iterative version.

1) Partition process is same in both recursive and iterative. The same techniques to choose optimal pivot can also be applied to iterative version.

2) To reduce the stack size, first push the indexes of smaller half.

3) Use insertion sort when the size reduces below a experimentally calculated threshold.

References:
http://en.wikipedia.org/wiki/Quicksort

This article is compiled by **Aashish Barnwal** and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/iterative-quick-sort/

Category: Arrays

Post navigation

← Output of C++ Program | Set 17 Longest Palindromic Substring | Set 1 →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 18

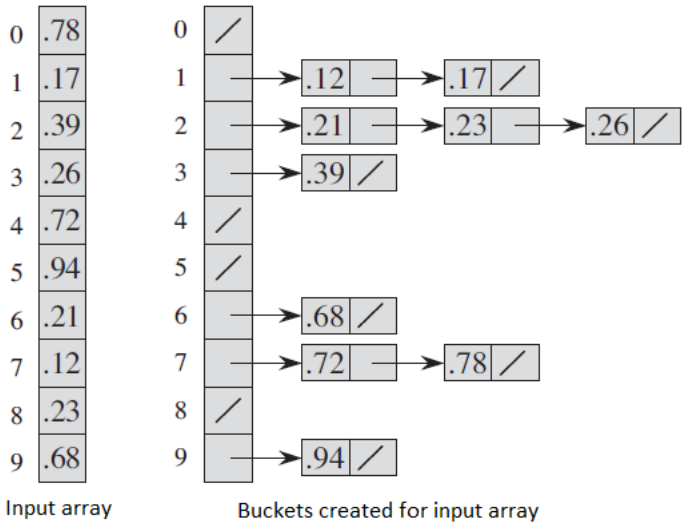
Bucket Sort

Bucket sort is mainly useful when input is uniformly distributed over a range. For example, consider the following problem.
Sort a large set of floating point numbers which are in range from 0.0 to 1.0 and are uniformly distributed across the range. How do we sort the numbers efficiently?

A simple way is to apply a comparison based sorting algorithm. The lower bound for Comparison based sorting algorithm (Merge Sort, Heap Sort, Quick-Sort .. etc) is $\Omega(n \text{ Log } n)$, i.e., they cannot do better than nLogn.
Can we sort the array in linear time? Counting sort can not be applied here as we use keys as index in counting sort. Here keys are floating point numbers.
The idea is to use bucket sort. Following is bucket algorithm.

```
bucketSort(arr[], n)
1) Create n empty buckets (Or lists).
2) Do following for every array element arr[i].
.......a) Insert arr[i] into bucket[n*array[i]]
3) Sort individual buckets using insertion sort.
4) Concatenate all sorted buckets.
```

Following diagram (taken from CLRS book) demonstrates working of bucket sort.



Time Complexity: If we assume that insertion in a bucket takes O(1) time then steps 1 and 2 of the above algorithm clearly take O(n) time. The O(1) is easily possible if we use a linked list to represent a bucket (In the following code, C++ vector is used for simplicity). Step 4 also takes O(n) time as there will be n items in all buckets.
The main step to analyze is step 3. This step also takes O(n) time on average if all numbers are uniformly distributed (please refer CLRS book for more details)

Following is C++ implementation of the above algorithm.

```
// C++ program to sort an array using bucket sort
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

// Function to sort arr[] of size n using bucket sort
void bucketSort(float arr[], int n)
{
    // 1) Create n empty buckets
    vector<float> b[n];

    // 2) Put array elements in different buckets
    for (int i=0; i<n; i++)
    {
       int bi = n*arr[i]; // Index in bucket
       b[bi].push_back(arr[i]);
    }

    // 3) Sort individual buckets
    for (int i=0; i<n; i++)
       sort(b[i].begin(), b[i].end());

    // 4) Concatenate all buckets into arr[]
    int index = 0;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < b[i].size(); j++)
```

```
            arr[index++] = b[i][j];
}

/* Driver program to test above funtion */
int main()
{
    float arr[] = {0.897, 0.565, 0.656, 0.1234, 0.665, 0.3434};
    int n = sizeof(arr)/sizeof(arr[0]);
    bucketSort(arr, n);

    cout << "Sorted array is \n";
    for (int i=0; i<n; i++)
       cout << arr[i] << " ";
    return 0;
}
```

Output:

```
 Sorted array is
0.1234 0.3434 0.565 0.656 0.665 0.897
```

References:

Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest

http://en.wikipedia.org/wiki/Bucket_sort

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/bucket-sort-2/

Chapter 19

Merge Sort for Linked Lists

Merge sort is often preferred for sorting a linked list. The slow random-access performance of a linked list makes some other algorithms (such as quicksort) perform poorly, and others (such as heapsort) completely impossible.

Let head be the first node of the linked list to be sorted and headRef be the pointer to head. Note that we need a reference to head in MergeSort() as the below implementation changes next links to sort the linked lists (not data at the nodes), so head node has to be changed if the data at original head is not the smallest value in linked list.

```
MergeSort(headRef)
1) If head is NULL or there is only one element in the Linked List
    then return.
2) Else divide the linked list into two halves.
      FrontBackSplit(head, &a, &b); /* a and b are two halves */
3) Sort the two halves a and b.
      MergeSort(a);
      MergeSort(b);
4) Merge the sorted a and b (using SortedMerge() discussed here)
   and update the head pointer using headRef.
     *headRef = SortedMerge(a, b);
```

```
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* function prototypes */
struct node* SortedMerge(struct node* a, struct node* b);
void FrontBackSplit(struct node* source,
          struct node** frontRef, struct node** backRef);
```

```
/* sorts the linked list by changing next pointers (not data) */
void MergeSort(struct node** headRef)
{
  struct node* head = *headRef;
  struct node* a;
  struct node* b;

  /* Base case -- length 0 or 1 */
  if ((head == NULL) || (head->next == NULL))
  {
    return;
  }

  /* Split head into 'a' and 'b' sublists */
  FrontBackSplit(head, &a, &b);

  /* Recursively sort the sublists */
  MergeSort(&a);
  MergeSort(&b);

  /* answer = merge the two sorted lists together */
  *headRef = SortedMerge(a, b);
}

/* See http://geeksforgeeks.org/?p=3622 for details of this
   function */
struct node* SortedMerge(struct node* a, struct node* b)
{
  struct node* result = NULL;

  /* Base cases */
  if (a == NULL)
     return(b);
  else if (b==NULL)
     return(a);

  /* Pick either a or b, and recur */
  if (a->data <= b->data)
  {
     result = a;
     result->next = SortedMerge(a->next, b);
  }
  else
  {
     result = b;
     result->next = SortedMerge(a, b->next);
  }
  return(result);
}

/* UTILITY FUNCTIONS */
/* Split the nodes of the given list into front and back halves,
     and return the two lists using the reference parameters.
```

```
     If the length is odd, the extra node should go in the front list.
     Uses the fast/slow pointer strategy.  */
void FrontBackSplit(struct node* source,
          struct node** frontRef, struct node** backRef)
{
  struct node* fast;
  struct node* slow;
  if (source==NULL || source->next==NULL)
  {
    /* length < 2 cases */
    *frontRef = source;
    *backRef = NULL;
  }
  else
  {
    slow = source;
    fast = source->next;

    /* Advance 'fast' two nodes, and advance 'slow' one node */
    while (fast != NULL)
    {
      fast = fast->next;
      if (fast != NULL)
      {
        slow = slow->next;
        fast = fast->next;
      }
    }

    /* 'slow' is before the midpoint in the list, so split it in two
      at that point. */
    *frontRef = source;
    *backRef = slow->next;
    slow->next = NULL;
  }
}

/* Function to print nodes in a given linked list */
void printList(struct node *node)
{
  while(node!=NULL)
  {
   printf("%d ", node->data);
   node = node->next;
  }
}

/* Function to insert a node at the beginging of the linked list */
void push(struct node** head_ref, int new_data)
{
  /* allocate node */
  struct node* new_node =
            (struct node*) malloc(sizeof(struct node));
```

```
  /* put in the data  */
  new_node->data  = new_data;

  /* link the old list off the new node */
  new_node->next = (*head_ref);

  /* move the head to point to the new node */
  (*head_ref)    = new_node;
}

/* Drier program to test above functions*/
int main()
{
  /* Start with the empty list */
  struct node* res = NULL;
  struct node* a = NULL;

  /* Let us create a unsorted linked lists to test the functions
   Created lists shall be a: 2->3->20->5->10->15 */
  push(&a, 15);
  push(&a, 10);
  push(&a, 5);
  push(&a, 20);
  push(&a, 3);
  push(&a, 2);

  /* Sort the above created Linked List */
  MergeSort(&a);

  printf("\n Sorted Linked List is: \n");
  printList(a);

  getchar();
  return 0;
}
```

Time Complexity: O(nLogn)

Sources:
http://en.wikipedia.org/wiki/Merge_sort
http://cslibrary.stanford.edu/105/LinkedListProblems.pdf

Please write comments if you find the above code/algorithm incorrect, or find better ways to solve the same problem.

Source

http://www.geeksforgeeks.org/merge-sort-for-linked-list/

Category: Linked Lists

Post navigation

← Identical Linked Lists Segregate Even and Odd numbers →

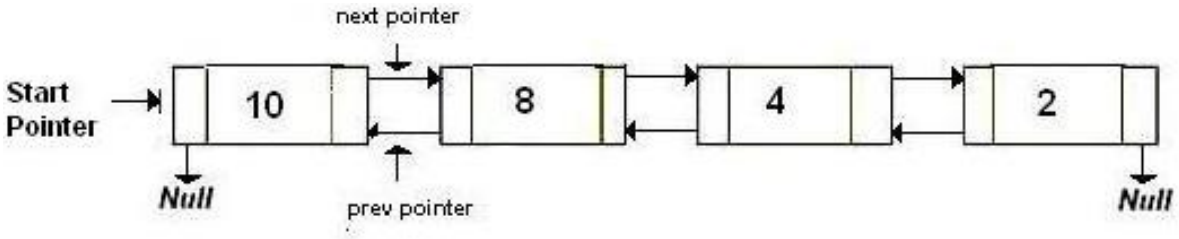
Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 20

Merge Sort for Doubly Linked List

Given a doubly linked list, write a function to sort the doubly linked list in increasing order using merge sort.

For example, the following doubly linked list should be changed to 24810



We strongly recommend to minimize your browser and try this yourself first.
Merge sort for singly linked list is already discussed. The important change here is to modify the previous pointers also when merging two lists.

Below is C implementation of merge sort for doubly linked list.

```
// C program for merge sort on doubly linked list
#include<stdio.h>
#include<stdlib.h>
struct node
{
    int data;
    struct node *next, *prev;
};

struct node *split(struct node *head);

// Function to merge two linked lists
struct node *merge(struct node *first, struct node *second)
{
    // If first linked list is empty
    if (!first)
        return second;
```

```
    // If second linked list is empty
    if (!second)
        return first;

    // Pick the smaller value
    if (first->data < second->data)
    {
        first->next = merge(first->next,second);
        first->next->prev = first;
        first->prev = NULL;
        return first;
    }
    else
    {
        second->next = merge(first,second->next);
        second->next->prev = second;
        second->prev = NULL;
        return second;
    }
}

// Function to do merge sort
struct node *mergeSort(struct node *head)
{
    if (!head || !head->next)
        return head;
    struct node *second = split(head);

    // Recur for left and right halves
    head = mergeSort(head);
    second = mergeSort(second);

    // Merge the two sorted halves
    return merge(head,second);
}

// A utility function to insert a new node at the
// beginning of doubly linked list
void insert(struct node **head, int data)
{
    struct node *temp =
        (struct node *)malloc(sizeof(struct node));
    temp->data = data;
    temp->next = temp->prev = NULL;
    if (!(*head))
        (*head) = temp;
    else
    {
        temp->next = *head;
        (*head)->prev = temp;
        (*head) = temp;
    }
}
```

```
// A utility function to print a doubly linked list in
// both forward and backward directions
void print(struct node *head)
{
    struct node *temp = head;
    printf("Forward Traversal using next poitner\n");
    while (head)
    {
        printf("%d ",head->data);
        temp = head;
        head = head->next;
    }
    printf("\nBackword Traversal using prev pointer\n");
    while (temp)
    {
        printf("%d ", temp->data);
        temp = temp->prev;
    }
}

// Utility function to swap two integers
void swap(int *A, int *B)
{
    int temp = *A;
    *A = *B;
    *B = temp;
}

// Split a doubly linked list (DLL) into 2 DLLs of
// half sizes
struct node *split(struct node *head)
{
    struct node *fast = head,*slow = head;
    while (fast->next && fast->next->next)
    {
        fast = fast->next->next;
        slow = slow->next;
    }
    struct node *temp = slow->next;
    slow->next = NULL;
    return temp;
}

// Driver program
int main(void)
{
    struct node *head = NULL;
    insert(&head,5);
    insert(&head,20);
    insert(&head,4);
    insert(&head,3);
    insert(&head,30);
    insert(&head,10);
    printf("Linked List before sorting\n");
```

```
    print(head);
    head = mergeSort(head);
    printf("\n\nLinked List after sorting\n");
    print(head);
    return 0;
}
```

Output:

```
 Linked List before sorting
Forward Traversal using next pointer
10 30 3 4 20 5
Backward Traversal using prev pointer
5 20 4 3 30 10

Linked List after sorting
Forward Traversal using next pointer
3 4 5 10 20 30
Backward Traversal using prev pointer
30 20 10 5 4 3
```

Thanks to Goku for providing above implementation in a comment here.

Time Complexity: Time complexity of the above implementation is same as time complexity of MergeSort for arrays. It takes Θ(nLogn) time.

You may also like to see QuickSort for doubly linked list

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/merge-sort-for-doubly-linked-list/

Category: Linked Lists

Chapter 21

QuickSort on Singly Linked List

QuickSort on Doubly Linked List is discussed here. QuickSort on Singly linked list was given as an exercise. Following is C++ implementation for same. The important things about implementation are, it changes pointers rather swapping data and time complexity is same as the implementation for Doubly Linked List.
In **partition()**, we consider last element as pivot. We traverse through the current list and if a node has value greater than pivot, we move it after tail. If the node has smaller value, we keep it at its current position.
In **QuickSortRecur()**, we first call partition() which places pivot at correct position and returns pivot. After pivot is placed at correct position, we find tail node of left side (list before pivot) and recur for left list. Finally, we recur for right list.

```
// C++ program for Quick Sort on Singly Linled List
#include <iostream>
#include <cstdio>
using namespace std;

/* a node of the singly linked list */
struct node
{
    int data;
    struct node *next;
};

/* A utility function to insert a node at the beginning of linked list */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node = new node;

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}
```

```
/* A utility function to print linked list */
void printList(struct node *node)
{
    while (node != NULL)
    {
        printf("%d  ", node->data);
        node = node->next;
    }
    printf("\n");
}

// Returns the last node of the list
struct node *getTail(struct node *cur)
{
    while (cur != NULL && cur->next != NULL)
        cur = cur->next;
    return cur;
}

// Partitions the list taking the last element as the pivot
struct node *partition(struct node *head, struct node *end,
                       struct node **newHead, struct node **newEnd)
{
    struct node *pivot = end;
    struct node *prev = NULL, *cur = head, *tail = pivot;

    // During partition, both the head and end of the list might change
    // which is updated in the newHead and newEnd variables
    while (cur != pivot)
    {
        if (cur->data < pivot->data)
        {
            // First node that has a value less than the pivot - becomes
            // the new head
            if ((*newHead) == NULL)
                (*newHead) = cur;

            prev = cur;
            cur = cur->next;
        }
        else // If cur node is greater than pivot
        {
            // Move cur node to next of tail, and change tail
            if (prev)
                prev->next = cur->next;
            struct node *tmp = cur->next;
            cur->next = NULL;
            tail->next = cur;
            tail = cur;
            cur = tmp;
        }
    }
```

```
    // If the pivot data is the smallest element in the current list,
    // pivot becomes the head
    if ((*newHead) == NULL)
        (*newHead) = pivot;

    // Update newEnd to the current last node
    (*newEnd) = tail;

    // Return the pivot node
    return pivot;
}


//here the sorting happens exclusive of the end node
struct node *quickSortRecur(struct node *head, struct node *end)
{
    // base condition
    if (!head || head == end)
        return head;

    node *newHead = NULL, *newEnd = NULL;

    // Partition the list, newHead and newEnd will be updated
    // by the partition function
    struct node *pivot = partition(head, end, &newHead, &newEnd);

    // If pivot is the smallest element - no need to recur for
    // the left part.
    if (newHead != pivot)
    {
        // Set the node before the pivot node as NULL
        struct node *tmp = newHead;
        while (tmp->next != pivot)
            tmp = tmp->next;
        tmp->next = NULL;

        // Recur for the list before pivot
        newHead = quickSortRecur(newHead, tmp);

        // Change next of last node of the left half to pivot
        tmp = getTail(newHead);
        tmp->next =  pivot;
    }

    // Recur for the list after the pivot element
    pivot->next = quickSortRecur(pivot->next, newEnd);

    return newHead;
}

// The main function for quick sort. This is a wrapper over recursive
// function quickSortRecur()
void quickSort(struct node **headRef)
{
```

```
    (*headRef) = quickSortRecur(*headRef, getTail(*headRef));
    return;
}

// Driver program to test above functions
int main()
{
    struct node *a = NULL;
    push(&a, 5);
    push(&a, 20);
    push(&a, 4);
    push(&a, 3);
    push(&a, 30);

    cout << "Linked List before sorting \n";
    printList(a);

    quickSort(&a);

    cout << "Linked List after sorting \n";
    printList(a);

    return 0;
}
```

Output:

```
 Linked List before sorting
30  3  4  20  5
Linked List after sorting
3  4  5  20  30
```

This article is contributed by **Balasubramanian.N**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/quicksort-on-singly-linked-list/

Category: Linked Lists

Post navigation

← [TopTalent.in] Interview with Manpreet Who Got Offers From Amazon, Hoppr, Browserstack, Reliance via TopTalent.in Print all possible strings of length k that can be formed from a set of n characters →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 22

QuickSort on Doubly Linked List

Following is a typical recursive implementation of QuickSort for arrays. The implementation uses last element as pivot.

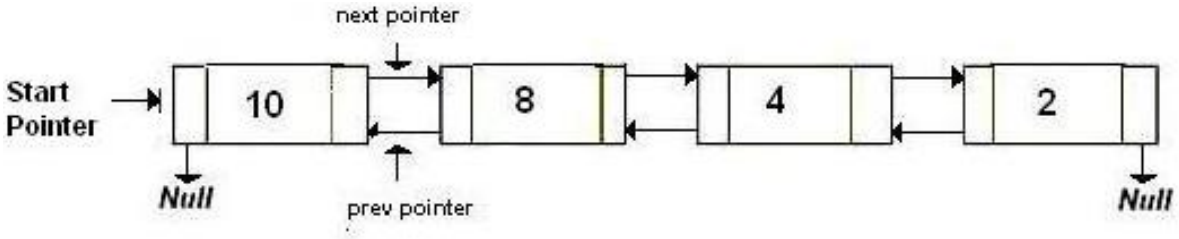
```
/* A typical recursive implementation of Quicksort for array*/

/* This function takes last element as pivot, places the pivot element at its
   correct position in sorted array, and places all smaller (smaller than
   pivot) to left of pivot and all greater elements to right of pivot */
int partition (int arr[], int l, int h)
{
    int x = arr[h];
    int i = (l - 1);

    for (int j = l; j <= h- 1; j++)
    {
        if (arr[j] <= x)
        {
            i++;
            swap (&arr[i], &arr[j]);
        }
    }
    swap (&arr[i + 1], &arr[h]);
    return (i + 1);
}

/* A[] --> Array to be sorted, l  --> Starting index, h  --> Ending index */
void quickSort(int A[], int l, int h)
{
    if (l < h)
    {
        int p = partition(A, l, h); /* Partitioning index */
        quickSort(A, l, p - 1);
        quickSort(A, p + 1, h);
    }
}
```

Can we use same algorithm for Linked List?
Following is C++ implementation for doubly linked list. The idea is simple, we first find out pointer to last node. Once we have pointer to last node, we can recursively sort the linked list using pointers to first and last nodes of linked list, similar to the above recursive function where we pass indexes of first and last array elements. The partition function for linked list is also similar to partition for arrays. Instead of returning index of the pivot element, it returns pointer to the pivot element. In the following implementation, quickSort() is just a wrapper function, the main recursive function is _quickSort() which is similar to quickSort() for array implementation.




```
// A C++ program to sort a linked list using Quicksort
#include <iostream>
#include <stdio.h>
using namespace std;

/* a node of the doubly linked list */
struct node
{
    int data;
    struct node *next;
    struct node *prev;
};

/* A utility function to swap two elements */
void swap ( int* a, int* b )
{   int t = *a;      *a = *b;       *b = t;   }

// A utility function to find last node of linked list
struct node *lastNode(node *root)
{
    while (root && root->next)
        root = root->next;
    return root;
}

/* Considers last element as pivot, places the pivot element at its
   correct position in sorted array, and places all smaller (smaller than
   pivot) to left of pivot and all greater elements to right of pivot */
node* partition(node *l, node *h)
{
    // set pivot as h element
    int x  = h->data;

    // similar to i = l-1 for array implementation
    node *i = l->prev;
```

```
    // Similar to "for (int j = l; j <= h- 1; j++)"
    for (node *j = l; j != h; j = j->next)
    {
        if (j->data <= x)
        {
            // Similar to i++ for array
            i = (i == NULL)? l : i->next;

            swap(&(i->data), &(j->data));
        }
    }
    i = (i == NULL)? l : i->next; // Similar to i++
    swap(&(i->data), &(h->data));
    return i;
}

/* A recursive implementation of quicksort for linked list */
void _quickSort(struct node* l, struct node *h)
{
    if (h != NULL && l != h && l != h->next)
    {
        struct node *p = partition(l, h);
        _quickSort(l, p->prev);
        _quickSort(p->next, h);
    }
}

// The main function to sort a linked list. It mainly calls _quickSort()
void quickSort(struct node *head)
{
    // Find last node
    struct node *h = lastNode(head);

    // Call the recursive QuickSort
    _quickSort(head, h);
}

// A utility function to print contents of arr
void printList(struct node *head)
{
    while (head)
    {
        cout << head->data << "  ";
        head = head->next;
    }
    cout << endl;
}

/* Function to insert a node at the beginging of the Doubly Linked List */
void push(struct node** head_ref, int new_data)
{
    struct node* new_node = new node;     /* allocate node */
    new_node->data  = new_data;
```

```
    /* since we are adding at the begining, prev is always NULL */
    new_node->prev = NULL;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* change prev of head node to new node */
    if ((*head_ref) !=  NULL)  (*head_ref)->prev = new_node ;

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* Driver program to test above function */
int main()
{
    struct node *a = NULL;
    push(&a, 5);
    push(&a, 20);
    push(&a, 4);
    push(&a, 3);
    push(&a, 30);

    cout << "Linked List before sorting \n";
    printList(a);

    quickSort(a);

    cout << "Linked List after sorting \n";
    printList(a);

    return 0;
}
```

Output :

```
 Linked List before sorting
30  3  4  20  5
Linked List after sorting
3  4  5  20  30
```

Time Complexity: Time complexity of the above implementation is same as time complexity of QuickSort() for arrays. It takes O(n^2) time in worst case and O(nLogn) in average and best cases. The worst case occurs when the linked list is already sorted.

Can we implement random quick sort for linked list?
Quicksort can be implemented for Linked List only when we can pick a fixed point as pivot (like last element in above implementation). Random QuickSort cannot be efficiently implemented for Linked Lists by picking random pivot.

Exercise:
The above implementation is for doubly linked list. Modify it for singly linked list. Note that we don't have

prev pointer in singly linked list.
Refer QuickSort on Singly Linked Listfor solution.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/quicksort-for-linked-list/

Category: Linked Lists

Post navigation

← B-Tree | Set 2 (Insert) Longest prefix matching – A Trie based solution in Java →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 23

Find k closest elements to a given value

Given a sorted array arr[] and a value X, find the k closest elements to X in arr[].
Examples:

```
Input: K = 4, X = 35
       arr[] = {12, 16, 22, 30, 35, 39, 42,
                45, 48, 50, 53, 55, 56}
Output: 30 39 42 45
```

Note that if the element is present in array, then it should not be in output, only the other closest elements are required.

In the following solutions, it is assumed that all elements of array are distinct.

A **simple solution** is to do linear search for k closest elements.
1) Start from the first element and search for the crossover point (The point before which elements are smaller than or equal to X and after which elements are greater). This step takes O(n) time.
2) Once we find the crossover point, we can compare elements on both sides of crossover point to print k closest elements. This step takes O(k) time.

The time complexity of the above solution is O(n).

An **Optimized Solution** is to find k elements in O(Logn + k) time. The idea is to use Binary Search to find the crossover point. Once we find index of crossover point, we can print k closest elements in O(k) time.

```
#include<stdio.h>

/* Function to find the cross over point (the point before
   which elements are smaller than or equal to x and after
   which greater than x)*/
int findCrossOver(int arr[], int low, int high, int x)
{
  // Base cases
  if (arr[high] <= x) // x is greater than all
    return high;
```

```
  if (arr[low] > x)  // x is smaller than all
    return low;

  // Find the middle point
  int mid = (low + high)/2;  /* low + (high - low)/2 */

  /* If x is same as middle element, then return mid */
  if (arr[mid] <= x && arr[mid+1] > x)
    return mid;

  /* If x is greater than arr[mid], then either arr[mid + 1]
    is ceiling of x or ceiling lies in arr[mid+1...high] */
  if(arr[mid] < x)
      return findCrossOver(arr, mid+1, high, x);

  return findCrossOver(arr, low, mid - 1, x);
}

// This function prints k closest elements to x in arr[].
// n is the number of elements in arr[]
void printKclosest(int arr[], int x, int k, int n)
{
    // Find the crossover point
    int l = findCrossOver(arr, 0, n-1, x); // le
    int r = l+1;   // Right index to search
    int count = 0; // To keep track of count of elements already printed

    // If x is present in arr[], then reduce left index
    // Assumption: all elements in arr[] are distinct
    if (arr[l] == x) l--;

    // Compare elements on left and right of crossover
    // point to find the k closest elements
    while (l >= 0 && r < n && count < k)
    {
        if (x - arr[l] < arr[r] - x)
            printf("%d ", arr[l--]);
        else
            printf("%d ", arr[r++]);
        count++;
    }

    // If there are no more elements on right side, then
    // print left elements
    while (count < k && l >= 0)
        printf("%d ", arr[l--]), count++;

    // If there are no more elements on left side, then
    // print right elements
    while (count < k && r < n)
        printf("%d ", arr[r++]), count++;
}

/* Driver program to check above functions */
```

```
int main()
{
   int arr[] ={12, 16, 22, 30, 35, 39, 42,
               45, 48, 50, 53, 55, 56};
   int n = sizeof(arr)/sizeof(arr[0]);
   int x = 35, k = 4;
   printKclosest(arr, x, 4, n);
   return 0;
}
```

Output:

```
 39 30 42 45
```

The time complexity of this method is O(Logn + k).

Exercise: Extend the optimized solution to work for duplicates also, i.e., to work for arrays where elements don't have to be distinct.

This article is contributed by **Rahul Jain**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/find-k-closest-elements-given-value/

Category: Arrays

Post navigation

← Can we use % operator on floating point numbers? Check if binary representation of a number is palindrome →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 24

Find the closest pair from two sorted arrays

Given two sorted arrays and a number x, find the pair whose sum is closest to x and **the pair has an element from each array**.

We are given two arrays ar1[0…m-1] and ar2[0..n-1] and a number x, we need to find the pair ar1[i] + ar2[j] such that absolute value of (ar1[i] + ar2[j] – x) is minimum.

Example:

```
Input:  ar1[] = {1, 4, 5, 7};
        ar2[] = {10, 20, 30, 40};
        x = 32
Output:  1 and 30

Input:  ar1[] = {1, 4, 5, 7};
        ar2[] = {10, 20, 30, 40};
        x = 50
Output:  7 and 40
```

We strongly recommend to minimize your browser and try this yourself first.

A **Simple Solution** is to run two loops. The outer loop considers every element of first array and inner loop checks for the pair in second array. We keep track of minimum difference between ar1[i] + ar2[j] and x.

We can do it **in O(n) time** using following steps.
1) Merge given two arrays into an auxiliary array of size m+n using merge process of merge sort. While merging keep another boolean array of size m+n to indicate whether the current element in merged array is from ar1[] or ar2[].

2) Consider the merged array and use the linear time algorithm to find the pair with sum closest to x. One extra thing we need to consider only those pairs which have one element from ar1[] and other from ar2[], we use the boolean array for this purpose.

Can we do it in a single pass and O(1) extra space?
The idea is to start from left side of one array and right side of another array, and use the algorithm same as step 2 of above approach. Following is detailed algorithm.

```
1) Initialize a variable diff as infinite (Diff is used to store the
   difference between pair and x).  We need to find the minimum diff.
2) Initialize two index variables l and r in the given sorted array.
       (a) Initialize first to the leftmost index in ar1:  l = 0
       (b) Initialize second  the rightmost index in ar2:  r = n-1
3) Loop while l = 0
       (a) If  abs(ar1[l] + ar2[r] - sum)
Following is C++ implementation of this approach.

// C++ program to find the pair from two sorted arays such
// that the sum of pair is closest to a given number x
#include <iostream>
#include <climits>
#include <cstdlib>
using namespace std;

// ar1[0..m-1] and ar2[0..n-1] are two given sorted arrays
// and x is given number. This function prints the pair  from
// both arrays such that the sum of the pair is closest to x.
void printClosest(int ar1[], int ar2[], int m, int n, int x)
{
    // Initialize the diff between pair sum and x.
    int diff = INT_MAX;

    // res_l and res_r are result indexes from ar1[] and ar2[]
    // respectively
    int res_l, res_r;

    // Start from left side of ar1[] and right side of ar2[]
    int l = 0, r = n-1;
    while (l<m && r>=0)
    {
       // If this pair is closer to x than the previously
       // found closest, then update res_l, res_r and diff
       if (abs(ar1[l] + ar2[r] - x) < diff)
       {
           res_l = l;
           res_r = r;
           diff = abs(ar1[l] + ar2[r] - x);
       }

       // If sum of this pair is more than x, move to smaller
       // side
       if (ar1[l] + ar2[r] > x)
           r--;
       else  // move to the greater side
           l++;
    }

    // Print the result
    cout << "The closest pair is [" << ar1[res_l] << ", "
         << ar2[res_r] << "] \n";
}
```

```
// Driver program to test above functions
int main()
{
    int ar1[] = {1, 4, 5, 7};
    int ar2[] = {10, 20, 30, 40};
    int m = sizeof(ar1)/sizeof(ar1[0]);
    int n = sizeof(ar2)/sizeof(ar2[0]);
    int x = 38;
    printClosest(ar1, ar2, m, n, x);
    return 0;
}
```

Output:

```
The closest pair is [7, 30]
```

This article is contributed by Harsh. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/given-two-sorted-arrays-number-x-find-pair-whose-sum-closest-x/

Category: Arrays

Post navigation

← Adobe Interview | Set 17 (For MTS-1) Multiply two polynomials →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 25

Find common elements in three sorted arrays

Given three arrays sorted in non-decreasing order, print all common elements in these arrays.

Examples:

```
ar1[] = {1, 5, 10, 20, 40, 80}
ar2[] = {6, 7, 20, 80, 100}
ar3[] = {3, 4, 15, 20, 30, 70, 80, 120}
Output: 20, 80

ar1[] = {1, 5, 5}
ar2[] = {3, 4, 5, 5, 10}
ar3[] = {5, 5, 10, 20}
Outptu: 5, 5
```

A simple solution is to first find intersection of two arraysand store the intersection in a temporary array, then find the intersection of third array and temporary array. Time complexity of this solution is O(n1 + n2 + n3) where n1, n2 and n3 are sizes of ar1[], ar2[] and ar3[] respectively.

The above solution requires extra space and two loops, we can find the common elements using a single loop and without extra space. The idea is similar to intersection of two arrays. Like two arrays loop, we run a loop and traverse three arrays.
Let the current element traversed in ar1[] be x, in ar2[] be y and in ar3[] be z. We can have following cases inside the loop.
1) If x, y and z are same, we can simply print any of them as common element and move ahead in all three arrays.
2) Else If x y and y > z), we can simply move ahead in ar3[] as z cannot be a common element.

Following are implementations of the above idea.

Chapter 26

C++

```
// C++ program to print common elements in three arrays
#include <iostream>
using namespace std;

// This function prints common elements in ar1
int findCommon(int ar1[], int ar2[], int ar3[], int n1, int n2, int n3)
{
    // Initialize starting indexes for ar1[], ar2[] and ar3[]
    int i = 0, j = 0, k = 0;

    // Iterate through three arrays while all arrays have elements
    while (i < n1 && j < n2 && k < n3)
    {
         // If x = y and y = z, print any of them and move ahead
         // in all arrays
         if (ar1[i] == ar2[j] && ar2[j] == ar3[k])
         {   cout << ar1[i] << " ";   i++; j++; k++; }

         // x < y
         else if (ar1[i] < ar2[j])
             i++;

         // y < z
         else if (ar2[j] < ar3[k])
             j++;

         // We reach here when x > y and z < y, i.e., z is smallest
         else
             k++;
    }
}

// Driver program to test above function
int main()
{
    int ar1[] = {1, 5, 10, 20, 40, 80};
    int ar2[] = {6, 7, 20, 80, 100};
```

```
    int ar3[] = {3, 4, 15, 20, 30, 70, 80, 120};
    int n1 = sizeof(ar1)/sizeof(ar1[0]);
    int n2 = sizeof(ar2)/sizeof(ar2[0]);
    int n3 = sizeof(ar3)/sizeof(ar3[0]);

    cout << "Common Elements are ";
    findCommon(ar1, ar2, ar3, n1, n2, n3);
    return 0;
}
```

Chapter 27

Python

```
# Python function to print common elements in three sorted arrays
def findCommon(ar1, ar2, ar3, n1, n2, n3):

    # Initialize starting indexes for ar1[], ar2[] and ar3[]
    i, j, k = 0, 0, 0

    # Iterate through three arrays while all arrays have elements
    while (i < n1 and j < n2 and k< n3):

        # If x = y and y = z, print any of them and move ahead
        # in all arrays
        if (ar1[i] == ar2[j] and ar2[j] == ar3[k]):
            print ar1[i],
            i += 1
            j += 1
            k += 1

        # x < y
        elif ar1[i] < ar2[j]:
            i += 1

        # y < z
        elif ar2[j] < ar3[k]:
            j += 1

        # We reach here when x > y and z < y, i.e., z is smallest
        else:
            k += 1

#Driver program to check above function
ar1 = [1, 5, 10, 20, 40, 80]
ar2 = [6, 7, 20, 80, 100]
ar3 = [3, 4, 15, 20, 30, 70, 80, 120]
n1 = len(ar1)
n2 = len(ar2)
n3 = len(ar3)
print "Common elements are",
```

```
findCommon(ar1, ar2, ar3, n1, n2, n3)

# This code is contributed by __Devesh Agrawal__
```

Output:

```
 Common Elements are 20 80
```

Time complexity of the above solution is O(n1 + n2 + n3). In worst case, the largest sized array may have all small elements and middle sized array has all middle elements.

This article is compiled by **Rahul Gupta** Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/find-common-elements-three-sorted-arrays/

Chapter 28

Find the Minimum length Unsorted Subarray, sorting which makes the complete array sorted

Given an unsorted array arr[0..n-1] of size n, find the minimum length subarray arr[s..e] such that sorting this subarray makes the whole array sorted.

Examples:
1) If the input array is [10, 12, 20, 30, 25, 40, 32, 31, 35, 50, 60], your program should be able to find that the subarray lies between the indexes 3 and 8.

2) If the input array is [0, 1, 15, 25, 6, 7, 30, 40, 50], your program should be able to find that the subarray lies between the indexes 2 and 5.

Solution:
1) Find the candidate unsorted subarray
a) Scan from left to right and find the first element which is greater than the next element. Let *s* be the index of such an element. In the above example 1, *s* is 3 (index of 30).
b) Scan from right to left and find the first element (first in right to left order) which is smaller than the next element (next in right to left order). Let *e* be the index of such an element. In the above example 1, e is 7 (index of 31).

2) Check whether sorting the candidate unsorted subarray makes the complete array sorted or not. If not, then include more elements in the subarray.
a) Find the minimum and maximum values in *arr[s..e]*. Let minimum and maximum values be *min* and *max*. *min* and *max* for [30, 25, 40, 32, 31] are 25 and 40 respectively.
b) Find the first element (if there is any) in *arr[0..s-1]* which is greater than *min*, change *s* to index of this element. There is no such element in above example 1.
c) Find the last element (if there is any) in *arr[e+1..n-1]* which is smaller than max, change *e* to index of this element. In the above example 1, e is changed to 8 (index of 35)

3) Print *s* and *e*.

Implementation:

```
#include<stdio.h>

void printUnsorted(int arr[], int n)
{
  int s = 0, e = n-1, i, max, min;

  // step 1(a) of above algo
  for (s = 0; s < n-1; s++)
  {
    if (arr[s] > arr[s+1])
      break;
  }
  if (s == n-1)
  {
    printf("The complete array is sorted");
    return;
  }

  // step 1(b) of above algo
  for(e = n - 1; e > 0; e--)
  {
    if(arr[e] < arr[e-1])
      break;
  }

  // step 2(a) of above algo
  max = arr[s]; min = arr[s];
  for(i = s + 1; i <= e; i++)
  {
    if(arr[i] > max)
      max = arr[i];
    if(arr[i] < min)
      min = arr[i];
  }

  // step 2(b) of above algo
  for( i = 0; i < s; i++)
  {
    if(arr[i] > min)
    {
      s = i;
      break;
    }
  }

  // step 2(c) of above algo
  for( i = n -1; i >= e+1; i--)
  {
    if(arr[i] < max)
    {
      e = i;
      break;
    }
  }
```

```
  // step 3 of above algo
  printf(" The unsorted subarray which makes the given array "
         " sorted lies between the indees %d and %d", s, e);
  return;
}

int main()
{
  int arr[] = {10, 12, 20, 30, 25, 40, 32, 31, 35, 50, 60};
  int arr_size = sizeof(arr)/sizeof(arr[0]);
  printUnsorted(arr, arr_size);
  getchar();
  return 0;
}
```

Time Complexity: O(n)

Please write comments if you find the above code/algorithm incorrect, or find better ways to solve the same problem.

Source

http://www.geeksforgeeks.org/minimum-length-unsorted-subarray-sorting-which-makes-the-complete-array-sorted/

Chapter 29

K'th Smallest/Largest Element in Unsorted Array | Set 2 (Expected Linear Time)

We recommend to read following post as a prerequisite of this post.

K'th Smallest/Largest Element in Unsorted Array | Set 1

Given an array and a number k where k is smaller than size of array, we need to find the k'th smallest element in the given array. It is given that ll array elements are distinct.

Examples:

```
 Input: arr[] = {7, 10, 4, 3, 20, 15}
       k = 3
Output: 7

Input: arr[] = {7, 10, 4, 3, 20, 15}
       k = 4
Output: 10
```

We have discussed three different solutions here.

In this post method 4 is discussed which is mainly an extension of method 3 (QuickSelect) discussed in the previouspost. The idea is to randomly pick a pivot element. To implement randomized partition, we use a random function, rand() to generate index between l and r, swap the element at randomly generated index with the last element, and finally call the standard partition process which uses last element as pivot.

Following is C++ implementation of above Randomized QuickSelect.

```
// C++ implementation of randomized quickSelect
#include<iostream>
#include<climits>
#include<cstdlib>
using namespace std;

int randomPartition(int arr[], int l, int r);
```

```
// This function returns k'th smallest element in arr[l..r] using
// QuickSort based method.  ASSUMPTION: ALL ELEMENTS IN ARR[] ARE DISTINCT
int kthSmallest(int arr[], int l, int r, int k)
{
    // If k is smaller than number of elements in array
    if (k > 0 && k <= r - l + 1)
    {
        // Partition the array around a random element and
        // get position of pivot element in sorted array
        int pos = randomPartition(arr, l, r);

        // If position is same as k
        if (pos-l == k-1)
            return arr[pos];
        if (pos-l > k-1)  // If position is more, recur for left subarray
            return kthSmallest(arr, l, pos-1, k);

        // Else recur for right subarray
        return kthSmallest(arr, pos+1, r, k-pos+l-1);
    }

    // If k is more than number of elements in array
    return INT_MAX;
}

void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Standard partition process of QuickSort().  It considers the last
// element as pivot and moves all smaller element to left of it and
// greater elements to right. This function is used by randomPartition()
int partition(int arr[], int l, int r)
{
    int x = arr[r], i = l;
    for (int j = l; j <= r - 1; j++)
    {
        if (arr[j] <= x)
        {
            swap(&arr[i], &arr[j]);
            i++;
        }
    }
    swap(&arr[i], &arr[r]);
    return i;
}

// Picks a random pivot element between l and r and partitions
// arr[l..r] arount the randomly picked element using partition()
int randomPartition(int arr[], int l, int r)
```

```
{
    int n = r-l+1;
    int pivot = rand() % n;
    swap(&arr[l + pivot], &arr[r]);
    return partition(arr, l, r);
}

// Driver program to test above methods
int main()
{
    int arr[] = {12, 3, 5, 7, 4, 19, 26};
    int n = sizeof(arr)/sizeof(arr[0]), k = 3;
    cout << "K'th smallest element is " << kthSmallest(arr, 0, n-1, k);
    return 0;
}
```

Output:

```
K'th smallest element is 5
```

Time Complexity:
The worst case time complexity of the above solution is still $O(n^2)$. In worst case, the randomized function may always pick a corner element. The expected time complexity of above randomized QuickSelect is $\Theta(n)$, see CLRS book or MIT video lecture for proof. The assumption in the analysis is, random number generator is equally likely to generate any number in the input range.

Sources:

MIT Video Lecture on Order Statistics, Median

Introduction to Algorithms by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L.

This article is contributed by **Shivam**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/kth-smallestlargest-element-unsorted-array-set-2-expected-linear-time/

Category: Arrays

Post navigation

← Amazon Interview Experience | Set 151 (For SDE) Amazon Interview Experience | Set 152 →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 30

K'th Smallest/Largest Element in Unsorted Array | Set 2 (Expected Linear Time)

We recommend to read following post as a prerequisite of this post.

K'th Smallest/Largest Element in Unsorted Array | Set 1

Given an array and a number k where k is smaller than size of array, we need to find the k'th smallest element in the given array. It is given that ll array elements are distinct.

Examples:

```
 Input: arr[] = {7, 10, 4, 3, 20, 15}
       k = 3
Output: 7

Input: arr[] = {7, 10, 4, 3, 20, 15}
       k = 4
Output: 10
```

We have discussed three different solutions here.

In this post method 4 is discussed which is mainly an extension of method 3 (QuickSelect) discussed in the previouspost. The idea is to randomly pick a pivot element. To implement randomized partition, we use a random function, rand() to generate index between l and r, swap the element at randomly generated index with the last element, and finally call the standard partition process which uses last element as pivot.

Following is C++ implementation of above Randomized QuickSelect.

```
// C++ implementation of randomized quickSelect
#include<iostream>
#include<climits>
#include<cstdlib>
using namespace std;

int randomPartition(int arr[], int l, int r);
```

```
// This function returns k'th smallest element in arr[l..r] using
// QuickSort based method.  ASSUMPTION: ALL ELEMENTS IN ARR[] ARE DISTINCT
int kthSmallest(int arr[], int l, int r, int k)
{
    // If k is smaller than number of elements in array
    if (k > 0 && k <= r - l + 1)
    {
        // Partition the array around a random element and
        // get position of pivot element in sorted array
        int pos = randomPartition(arr, l, r);

        // If position is same as k
        if (pos-l == k-1)
            return arr[pos];
        if (pos-l > k-1)  // If position is more, recur for left subarray
            return kthSmallest(arr, l, pos-1, k);

        // Else recur for right subarray
        return kthSmallest(arr, pos+1, r, k-pos+l-1);
    }

    // If k is more than number of elements in array
    return INT_MAX;
}

void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Standard partition process of QuickSort().  It considers the last
// element as pivot and moves all smaller element to left of it and
// greater elements to right. This function is used by randomPartition()
int partition(int arr[], int l, int r)
{
    int x = arr[r], i = l;
    for (int j = l; j <= r - 1; j++)
    {
        if (arr[j] <= x)
        {
            swap(&arr[i], &arr[j]);
            i++;
        }
    }
    swap(&arr[i], &arr[r]);
    return i;
}

// Picks a random pivot element between l and r and partitions
// arr[l..r] arount the randomly picked element using partition()
int randomPartition(int arr[], int l, int r)
```

```
{
    int n = r-l+1;
    int pivot = rand() % n;
    swap(&arr[l + pivot], &arr[r]);
    return partition(arr, l, r);
}

// Driver program to test above methods
int main()
{
    int arr[] = {12, 3, 5, 7, 4, 19, 26};
    int n = sizeof(arr)/sizeof(arr[0]), k = 3;
    cout << "K'th smallest element is " << kthSmallest(arr, 0, n-1, k);
    return 0;
}
```

Output:

```
K'th smallest element is 5
```

Time Complexity:
The worst case time complexity of the above solution is still $O(n^2)$. In worst case, the randomized function may always pick a corner element. The expected time complexity of above randomized QuickSelect is $\Theta(n)$, see CLRS book or MIT video lecture for proof. The assumption in the analysis is, random number generator is equally likely to generate any number in the input range.

Sources:

MIT Video Lecture on Order Statistics, Median

Introduction to Algorithms by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L.

This article is contributed by **Shivam**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/kth-smallestlargest-element-unsorted-array-set-2-expected-linear-time/

Category: Arrays

Post navigation

← Amazon Interview Experience | Set 151 (For SDE) Amazon Interview Experience | Set 152 →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 31

K'th Smallest/Largest Element in Unsorted Array | Set 3 (Worst Case Linear Time)

We recommend to read following posts as a prerequisite of this post.

K'th Smallest/Largest Element in Unsorted Array | Set 1

K'th Smallest/Largest Element in Unsorted Array | Set 2 (Expected Linear Time)

Given an array and a number k where k is smaller than size of array, we need to find the k'th smallest element in the given array. It is given that ll array elements are distinct.

Examples:

```
 Input: arr[] = {7, 10, 4, 3, 20, 15}
       k = 3
Output: 7

Input: arr[] = {7, 10, 4, 3, 20, 15}
       k = 4
Output: 10
```

In previous post, we discussed an expected linear time algorithm. In this post, a worst case linear time method is discussed. *The idea in this new method is similar to quickSelect(), we get worst case linear time by selecting a pivot that divides array in a balanced way (there are not very few elements on one side and many on other side).* After the array is divided in a balanced way, we apply the same steps as used in quickSelect() to decide whether to go left or right of pivot.

Following is complete algorithm.

```
 kthSmallest(arr[0..n-1], k)
1) Divide arr[] into  n/5rceil; groups where size of each group is 5
   except possibly the last group which may have less  than 5 elements.

2) Sort the above created  n/5  groups and find median
   of all groups. Create an auxiliary array 'median[]' and store medians
```

```
   of all  n/5  groups in this median array.

// Recursively call this method to find median of median[0.. n/5 -1]
3) medOfMed = kthSmallest(median[0.. n/5 -1],  n/10 )

4) Partition arr[] around medOfMed and obtain its position.
     pos = partition(arr, n, medOfMed)

5) If pos == k return medOfMed
6) If pos < k return kthSmallest(arr[l..pos-1], k)
7) If poa > k return kthSmallest(arr[pos+1..r], k-pos+l-1)
```

In above algorithm, last 3 steps are same as algorithm in previous post. The first four steps are used to obtain a good point for partitioning the array (to make sure that there are not too many elements either side of pivot).

Following is C++ implementation of above algorithm.

```
// C++ implementation of worst case linear time algorithm
// to find k'th smallest element
#include<iostream>
#include<algorithm>
#include<climits>
using namespace std;

int partition(int arr[], int l, int r, int k);

// A simple function to find median of arr[].  This is called
// only for an array of size 5 in this program.
int findMedian(int arr[], int n)
{
    sort(arr, arr+n);  // Sort the array
    return arr[n/2];   // Return middle element
}

// Returns k'th smallest element in arr[l..r] in worst case
// linear time. ASSUMPTION: ALL ELEMENTS IN ARR[] ARE DISTINCT
int kthSmallest(int arr[], int l, int r, int k)
{
    // If k is smaller than number of elements in array
    if (k > 0 && k <= r - l + 1)
    {
        int n = r-l+1; // Number of elements in arr[l..r]

        // Divide arr[] in groups of size 5, calculate median
        // of every group and store it in median[] array.
        int i, median[(n+4)/5]; // There will be floor((n+4)/5) groups;
        for (i=0; i<n/5; i++)
            median[i] = findMedian(arr+l+i*5, 5);
        if (i*5 < n) //For last group with less than 5 elements
        {
            median[i] = findMedian(arr+l+i*5, n%5);
            i++;
```

```
        }

        // Find median of all medians using recursive call.
        // If median[] has only one element, then no need
        // of recursive call
        int medOfMed = (i == 1)? median[i-1]:
                                 kthSmallest(median, 0, i-1, i/2);

        // Partition the array around a random element and
        // get position of pivot element in sorted array
        int pos = partition(arr, l, r, medOfMed);

        // If position is same as k
        if (pos-l == k-1)
            return arr[pos];
        if (pos-l > k-1)  // If position is more, recur for left
            return kthSmallest(arr, l, pos-1, k);

        // Else recur for right subarray
        return kthSmallest(arr, pos+1, r, k-pos+l-1);
    }

    // If k is more than number of elements in array
    return INT_MAX;
}

void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

// It searches for x in arr[l..r], and partitions the array
// around x.
int partition(int arr[], int l, int r, int x)
{
    // Search for x in arr[l..r] and move it to end
    int i;
    for (i=l; i<r; i++)
        if (arr[i] == x)
           break;
    swap(&arr[i], &arr[r]);

    // Standard partition algorithm
    i = l;
    for (int j = l; j <= r - 1; j++)
    {
        if (arr[j] <= x)
        {
            swap(&arr[i], &arr[j]);
            i++;
        }
    }
```

```
    swap(&arr[i], &arr[r]);
    return i;
}

// Driver program to test above methods
int main()
{
    int arr[] = {12, 3, 5, 7, 4, 19, 26};
    int n = sizeof(arr)/sizeof(arr[0]), k = 3;
    cout << "K'th smallest element is "
         << kthSmallest(arr, 0, n-1, k);
    return 0;
}
```

Output:

```
K'th smallest element is 5
```

Time Complexity:
The worst case time complexity of the above algorithm is O(n). Let us analyze all steps.

The steps 1) and 2) take O(n) time as finding median of an array of size 5 takes O(1) time and there are n/5 arrays of size 5.
The step 3) takes T(n/5) time. The step 4 is standard partition and takes O(n) time.
The interesting steps are 6) and 7). At most, one of them is executed. These are recursive steps. What is the worst case size of these recursive calls. The answer is maximum number of elements greater than medOfMed (obtained in step 3) or maximum number of elements smaller than medOfMed.
How many elements are greater than medOfMed and how many are smaller?
At least half of the medians found in step 2 are greater than or equal to medOfMed. Thus, at least half of the n/5 groups contribute 3 elements that are greater than medOfMed, except for the one group that has fewer than 5 elements. Therefore, the number of elements greater than medOfMed is at least.

$$3\left(\left\lceil\frac{1}{2}\left\lceil\frac{n}{5}\right\rceil\right\rceil-2\right) \geq \frac{3n}{10}-6$$

Similarly, the number of elements that are less than medOfMed is at least 3n/10 – 6. In the worst case, the function recurs for at most n – (3n/10 – 6) which is 7n/10 + 6 elements.

Note that 7n/10 + 6 20 and that any input of 80 or fewer elements requires O(1) time. We can therefore obtain the recurrence

$$T(n) \leq \begin{cases} \Theta(1) & \text{if } n \leq 80\,, \\ T(\lceil n/5\rceil) + T(7n/10+6) + O(n) & \text{if } n > 80\,. \end{cases}$$

We show that the running time is linear by substitution. Assume that T(n) cn for some constant c and all n > 80. Substituting this inductive hypothesis into the right-hand side of the recurrence yields

```
T(n)
since we can pick c large enough so that c(n/10 - 7) is larger than the function described by the O(n) term for
Note that the above algorithm is linear in worst case, but the constants are very high for this algorithm. The
Sources:
```

MIT Video Lecture on Order Statistics, Median

Introduction to Algorithms by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L.

http://staff.ustc.edu.cn/~csli/graduate/algorithms/book6/chap10.htm
This article is contributed by Shivam. Please write comments if you find anything incorrect, or you want to sh

Source

http://www.geeksforgeeks.org/kth-smallestlargest-element-unsorted-array-set-3-worst-case-linear-time/

Category: Arrays

Post navigation

← Check a given sentence for a given set of simple grammer rules [TopTalent.in] Interview With Faraz Who Got Into MobiKwik →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.