MINOR PROJECT

Analysis and Design of Bitwise Sort

By:

Vivek Kumar

(17115091)

Project Guide:

Dr. Pradeep Singh

(Assistant Prof.)

NIT Raipur

Outline

- Introduction.
- Design and Implementation.
- Time Complexity Analysis.
- Space Complexity Analysis.
- Possible Data Types.
- Limitations.
- References.

Introduction

Sorting is an important operation in computer programming. For any sequence of records or data, sort is an ordering procedure by a type of keyword. The sorted sequence is benefit for record searching, insertion and deletion. Thus enhance the efficiency of these operations.

A Sorting Algorithm is used to rearrange a given array or list elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of element in the respective data structure.

5 Sorting Algorithms

- Bubble Sort.
- Selection Sort.
- Insertion Sort.
- Quick Sort.
- Merge Sort.

A. Bubble Sort

```
void bubbleSort(int arr[], int n)
{   int i, j;
    for (i = 0; i < n-1; i++)
    for (j = 0; j < n-i-1; j++)
        if (arr[j] > arr[j+1])
            swap(&arr[j], &arr[j+1]);
}
```

Worst and Average Case Time Complexity: $O(n^2)$. Worst case occurs when array is reverse sorted.

Best Case Time Complexity: O(n). Best case occurs when array is already sorted.

Auxiliary Space: O(1)

B. Selection Sort

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

Time Complexity: $O(n^2)$ as there are two nested loops.

Auxiliary Space: O(1)

The good thing about selection sort is it never makes more than O(n) swaps and can be useful when memory write is a costly operation.

```
void selectionSort(int arr[], int n)

{   int i, j, min_idx;

    for (i = 0; i < n-1; i++)

    {    min_idx = i;

        for (j = i+1; j < n; j++)

        if (arr[j] < arr[min_idx])  min_idx = j;

        swap(&arr[min_idx], &arr[i]);

} }
```

C. Insertion Sort

Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

Time Complexity: O(n*2)

Auxiliary Space: O(1)

```
void insertionSort(int arr[], int n)
{  int i, key, j;
   for (i = 1; i < n; i++)
   {
      key = arr[i]; j = i - 1;
      while (j >= 0 && arr[j] > key)
      {
         arr[j + 1] = arr[j];  j = j - 1;   }
      arr[j + 1] = key;
   } }
```

D. Merge Sort

Merge Sort is a Divide and Conquer algorithm. It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves.

MergeSort(arr[], l, r)

If r $>$ l

1. Find the middle point to divide the array into two halves:

middle m $=$ (l$+$r)/2

2. Call mergeSort for first half: Call mergeSort(arr, l, m)

3. Call mergeSort for second half: Call mergeSort(arr, m$+$1, r)

4. Merge the two halves sorted in step 2 and 3: Call merge(arr, l, m, r)

Time Complexity: Sorting arrays on different machines. Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.

$T(n) = 2T(n/2) + \theta(n)$

Auxiliary Space: O(n)

Algorithmic Paradigm: Divide and Conquer

Sorting In Place: No in a typical implementation

Stable: Yes

E. Quick Sort

Like MergeSort, QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

- Always pick first element as pivot.
- Always pick last element as pivot (implemented below)
- Pick a random element as pivot.
- Pick median as pivot.

Time taken by QuickSort in general can be written as following.

$$T(n) = T(k) + T(n\text{-}k\text{-}1) + (n).$$

Bitwise Sort

- Bitwise sort divides the array in two disjoint array, one having bth bit zero we call this array ‘left array’ and another having bth bit one we call this ‘right array’.
- Where b starts from msb and goes down till 0.
- Same operation will be performed on both left and right array separately using b-1th bit.

Visual presentation

- https://repl.it/@VivekKumar4/bitWiseSortVisual#main.py
- Click on above link.
- And run the python script there.
- And see how an array of integer getting sorted by bitwise sort algorithm.

Design and Implementation

bitwiseSort(arr[],i,j,b):

#i=starting index, j=end index, b=bit position, arr=array of integers

Step0: if i>=j or b=0 then goto step8.

Step1: set mid:=i-1, ind:=i.

Step2: if b & arr[ind] == 0 then goto step 3 else goto step4.

Step3: mid++; swap(arr[mid],arr[ind]).

Step4: ind++.

Step5: if ind<=j then go to step2 else goto step6.

Step6: bitwiseSort(arr,i,mid,b>>1).

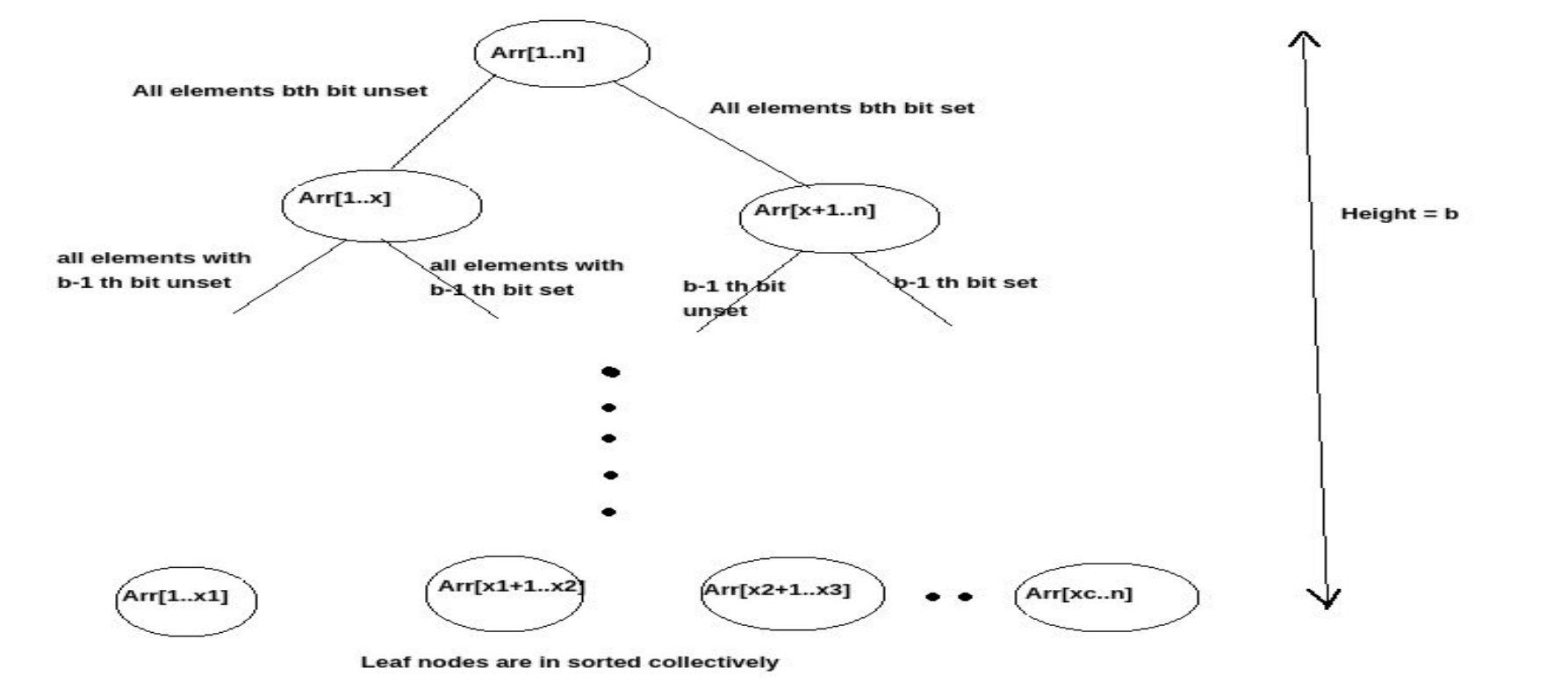
Step7: bitwiseSort(arr,mid+1,j,b>>1).

Step8: exit.

Program Code

```
1   void bitwise_sort(int a[],int l,int r,int b)
2   {
3       if(b == 0 || l >= r) return;
4       int i=l-1;
5       for(int j=l,j<r+1,j++) {
6           if((a[j] & b) == 0) {
7               swap(a[++i],a[j]);
8           }
9       }
10      bitwise_sort(a,l,i,b>>1);
11      bitwise_sort(a,i+1,r,b>>1);
12  }
13
```

Time Complexity Analysis



Time Complexity Analysis | Recurrence Relation

- $T(N,B)=T(X,B-1)+T(Y,B-1)+O(N)$.
- $X+Y=N$.
- $T(1,K)=O(1)$.
- $T(Z,0)=O(1)$.
- Height of the tree is size of datatype.
- Basically we are visiting each bit exactly once of the array.
- So, if we have n elements in the array and each element is represented by b bits then there is n*b bits in total.
- Hence in total n*b operations are being performed .
- And b is constant for any specific data type.
- Hence time complexity is O(n) in all cases.

Space Complexity Analysis

At max b stack frame gonna be used to sort.

Where b is the number of bits to represent a number, which is a constant.

So, overall O(1) space used.

Example

Arr = [10,9,13,4,15,1]

10	9	13	4	15	1
1	**1**	**1**	**0**	**1**	**0**
0	0	1	1	1	0
1	0	0	0	1	0
0	1	1	0	1	1

1st row has numbers to be sorted and their binary in corresponding column

4	**1**	13	10	15	9
0	**0**	1	1	1	1
1	**0**	**1**	**0**	**1**	**0**
0	**0**	0	1	1	0
0	**1**	1	0	1	1

After first iteration and in 2nd iteration 3rd row bit being used

1	4	10	9	15	13
0	0	1	1	1	1
0	1	0	0	1	1
0	0	1	0	1	0
1	0	0	1	1	1

1	4	9	10	13	15
0	0	1	1	1	1
0	1	0	0	1	1
0	0	0	1	0	1
1	0	1	0	1	1

1	4	9	10	13	15
0	0	1	1	1	1
0	1	0	0	1	1
0	0	0	1	0	1
1	0	1	0	1	1

Data Types that can be sorted using Bitwise Sort

- Integers.
- Chars.
- This Algorithm can be extended to sort strings as well.
- First we will apply for first character : will result set of group having same first character
- In next turn apply for second character with each group individually
- .. so on till max length of character.
- If we talk about time complexity again we are visiting all the bits exactly once.
- If we have n strings each of length m then total number of bits will be n*m*8.
- That means total n*m*8 operations are being performed.
- Hence time complexity : O(n*m) in all cases.

Implementation for array of string sorting | C++

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

void bsort_util(string arr[],int l,int r,int ind,int b){
    if(l>=r) return;
    if(b==0){
        int i = l-1;
        for(int j=l;j<=r;j++){
            if(arr[j].length()==ind+1){
                i++;
                swap(arr[i],arr[j]);
            }
        }
        bsort_util(arr,i+1,r,ind+1,1<<8);
    }
```

```
    }
    else{
        int i = l-1;
        for(int j=l;j<=r;j++){
            if((arr[j][ind] & b) == 0){
                i++;
                swap(arr[i],arr[j]);
            }
        }
        bsort_util(arr,l,i,ind,b>>1);
        bsort_util(arr,i+1,r,ind,b>>1);
    }
}

void bsort(string arr[],int size){
    bsort_util(arr,0,size-1,0,1<<8);
}
```

Limitations

- Only comparison by value is possible.
- Custom comparison can’t be implemented.
- So, very complex to implement for custom data types.

References

- https://ieeexplore.ieee.org/document/5987184
- https://www.researchgate.net/publication/222812966_The_design_of_divide_and_conquer_algorithms
- https://academic.oup.com/comjnl/article/5/1/10/395338