Analysis of Algorithms

Sorting

Prof. Muhammad Saeed

Sorting Algorithms

1. Bubble Sort
2. Selection Sort
3. Insertion Sort
4. Shellsort
5. Mergesort
6. Quicksort
7. Bucket Sort
8. Heapsort
9. Radix Sort
10. Topological Sort

Sorting

Theorems

1) The average number of inversions in an array of n distinct numbers is n(n-1)/4.

2) Any algorithm that sorts by exchanging adjacent elements requires $\Omega(n^2)$ time on average.

3)

a) Any sorting algorithm that uses only comparisons between elements requires at least [logn!] comparisons in the worst case.

b) Any sorting algorithm that uses only comparisons between elements requires Ω(nlogn) comparisons.

Sorting

1. Bubble Sort

```
void bubbleSort(int numbers[], int n)
{                                   // n is array size
    int i, j, temp;
    for (i = (n - 1); i >= 0; i--)
    {
        for (j = 1; j <= i; j++)
        {
            if (numbers[j-1] > numbers[j])
            {
                temp = numbers[j-1];
                numbers[j-1] = numbers[j];
                numbers[j] = temp;
            }
        }
    }
}                                   //T(n) = O(n^2)
```

Analysis of Bubble Sort

$$\sum_{i=n-1}^{0} \sum_{j=1}^{i} 1 = \sum_{i=0}^{n-1} \sum_{j=1}^{i} 1$$

$$\sum_{i=0}^{n-1} \sum_{j=1}^{i} 1 = \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$$

$$T(n) = O(n^2)$$

Sorting

2. Selection Sort

```
void selectionSort(int numbers[], int n)
{                                   // n is array size
    int i, j, min, temp;
    for (i = 0; i < n-1; i++)
    {
        min = i;
        for (j = i+1; j < n; j++)
        {
            if (numbers[j] < numbers[min])
            min = j;
        }
        temp = numbers[i];
        numbers[i] = numbers[min];
        numbers[min] = temp;
    }
}
```

Analysis of Selection Sort

For each i from 1 to n-1, there is one exchange and n-i comparisons, so there is a total of n-1 exchanges and (n-1) + (n-2) + . . . + 2 + 1 = n(n-1)/2 comparisons.

$$\sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n-1)-(i+1)+1$$

$$= \sum_{i=0}^{n-2} (n-i-1) = \sum_{i=0}^{n-2} (n-i-1) = \sum_{i=0}^{n-2} n - \sum_{i=0}^{n-2} i - \sum_{i=0}^{n-2} 1$$

$$= n(n-1) - \frac{n(n-1)}{2} - n + 1 = \frac{n(n-1)}{2} - n + 1 = O(n^2)$$

In the worst case, this could be quadratic, $T(n) = O(n^2)$, but in the average case, this quantity is $O(n\log n)$. It implies that the running time of Selection sort is quite insensitive to the input.

3. Insertion Sort

```
void insertionSort(int numbers[], int n)
{                                          // n is array size
    int i, j, index;
    for (i=1; i < n; i++)
    {
        index = numbers[i];
        j = i;
        while ((j > 0) && (numbers[j-1] > index))
        {
            numbers[j] = numbers[j-1];
            j = j - 1;
        }
        numbers[j] = index;
    }
}
```

Analysis of Insertion Sort

Best-Case

The while-loop is executed only once for each *j*. This happens if given array *A* is already sorted. It is a linear function of *n*.

$$T(n) = an + b = O(n)$$

Worst-Case

The worst-case occurs, j is executed for i times for each value of i. This can happen if array A starts out in reverse order

$$\sum_{i=1}^{n-1} \sum_{j=i}^{1} 1 = \sum_{i=1}^{n-1} i$$

$$= \frac{n(n-1)}{2} = O(n^2)$$

It is a quadratic function of *n*. $T(n) = O(n^2)$

4. Shellsort

```
void shellSort(int numbers[], int n)
{
   int i, j, temp, increment =5;
   while (increment > 0)
      {
         for (i=0; i < n; i++)
            {
                  j = i;
               temp = numbers[i];
                        while ((j >= increment) && (numbers[j-increment] > temp))
                           {
                                 numbers[j] = numbers[j - increment];
                                 j = j - increment;
                           }
                  numbers[j] = temp;
               }
         if (increment/2 != 0)
            increment = increment/2;
         else if (increment == 1)    increment = 0;
               else     increment = 1;
      } }
```

‘Analysis’ of Shellsort

Shell Sort for increment sequence 5, 3, 1

Array	81	94	11	96	12	35	17	95	28	58	41	75	15
Pass 1	35	17	11	28	12	41	75	15	96	58	81	94	95
Pass 2	28	12	11	35	15	41	58	17	94	75	81	96	95
Pass 3	11	12	15	17	28	35	41	58	75	81	94	95	96

1) The worst-case running time of shellsort, using Shell’s increments, is $\Theta(n^2)$

2) The worst-case running time of shellsort, using Hibbard’s increments, is $\Theta(n^{3/2})$

5. Mergesort

```
void mergeSort(int numbers[], int temp[], int n)
{                                          // n is array size
    m_sort(numbers, temp, 0, n - 1);
}

void m_sort(int numbers[], int temp[], int left, int right)
{
    int mid;
    if (right > left)
    {
        mid = (right + left) / 2;
        m_sort(numbers, temp, left, mid);
        m_sort(numbers, temp, mid+1, right);
        merge(numbers, temp, left, mid+1, right);
    }
}
```

// .....Continued

…….Mergesort Continued

```
void merge(int numbers[], int temp[], int left, int mid, int right)
{
        int i, left_end, num_elements, tmp_pos;
        left_end = mid - 1;
        tmp_pos = left;
        num_elements = right - left + 1;
        while ((left <= left_end) && (mid <= right))
        {
            if (numbers[left] <= numbers[mid])
            {
                temp[tmp_pos] = numbers[left];
                tmp_pos = tmp_pos + 1;
                left = left +1;
            }
            else
            {
                temp[tmp_pos] = numbers[mid];
                tmp_pos = tmp_pos + 1;
                mid = mid + 1;
            }     }
```


// Continued

//……Mergesort Continued

```
        while (left <= left_end)
        {
                temp[tmp_pos] = numbers[left];
                left = left + 1;
                tmp_pos = tmp_pos + 1;
        }

        while (mid <= right)
        {
                temp[tmp_pos] = numbers[mid];
                mid = mid + 1;
                tmp_pos = tmp_pos + 1;
        }
        for (i=0; i <= num_elements; i++)
        {
                numbers[right] = temp[right];
                right = right - 1;
        }
}
```

Analysis of Mergesort I

We assume that Array size n is a power of 2 to split it in even halves. Sorting routine m_sort() is called recursively two times and merge( ) is linear.

$$T(1) = 1$$

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

$$\frac{T(n)}{n} = \frac{T(n/2)}{n/2} + 1$$

$$\frac{T(n/2)}{n/2} = \frac{T(n/4)}{n/4} + 1$$

$$\frac{T(n/4)}{n/4} = \frac{T(n/8)}{n/8} + 1$$

$$\ldots\ldots\ldots\ldots$$

$$\frac{T(2)}{2} = \frac{T(1)}{1} + 1$$

$$\textit{adding all the above equations :}$$

$$\frac{T(n)}{n} = \frac{T(1)}{1} + \log n$$

$$T(n) = n + n\log n = O(n\log n)$$

T(n)=O(nlogn)

Analysis of Mergesort II

We assume that Array size n is a power of 2 to split it in even halves. Sorting routine m_sort() is called recursively two times and merge( ) is linear.

$$T(1) = 1$$

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

$$2T(n/2) = 2(2(T(n/4)) + n/2)$$

$$T(n) = 4T(n/4) + 2n$$

$$4T(n/4) = 4(2(T(n/8)) + n/4)$$

$$T(n) = 8T(n/8) + 3n$$

....

$$T(n) = 2^k T(n/2^k) + k.n$$

$$\text{Using } k = \log\ n$$

$$T(n) = nT(1) + n\log n = n + n\log n$$

$$T(n) = O(n\log n)$$

T(n)=O(nlogn)

6. Quicksort

```
void QuickSort( int a[], int low, int high)
{
        int up, down, mid, pivot;
        if ( (high – low) <= 0)
                return;
        else
                if ((high – low) == 1)
                        if ( a[high] < a[low] )
                                Swap(a[low], a[high]);
                        return;
        mid = ( low + high )/2;
        pivot = a[mid];
        Swap( a[mid], a[low] );
        up = low + 1;
        down = high;
```

//Continued …….

//…… Quicksort Continued

```
        repeat
        {
                while ((up <= down) && (a[up] <= pivot ) )
                        up = up + 1;
                while( pivot < a[down] )
                        down = down – 1;
                if(up < down )
                        Swap ( a[up], a[down] );
        }until( up >= down);

        a[low] = a[down];
        a[down] = pivot;
        if( low < down -1)
                QuickSort(a[], low, down - 1);
        if( down +1 < high)
                QuickSort(a[], down + 1, high);
}
```

Analysis of Quicksort

$$T(0) = T(1) = 1$$

$$T(n) = T(i) + T(n-i-1) + cn$$

Mid value is smallest all the time, i = 0 and ignoring T(0)=1. i is the size of partition.

T(n) = O(n²)

Worst-Case:

$$T(n) = T(n-1) + cn,\ n > 1$$

Telescopin g repeatedly

$$T(n-1) = T(n-2) + c(n-1)$$

$$T(n-2) = T(n-3) + c(n-2)$$

....

$$T(2) = T(1) + c(2)$$

Adding all equations

$$T(n) = T(1) + c\sum_{i=2}^{n} i$$

$$T(n) = 1 + \frac{(n-1)n}{2} = O(n^2)$$

Analysis of Quicksort

$$T(0) = T(1) = 1$$

$$T(n) = T(i) + T(n-i-1) + cn$$

Mid value is in the middle
all the time, i = n/2.
i is the size of partition.

T(n)= O(nlogn)

Best-Case:

$$T(n) = 2T(\frac{n}{2}) + cn$$

$$\frac{T(n)}{n} = \frac{T(n/2)}{n/2} + c$$

$$\frac{T(n/2)}{n/2} = \frac{T(n/4)}{n/4} + c$$

$$\frac{T(n/4)}{n/4} = \frac{T(n/8)}{n/8} + c$$

……..

$$\frac{T(2)}{2} = \frac{T(1)}{1} + c$$

$$adding\ all\ the\ above\ equations:$$

$$\frac{T(n)}{n} = \frac{T(1)}{1} + c\log n$$

$$T(n) = n + cn\log n = O(n\log n)$$

Analysis of Quicksort

$$T(0) = T(1) = 1$$

$$T(n) = T(i) + T(n-i-1) + cn$$

Every partition is equally probable, the average value of T(i) and hence T(n-i-1),is $\frac{1}{n}\sum_{j=0}^{n-1} T(j)$

Average-Case:

$$T(n) = \frac{2}{n}\sum_{j=0}^{n-1} T(j) + cn \qquad \text{(i)}$$

$$nT(n) = 2\sum_{j=0}^{n-1} T(j) + cn^2 \qquad \text{(ii) \{multiply (i) by n\}}$$

$$(n-1)T(n-1) = 2\sum_{j=0}^{n-2} T(j) + c(n-1)^2 \qquad \text{(iii) \{For n = n - 1\}}$$

$$nT(n) - (n-1)T(n-1) = 2T(n-1) + 2cn - c \qquad \text{(iv) \{Subtract (iii) from (ii)\}}$$

$$nT(n) = (n+1)T(n-1) + 2cn \qquad \text{(v) \{rearrangi ng and neglecting c\}}$$

Continued …….

Analysis of Quicksort

Average-Case:

$$nT(n) = (n+1)T(n-1) + 2cn$$

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2c}{n+1}$$ (v) {Last Equation}

$$\frac{T(n-1)}{n} = \frac{T(n-2)}{n-1} + \frac{2c}{n}$$ (vi) {dividing by n(n+1)}

$$\frac{T(n-2)}{n-1} = \frac{T(n-3)}{n-2} + \frac{2c}{n-1}$$ (vii) {Telescoping}

....

$$\frac{T(2)}{3} = \frac{T(1)}{2} + \frac{2c}{3}$$

$$\frac{T(n)}{n+1} = \frac{T(1)}{2} + 2c\sum_{i=3}^{n+1}\frac{1}{i}$$

$$\frac{T(n)}{n+1} = \frac{1}{2} + \log(n+1) + \gamma - \frac{3}{2}, \gamma = 0.577$$

$$\frac{T(n)}{n+1} = O(\log n)$$

$$T(n) = O(n\log n)$$

T(n)=O(nlogn)

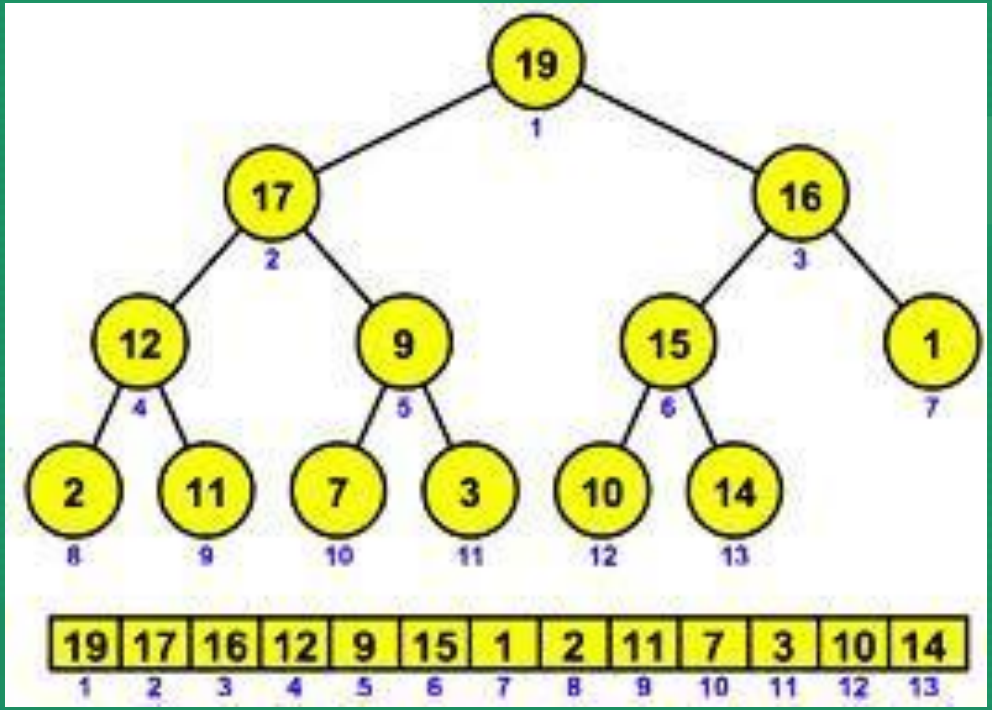
7. Bucket Sort

```
/*  To sort a sequence of 16 integers having 971 as maximum:
    19, 0, 45, 167, 570, 431, 638, 824, 752, 73, 9, 971, 431, 262, 9, 5
    Define an array of 972 integers ( 972 buckets )
*/
int i, j, buckets[972];
int sequence[16]={14, 0, 45, 167, 570, 431, 638, 824, 752, 73, 9, 971, 431,
262, 9, 5};
for( i =0; i<972; i++)
        buckets[i]=0;
for( i=0; i<16; i++)
        buckets[sequence[i]] += 1;
for( i=0; i<972; i++)                    //printing array
        for( j=1; j<= buckets[i]; j++)
                cout << i <<"  "<<endl;
```

1	0	0	0	0	1	0	0	0	2	0	1	0	…..	1
0	1	2	3	4	5	6	7	8	9	10	11	12	…..	971

8. Heapsort

```
void heapSort(int numbers[], int array_size)
{
    int i, temp;
    for (i = (array_size / 2)-1; i >= 0; i--)
        siftDown(numbers, i, array_size);
    for (i = array_size-1; i >= 1; i--)
    {
        temp = numbers[0];
        numbers[0] = numbers[i];
        numbers[i] = temp;
        siftDown(numbers, 0, i-1);
    }
}
```



// .....Continued

//……Heapsort Continued

```
void siftDown(int numbers[], int root, int bottom) {
    int done, maxChild, temp;
    done = 0;
    while ((root*2 <= bottom) && (!done)) {
        if (root*2 == bottom)
            maxChild = root * 2;
        else if (numbers[root * 2] > numbers[root * 2 + 1])
            maxChild = root * 2;
            else
                maxChild = root * 2 + 1;
        if (numbers[root] < numbers[maxChild]) {
            temp = numbers[root];
            numbers[root] = numbers[maxChild];
            numbers[maxChild] = temp;
            root = maxChild;
        }
        else
            done = 1;
    }
}
```

9. Radix Sort

64	8	216	512	27	729	0	1	343	125

Radix	0	1	2	3	4	5	6	7	8	9
Pass 1	0	1	512	343	64	125	216	27	8	729
Pass 2	0	512	125		343		64			
	1	216	27							
	8		729							
Pass 3	0	125	216	343		512		729		
	1									
	8									
	27									
	64									

10. Topological Sort

?

END of Sorting