Algorithms*

Manpreet Rai

November 10, 2023

1 Insertion Sort

Insertion Sort handles sorting in a very easy way, scanning from left to right, one element at a time. It picks an element, checks it to its left side, if found smaller to the left element it shifts the left elements by one towards right and places picked element on left of them until there is no element on left side greater than it.

```
int[] Sort(int[] sequence)
{
    for (int j = 1; j < sequence.Length; j++)
    {
        int key = sequence[j];
        int i = j - 1;
        while (i >= 0 && sequence[i] > key)
        {
            sequence[i + 1] = sequence[i];
            i--;
        }
        sequence[i + 1] = key;
    }

    return sequence;
}
```

2 Linear Search

Linear Search is just returning the index of an element if it is found among the array of elements, if not found Null is returned.

```
int? Search(int[] sequence, int searchValue)
{
    int? index = null;
    for (int i = 0; i < sequence.Length; i++)
    {
        if (sequence[i] == searchValue)
        {
            index = i;
            break;
        }
```

*Cormen T. H. et al., *Introduction to Algorithms*, Ed. 3rd.

```
    }

    return index;
}
```

3 N-Bit Addition

Two N-bit arrays are added and results are stored in N+1 bit array where left most digit shows the carry.

```
int[] A = new int[8];
int[] B = new int[8];
int[] C = new int[9];

public int[] Sum()
{
    int carry = 0;
    for (int i = A.Length - 1; i >= 0; i--)
    {
        C[i + 1] = (A[i] + B[i] + carry) % 2;
        if ((A[i] + B[i] + carry) >= 2)
            carry = 1;
        else
            carry = 0;
    }

    C[0] = carry;

    return C;
}
```

4 Selection Sort

Selection Sort is like Insertion Sort but in this, an element is picked from the pile starting from left side, and compared against the right sided elements to find any other element smallest than it, if found the element is swapped with it.

Note: Whenever we use swap like in Selection Sort or Bubble Sort, anything involving swapping, point to remember is we don't need to run loop till the last value, since with whom we want to swap the last element, it is already swapped at some point earlier, hence save yourself a loop condition and make sure to run the loop just about the Length - 1.

```
int[] Sort(int[] sequence)
{
    for (int i = 0; i < sequence.Length - 1; i++)
    {
        int min = i;

        for (int j = i + 1; j < sequence.Length; j++)
        {
            if (sequence[j] < sequence[min])
            {
                min = j;
            }
```

```
        }

        (sequence[min], sequence[i]) = (sequence[i], sequence[min]);
    }

    return sequence;
}
```

5 Bubble Sort

```
int[] Sort(int[] sequence)
{
    for (int i = 0; i < sequence.Length - 1; i++)
    {
        for (int j = sequence.Length - 1; j > i; j--)
        {
            if (sequence[j] < sequence[j - 1])
            {
                (sequence[j], sequence[j - 1]) = (sequence[j - 1], sequence[j]);
            }
        }
    }

    return sequence;
}
```

6 Merge Sort

Note: It is required to specify a condition for breaking recursion, to prevent endless trap. So in all recursion based algorithms like Merge Sort, Binary Search etc., specify a breaking condition or breakpoint for recursion at the starting.

```
void Merge(int[] a, int min, int mid, int max)
{
    int n1 = mid - min + 1;
    int n2 = max - mid;

    int[] left = new int[n1];
    int[] right = new int[n2];

    int i, j, k = min;
    for (i = 0; i < n1; i++) left[i] = a[min + i];
    for (j = 0; j < n2; j++) right[j] = a[mid + j + 1];

    i = j = 0;

    while (i < n1 && j < n2)
    {
        if (left[i] <= right[j])
        a[k++] = left[i++];

        else
        a[k++] = right[j++];
    }

```

```
    while (i < n1)
    {
        a[k++] = left[i++];
    }
    while (j < n2)
    {
        a[k++] = right[j++];
    }
}

public int[] Sort(int[] a, int min, int max)
{
    if (min < max) // Recursion break condition: min >= max
    {
        int mid = (min + max)/2;

        Sort(a, min, mid);
        Sort(a, mid + 1, max);

        Merge(a, min, mid, max);
    }

    return a;
}
```

7 Binary Search

```
// This BinarySearch operates on sorted arrays only
int? Search(int[] a, int key, int min, int max)
{
    if (min > max) return null;
    else
    {
        int mid = (min + max) / 2;

        if (a[mid] == key)
            return mid;
        else if (a[mid] < key)
            return Search(a, key, mid + 1, max);
        else
            return Search(a, key, min, mid - 1);
    }
}
```

8 Max Sub-Array

```
(int left, int right, int sum) FindMidCrossingArray(int[] a, int min, int mid, int max)
{
    int lSum = int.MinValue, rSum = int.MinValue, sum = 0, left = min, right = max;
    for (int i = mid; i >= min; i--)
    {
        sum += a[i];
        if (sum > lSum)
        {
            lSum = sum;
            left = i;
```

```
        }
    }
    sum = 0;
    for (int i = mid + 1; i <= max; i++)
    {
        sum += a[i];
        if (sum > rSum)
        {
            rSum = sum;
            right = i;
        }
    }

    return (left, right, lSum + rSum);
}

(int left, int right, int sum) FindMaxSubArray(int[] a, int min, int max)
{
    if (min == max) return (min, max, a[min]);

    int mid = (min + max) / 2;
    (int leftLow, int leftHigh, int leftSum) = FindMaxSubArray(a, min, mid);
    (int rightLow, int rightHigh, int rightSum) = FindMaxSubArray(a, mid + 1, max);
    (int crossLow, int crossHigh, int crossSum) = FindMidCrossingArray(a, min, mid, max);

    if (leftSum >= rightSum && leftSum >= crossSum)
        return (leftLow, leftHigh, leftSum);
    else if (rightSum >= leftSum && rightSum >= crossSum)
        return (rightLow, rightHigh, rightSum);
    else
        return (crossLow, crossHigh, crossSum);
}
```

9 Heap Sort

```
int Left(int i) => 2 * i + 1; // i = node at action
int Right(int i) => 2 * i + 2;
int _heapSize = 8;

void MaxHeapify(int[] a, int i)
{
    int l = Left(i);
    int r = Right(i);
    int largest;

    if (l < _heapSize && a[l] > a[i])
    largest = l;
    else
    largest = i;

    if (r < _heapSize && a[r] > a[largest])
        largest = r;

    if (largest != i)
    {
        (a[i], a[largest]) = (a[largest], a[i]);
        MaxHeapify(a, largest);
```

```
    }
}

void BuildMaxHeap(int[] a)
{
    for (int i = (_heapSize - 1) / 2; i >= 0; i--)
        MaxHeapify(a, i);
}

int[] Sort(int[] a)
{
    BuildMaxHeap(a);

    for (int i = (_heapSize - 1); i > 0; i--)
    {
        (a[0], a[i]) = (a[i], a[0]);
        _heapSize--;
        MaxHeapify(a, 0);
    }

    return a;
}
```

10 Quick Sort

```
int Partition(int[] a, int min, int max)
{
    int x = a[max];
    int i = min - 1;
    for (int j = min; j <= max - 1; j++)
    {
        if (a[j] <= x)
        {
            i++;
            (a[i], a[j]) = (a[j], a[i]);
        }
    }

    (a[i + 1], a[max]) = (a[max], a[i + 1]);
    return i + 1;
}

int[] Sort(int[] a, int min, int max)
{
    if (min < max)
    {
        int mid = Partition(a, min, max);
        Sort(a, min, mid - 1);
        Sort(a, mid + 1, max);
    }

    return a;
}
```

11 Count Sort

```
int Max(int[] a)
{
    int max = a[0];
    for (int i = 1; i < a.Length; i++)
    {
        if (a[i] > max)
        max = a[i];
    }

    return max;
}

int[] Sort(int[] a)
{
    int[] b = new int[a.Length];
    int[] c = new int[Max(a) + 1];

    // Count each element in a, store the count in c
    for (int i = 0; i < a.Length; i++)
        c[a[i]]++;

    // This provides position of elements in a, where to put in b
    for (int i = 1; i < c.Length; i++)
        c[i] += c[i - 1];

    for (int i = a.Length - 1; i >= 0; i--)
    {
        b[c[a[i]] - 1] = a[i];
        c[a[i]]--;
    }

    return b;
}
```