Algorithms

And data structures

Analysis of Algorithms

When we analyse an algorithm, we want to know

- Execution time
- Memory use

Typically, we want to know how space and time requirements increase with the size of the input. Ideally, we would like our algorithm's space requirements to be a constant factor of the input size and the execution time to be independent of the input size.

When we analyse the execution time of an algorithm, we can determine the total time requirements by considering two factors

- Time taken to execute each statement
- Frequency of execution of each statement.

By multiplying the two items together for each statement and summing we get the total cost of executing a given piece of code. While the former is determined by the compiler, the OS and the machine the latter is a function of the code or algorithm itself. By focussing on the latter we can determine the general execution time characteristic of an algorithm irrespective of the language it is implements in or the machine it runs on. In cases where the frequency of execution give us a complex expression such as $n^3 - 2n^2 + 4n - 3$ While it possible to calculate such an expression in many practical situations it is not worth the effort. The multiplicative constants and lower order terms are insignificant compared to the input size. If the input size is sufficiently large, we can focus on the **order of growth** of the running time. When we study the asymptotic efficiency of an algorithm, we are looking at how the running time increases with the input size in the limit as the input size increases.

$$\Theta(g(n)) = \begin{Bmatrix} f(n)\colon there\ exists\ positive\ constants\ c_1, c_2, n_0 such\ that \\ 0 \le c_1 g(n) \le f(n) \le c_2 g(n) for\ all\ n\ \ge n_0 \end{Bmatrix}$$

$g(n)$ is an asymototically tight bound for $f(n)$

We ignore the lower order terms using the following notation

$$g(n) \sim f(n)$$

Which means

$$lim_{N\to\infty}\frac{f(n)}{g(n)} = 1$$

We talk about the order of growth of the algorithm as the input size N grows. The following table shows some of the most important order of growth functions.

Order Of Growth Examples

Name	Function	Code	Descrip	Example
Constant	1	`int a =10;`	*statement*	*assignment*
Logarithmic	$logN$		*halfing*	*binary search*

Linear	N	`for (int i = 0; i < 10; i++)`
		`a+=i;`
			loop	*summing*

Linearithmic	$NlogN$		*divide and conquer*	*sorting*
Quadratic	N^2		*double loop*	*pair checking*
Cubic	N^3		*Triple loop*	*triple checking*
Exponential	2^N			

All these functions, except for the exponential case can be described by the following expression

$$g(n) \sim an^b(logn)^c$$

Where a, b and c are constants. Typically, we do not state the base of the logarithm as a logarithm in one base can be converted to a logarithm in another base using a constant. So, we can absorb this with the constant a in our previous expression,

$$log_b x = log_a x \times log_b a$$

Iterative Algorithm (Insertion Sort)

```
public override void Sort<T>(T[] a)
{
        if (a == null || a.Length == 1) return;

        for (int j = 1; j < a.Length; j++)                  $c_1 n$
        {
                T key = a[j];                               $c_2(n-1)$
                int i = j - 1;                              $c_3(n-1)$


                while (i >= 0 && Less(key, a[i]))           $c_4 \sum_{j=1}^{j=n-1} t_j$
                {
                        a[i + 1] = a[i];                    $c_5 \sum_{j=1}^{j=n-1} (t_j - 1)$
                        i--;                                $c_6 \sum_{j=1}^{j=n-1} (t_j - 1)$
                }

                a[i + 1] = key;                             $c_7(n-1)$

        }
}
```

A general expression for the running time is then given by as follows. Note that the test conditions on the loops execute one more time than the body of the loop as they will execute on one iteration when the test fails.

$$c_1 n + c_2(n-1)c_2 + (n-1) + c_4 \sum_{j=1}^{j=n-1} t_j + c_5 \sum_{j=1}^{j=n-1} (t_j - 1) + c_6 \sum_{j=1}^{j=n-1} (t_j - 1) + c_7(n-1)$$

In the worst case scenario $t_j = j + 1$ and our expression becomes

$$c_1 n + c_2(n-1)c_2 + (n-1) + c_4 \sum_{j=1}^{j=n-1} (j+1) + c_5 \sum_{j=1}^{j=n-1} j + c_6 \sum_{j=1}^{j=n-1} j + c_7(n-1)$$

From the properties of series we know that

$$\sum_{j=1}^{j=n-1} (j+1) = 2 + \cdots n = \frac{n(n+1)}{2} - 1$$

$$\sum_{j=1}^{j=n-1} (j) = 1 + 2 + \cdots n - 1 = \frac{n(n+1)}{2} - n$$

Questions – Analysis of Algorithms

Big O

What is the asymptotic running time of the following?

```
public static int SearchRecursive(int[] arr, int searchKey)
{
     if (arr == null) throw new ArgumentNullException();

     return SearchRecursive(arr, 0, arr.Length - 1, searchKey);
}

private static int SearchRecursive(int[] arr, int lo, int hi, int searchKey)
{
     // The search key is not in the array. Return the complement of the index
     // at which it should be inserted.
     if (lo > hi) return ~lo;

     int median = lo + (hi - lo) / 2;
     int comparisonResult = arr[median].CompareTo(searchKey); ;

     //  a direct hit
     if (comparisonResult == 0) return median;

     return comparisonResult < 0
            ? SearchRecursive(arr, median + 1, hi, searchKey)
            : SearchRecursive(arr, lo, median - 1, searchKey);
}
```

The running time is $O(\log n)$

Why do we not care about the base of the logarithm?

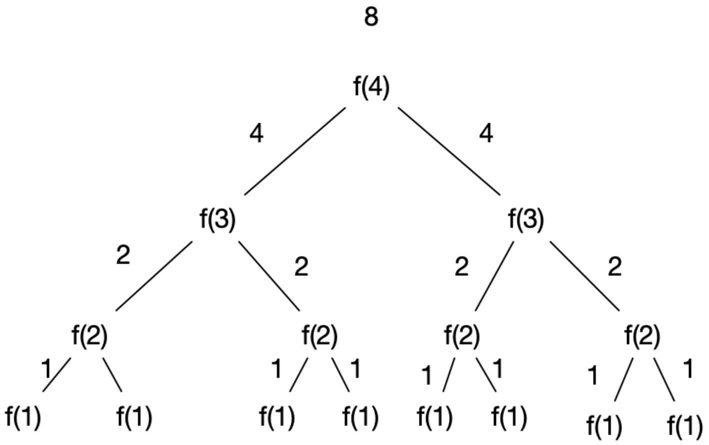
Because $\log_a x = \frac{\log_b x}{\log_b a} = \log_b x \frac{1}{\log_b a} =$

So $\log_a x$ *and* $\log_b x$ *differ by a constant factor and we don't worry about constant factors in asymptotic notation*

What is the running time of the following function and what does it do?

```
public static int Function(int n)
{
    if (n == 1)
          return 1;

    return Function(n - 1) + Function(n - 1);
}
```

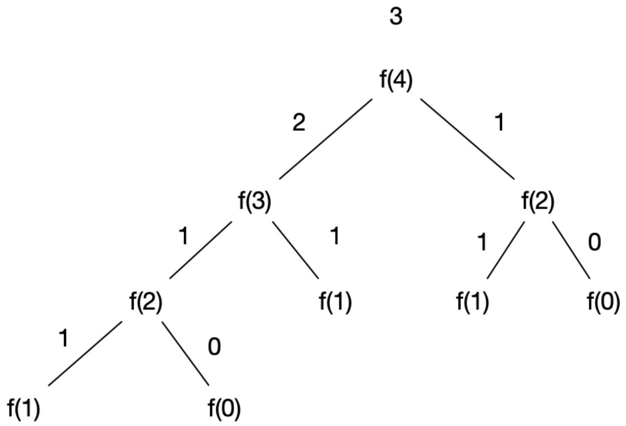
Look at the call graph of the specific case of Function(4). We get the total run time by summing the number of calls at each level. 1+2+4+8. As we move down from one level to the other each level has double the number of calls as the level before. In our case we have $1 + 2 + 4 + 8 = 2^0 + 2^1 + 2^2 + 2^3 = 2^4 - 1$



The running time is $O(2^{n-1}) = \frac{O(2^n)}{2} = O(2^n)$ *Why? Because* $2^{n-1} = \frac{1}{2^{n-1}2}$ *and we can ignore constant factors.*

What is the asymptotic running time of the following?

```
public static int FibonacciRecursive(int n)
{
    if (n == 0)
          return 0;
    if (n == 1)
          return 1;

    return FibonacciRecursive(n - 1) + FibonacciRecursive(n - 2);
}
```



The call graph for fibonacci is very similar to the previous question so $O(2^n)$ *is a correct upper bound. It is possible to prove a tighter upper bound as* $O(\phi^n)$ *where* $\phi = (1 + \sqrt{5})/2$

What is the asymptotic running time of the following?

```
public static int FibonacciRecursive(int n)
{
    if (n == 0)
          return 0;
    if (n == 1)
          return 1;

    return FibonacciRecursive(n - 1) + FibonacciRecursive(n - 2);
}
```

What is the asymptotic running time of the following?

```
public static int Function2(int n)
{
    int res = 0;
    for (int i = 0; i < n; i++)
          res += i;

    for (int i = 0; i < n; i++)
          res += i;

    return res;
}
```

The running time is $O(n)$ *We ignore the fact it is 2n as we drop the constant factors*

What is the asymptotic running time of the following?

```
public static int PairCount(int[] a)
{
    int count = 0;
    for (int i = 0; i < a.Length; i++)
    {
          for (int j = i + 1; j < a.Length; j++)
          {
                if (a[i] == a[j]) count++;
          }
    }

    return count;
}
```

The running time is $O(n^2)$. Remember that the sum of the first n integers is given by

s=$1 + 2 + \cdots (n - 2) + (n - 1) + n$

We can write 2s as

$$1 + 2 + \cdots (n - 2) + (n - 1) + n$$

$$n + (n - 1) + (n - 2) + \cdots + 2 + 1$$

$$2s = n(n + 1) \therefore s = \frac{n(n + 1)}{2}$$

So in our we are replacing n with n-1

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

In our asymptotic notation we call this $O(n^2)$ by dropping the lower order terms.

What is the asymptotic running time of the following?

```
public static void PrintPairs(int[] a, int[] b)
{
    for (int i = 0; i < a.Length; i++)
    {
        for (int j = 0; j < b.Length; j++)
        {
            Console.WriteLine($"({a[i]},{b[j]})");
        }
    }
}
```

$O(xy)$ *where x is the number of elements in a and y is the number of elements in b*

What is the asymptotic running time of the following?

```
public static void PrintPairsManyTimes(int[] a, int[] b)
{
    for (int i = 0; i < a.Length; i++)
    {
        for (int j = 0; j < b.Length; j++)
        {
            for (int k = 0; k < 10; k++)
            {
                Console.WriteLine($"({a[i]},{b[j]})");
            }
        }
    }
}
```

$O(10xy)$ *where x is the number of elements in a and y is the number of elements in b which is of course just* $O(xy)$

What is the asymptotic running time of the following?

```
public void ReverseArray(int[] a)
{
    for (int i = 0; i < a.Length/2; i++)
    {
        int temp = a[i];
        a[i] = a[a.Length-1-i];
        a[a.Length-1-i] = temp;
    }
}
```

$O(n)$ *We ignore the constant factor of* $\frac{n}{2}$

What is the asymptotic running time of sorting each string in an array and then sorting the array itself?

If we let the number of strings in the array be n and the length of each string be l then sorting each string takes $O(l \log l)$ We have to do this n time so we get $O(n \times l \log l)$ The sorting of the array itself is $O(n \log n)$ but each string comparison requires l character compares in the work case so it is actually $O(l \times n \times \log n)$ Adding the two thing together we obtain

$$O(n \times l \log l + l \times n \times \log n) = O(nl(\log l + \log n) =$$

What is the asymptotic running time of the following code?

```
public static bool IsPrimeNaive(int x)
{
        if (x <= 1) return false;

        for (int i = 2; i < x; i++)
        {
                if (x % i ==0)
                        return false;
        }
        return true;
}
```

The runtime is then $O(x)$

What is the asymptotic running time of the following code?

```
public bool IsPrimeUsingSquareRoot(int n)
{
        if (n < 2)
                return false;

        if (n == 2)
                return true;

        // The definition of a prime is an integer x
        // which is not exactly divisible by any
        // number other than itself and one. If a
        // number x is not prime it can be written as
        // the product of two factors a x b. If both
        // a and b were greater than the square root of
        // x then a x b would also be greater than x and hence
        // a x b is not x. SO testing all factors up to floor(root(x))
        // is sufficient as if one factor is floor(root(x)) the other factor must
        // be less than that

        // hence test the n-2 integers from
        // 2,..., Floor(Root(N))
        return Enumerable.Range(2, (int)Math.Floor(Math.Sqrt(n)))
                .All(i => n % i > 0);
}
```

The runtime is then $O(\sqrt{n})$

What is the asymptotic running time of the following code?

```
public static int Factorial(int x)
{
    if (x ==0) return 1;
    return x * Factorial(x-1);
}
```

The running time is simple $O(x)$

Sorting

Overview

We will consider Sorting different sorting algorithms in turn. Before we do the following table shows when we might want to use each one

Appropriate Sorting Algorithm

Description	**Algorithm**
Small number of elements	Insertion Sort
The collection is already mainly sorted	Insertion sort
Easy to code	Insertion sort / Selection sort
Want very good average case behaviour	Quick sort
The algorithm must be stable	Merge sort
Need good performance in worst case	Heap sort

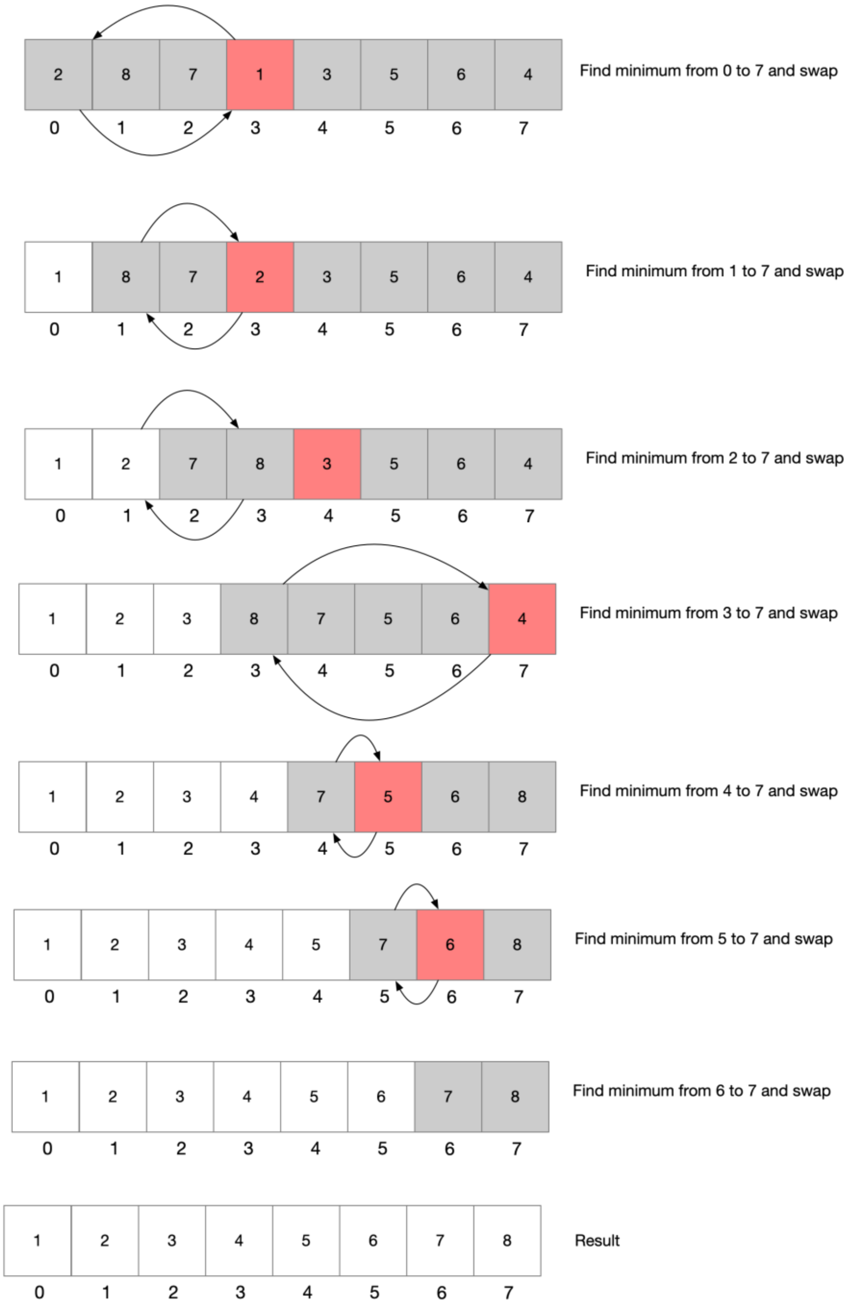
> **Stable Sort**
>
> A stable sort keeps elements with the same key in the same order relative to each other in the sorted output

Lower Bound for Comparison based algorithms

We can informally show why $N \times log_2 N$ is a theoretical lower bound for comparison-based algorithms as follows. Sorting involves selecting a single permutation from all possible permutations of the input array. If the input array has N elements there are N! permutations. At each stage of a comparison-based algorithm we do a comparison which in the best case halves the number of possible permutations. This gives us a best case of $log_2 N!$ Stirling's approximation shows that $log_2 N! \approx N \times log_2 N$

Selection Sort

Selection Sort is perhaps the simplest sorting algorithm to implement. It works be searching the entire array for the smallest element and inserting it in in the first position. Then it searches the entire array from the second element onwards and inserts it in the second position and so on



Notice how many comparisons we need to do at each stage for our 8 element array. We do 7 then 6 then 5 then 4 then 3 then 2. In general, we have $(n-1)+(n-2)+\ldots+2+1=\sum_{i=1}^{n-1} i=\frac{n(n-1)}{2}=O(n^2)$ The following diagrams shows visually this result. The grid has $n^2=36$ elements and half of these elements are involved in the comparisons.

i	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
0	6	5	4	3	2	1
1	1	5	4	3	2	6
2	1	2	4	3	5	6
3	1	2	3	4	5	6
4	1	2	3	4	5	6
5	1	2	3	4	5	6

The source code is as follows

```
public T[] SelectionSort<T>(T[] a) where T : IComparable<T>
{

    for (int i = 0; i < a.Length; i++)
    {
          int minIdx = i;
          for(int j=i+1;j<a.Length;j++)
          {
                if ((a[j].CompareTo(a[minIdx])) <0) minIdx = j;
          }
          Swap(a,i,minIdx);
    }

    return a;
}
```

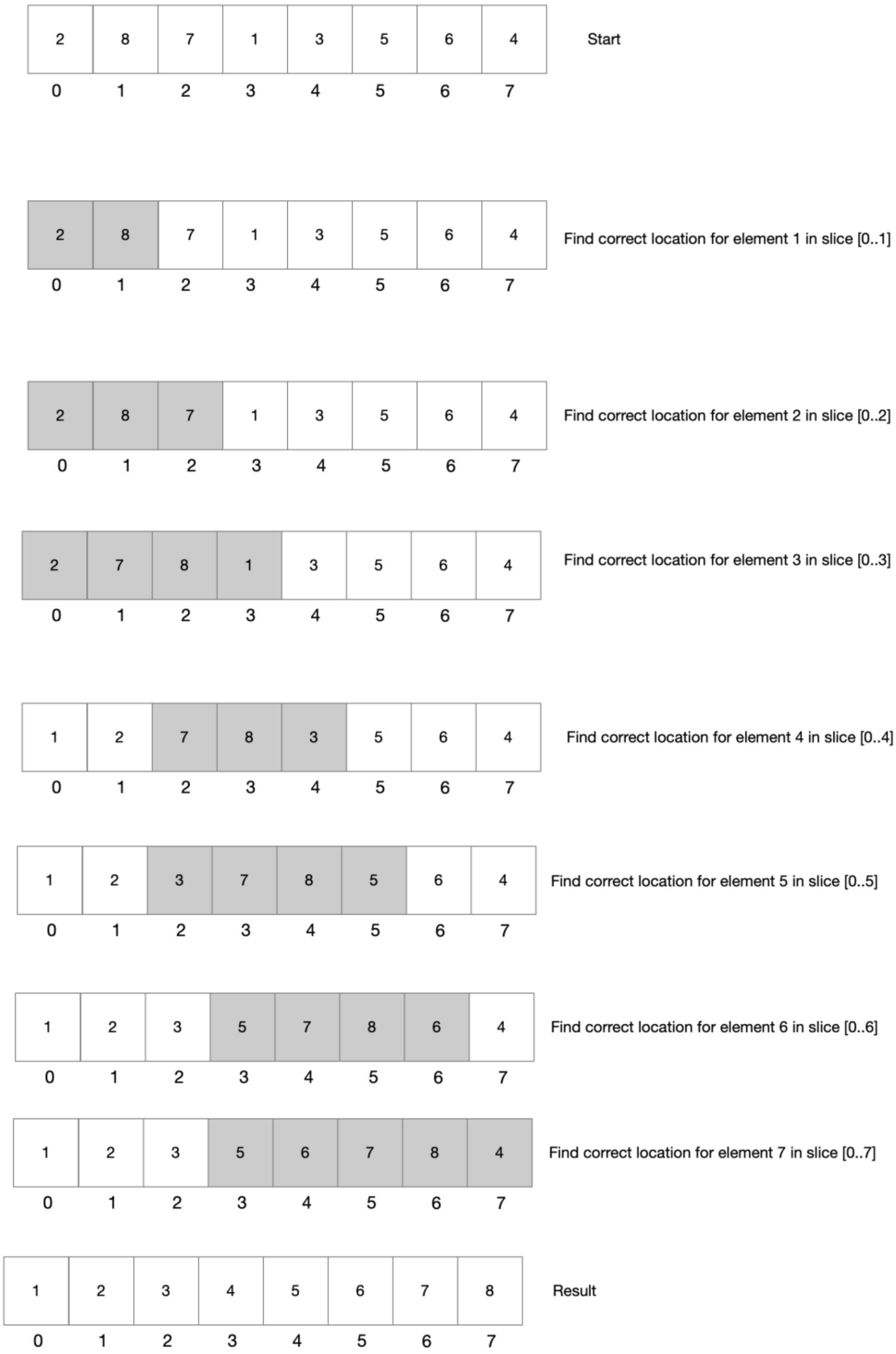
The performance is easily analysed by looking at the two loops. The outer loop runs from 0 to (n-1) giving n iterations. Each iteration of the inner loop runs from i+1 to n-1 giving us

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

Selection sort is quadratic in the best, worst and average case. The input used has no impact on the running time.

Insertion Sort

Insertion sort works like sorting a hand or cards



2	8	7	1	3	5	6	4
0	1	2	3	4	5	6	7

Start

2	8	7	1	3	5	6	4
0	1	2	3	4	5	6	7

Find correct location for element 1 in slice [0..1]

2	8	7	1	3	5	6	4
0	1	2	3	4	5	6	7

Find correct location for element 2 in slice [0..2]

2	7	8	1	3	5	6	4
0	1	2	3	4	5	6	7

Find correct location for element 3 in slice [0..3]

1	2	7	8	3	5	6	4
0	1	2	3	4	5	6	7

Find correct location for element 4 in slice [0..4]

1	2	3	7	8	5	6	4
0	1	2	3	4	5	6	7

Find correct location for element 5 in slice [0..5]

1	2	3	5	7	8	6	4
0	1	2	3	4	5	6	7

Find correct location for element 6 in slice [0..6]

1	2	3	5	6	7	8	4
0	1	2	3	4	5	6	7

Find correct location for element 7 in slice [0..7]

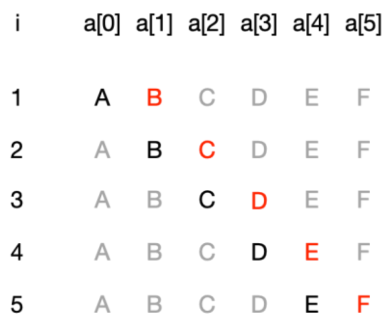
1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7

Result

The performance of this algorithm depends on the input data.

Best Case

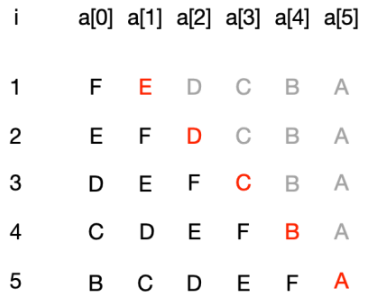
In the best case we have $n-1$ comparisons



i	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
1	A	B	C	D	E	F
2	A	B	C	D	E	F
3	A	B	C	D	E	F
4	A	B	C	D	E	F
5	A	B	C	D	E	F

Worst Case

In the worst case the number of comparisons is given by $\frac{n(n-1)}{2}$ The following diagram shows why. We have $1+2+\cdots+(n-1)=\frac{n(n-1)}{2}$



i	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
1	F	E	D	C	B	A
2	E	F	D	C	B	A
3	D	E	F	C	B	A
4	C	D	E	F	B	A
5	B	C	D	E	F	A

Average Case

The average case is half as bad as the worst case so we get $\frac{n^2}{4}$

Source Code

```
public override void Sort<T>(T[] a)
{
        if (a == null || a.Length == 1) return;

        for (int i = 1; i < a.Length; i++)
        {
                for (int j = i; j > 0 && Less(a[j], a[j - 1]); j--)
                {
                        Switch(a, j - 1, j);
                }
        }
}

public override void Sort<T>(T[] a)
{
        if (a == null || a.Length == 1) return;                          $c_1$

        for (int i = 1; i < a.Length; i++)                               $c_2(n)$
        {
                for (int j = i; j > 0 && Less(a[j], a[j - 1]); j--)      $c_3 \sum_{j=1}^{j=n-1} t_j$
                {
                        Switch(a, j - 1, j);                             $c_4 \sum_{j=1}^{j=n-1} t_j$

                }
        }
}
```

Sum of First N Integers

$$P(n) 1 + 2 + \cdots + n - 1 + n = \frac{n(n+1)}{2}$$

Base Case

Show the hypothesis hold for n = 0

$$S(0) = \frac{1(0)}{2} = 0$$

Inductive hypothesis

Assume P(k) hold for some unspecified value of k

$$P(k) 1 + 2 + \cdots + k - 1 + k = \frac{k(k+1)}{2}$$

Show that if the hypothesis holds for k it holds for k+1. We need to show that

$$(1 + 2 + \cdots + k - 1 + k) + (k+1) = \frac{(k+1)\big((k+1)+1\big)}{2}$$

Using the inductive hypothesis the left hand side can be written as

$$\frac{k(k+1)}{2} + (k+1)$$

Re-arranging this we get

$$\frac{k(k+1) + 2(k+1)}{2}$$

Factoring out on the numerator

$$\frac{(k+1)(k+2)}{2} = \frac{(k+1)\big((k+1)+1\big)}{2} = rhs$$

Sum of First N-1 Integers

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

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

Let

$$S = 1 + 2 + \cdots + n - 1 + n$$

$$2S = 1 + 2 + \cdots + n - 1 + n + 1 + 2 + \cdots + n - 1 + n$$

$$2S = 1 + 2 + \cdots + n - 1 + n +$$

$$n + n - 1 + \cdots + 2 + 1$$

$$2S = n(n+1)$$

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

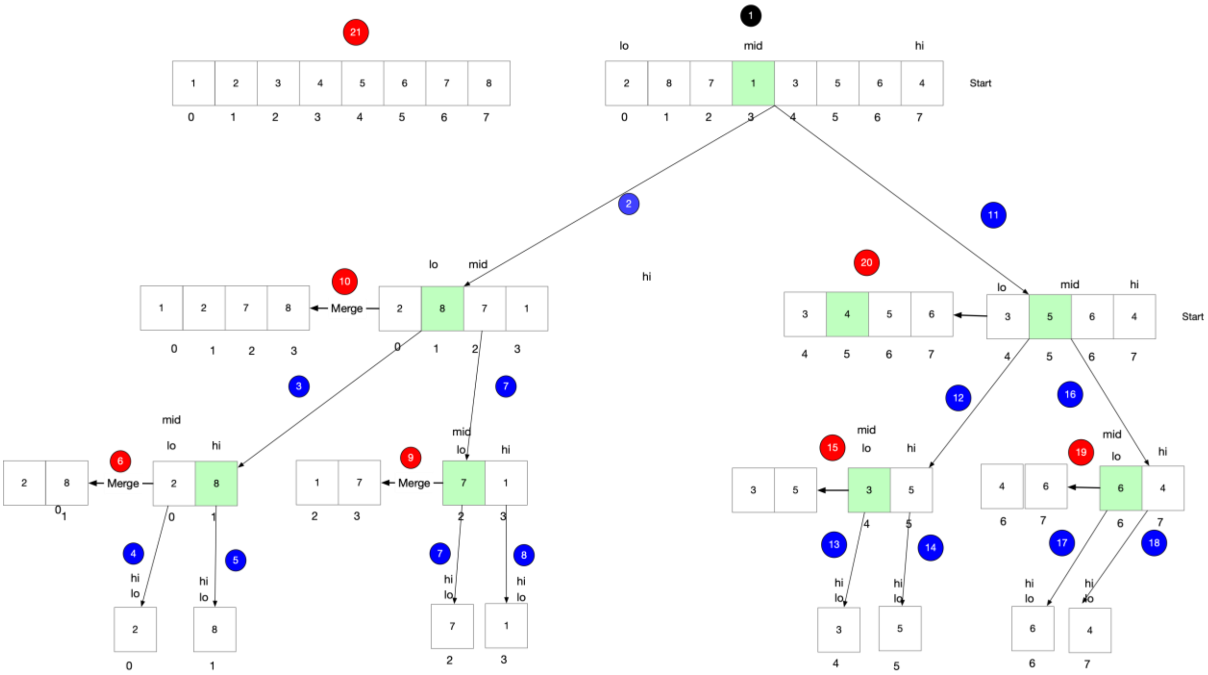
Proof by induction

$$S(1) = \frac{1(2)}{1} = 1$$

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

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

Merge Sort



```
void MergeSort<T>(T[] a, int lo, int hi) where T : IComparable<T>
{
    if (lo <hi)
    {
          int mid = (lo + hi) / 2;
          MergeSort(a,lo,mid);
          MergeSort(a,mid+1,hi);
          Merge(a,lo,mid,hi);
    }
}

void Merge<T>(T[] a, int lo, int mid, int hi) where T : IComparable<T>
{
    // Populate the left array
    T[] left = new T[mid - lo+1];
    for (int i = 0; i < left.Length; i++) left[i] = a[lo + i];

    // Populate the right array
    T[] right = new T[hi - mid];
    for (int i = 0; i < right.Length; i++) right[i] = a[mid +1+ i];

    // Start the merge
    int lIdx = 0, rIdx = 0;
    for (int i = lo; i <= hi; i++)
    {
          if (lIdx == left.Length) a[i] = right[rIdx++];
          else if (rIdx == right.Length) a[i] = left[lIdx++];
          else if (left[lIdx].CompareTo(right[rIdx]) < 0) a[i]
                = left[lIdx++];
          else a[i] = right[rIdx++];
    }
}
```

Merge sort guarantees to sort N items in time proportional to NLogN. It is perhaps the best-known example of the power of the divide and conquer paradigm. Its biggest downside is it requires bigger extra space proportional to N.

Quick Sort

For most typical applications quick sort is much faster than any other sorting algorithm. The implementation is based on the divide and conquer technique. In the worst case it does tend to quadratic but this is rare and we can make it better by introducing randomisations.

```
public void QuickSort<T>(T[] a, int lo, int hi) where T : IComparable<T>
{
    if (hi>lo)
    {
          int partition = Partition(a,lo,hi);
          QuickSort(a,lo,partition-1);
          QuickSort(a,partition+1,hi);
    }
}

int Partition<T>(T[] a, int lo, int hi) where T : IComparable<T>
{
    T pivot = a[hi];

    int i=lo-1;
    for(int j=lo;j<hi;j++)
    {
          // The element being compared is greater than the pivot
          // simply move the upper bound of the greater section which is
          // maintained by j
          if (a[j].CompareTo(pivot)>0)
          {
          }
          else
          {
                // The element being compared is leq the pivot. Move the
                // index  of smaller items (i) by one which temporarily
                // includes it in the smaller section
                i++;

                // Now we swap i and j so put the bigger element in the
                // big bucket and the small one in the small bucket
                Swap(a,i,j);
          }
    }

    // Finally we put the pivot in its place. We put it in the
    // first location above the small bucket meaning we know
    // we swap it with a bigger element
    Swap(a,i+1,hi);

    return i+1;
}
```

Counting Sort

Can be used with certain classes of data such as integers. If the number of possible values is small we can get very good constant time performance. It is not a comparison-based sort.

```
public int[] Sort(int[] a, int k)
{
    // Elements in a must be an integer between 0 and k
    // inclusive.
    int[] counts = new int[k + 1];

    // Counts now holds the frequencies of each integer
    // 0..k in the input array a
    for (int i = 0; i < a.Length; i++) counts[a[i]]++;

    // Each index i in counts now holds the number of
    // elements in the input array with value <= i
    for (int i = 1; i < counts.Length; i++) counts[i]
          = counts[i] + counts[i - 1];

    // Create an array to hold the sorted results.
    int[] b = new int[a.Length];

    // Walk back through a from a[i] to a[0]
    for (int i = a.Length - 1; i >= 0; i--) {

          // The value in the source array
          int x = a[i];

          // The number of elements in the input
          // array whose value is less than or
          // equal to x.
          int n = counts[x];

          // If there are n elements less than
          // or equal to x, then x must be the
          // nth element in the sorted list. In
          // a 0 based array the nth element is at
          // index n-1
          b[n-1] = x;

          // Decrement the number in counts[x]
          counts[x]=counts[x]-1;
    }

    return b;
}
```

If the elements in the input array of n elements are constrained to be in the set 0…k then the performance of CountingSort is O(n+k). Counting Sort is stable.

Bucket Sort

Questions – Sorting

Explain why $N \times log_2 N$ is a lower bound for comparison based algorithms?

Sorting involves selecting a single permutation from all possible permutations of the input array. If the input array has N elements there are N! permutations. At each stage of a comparison-based algorithm we do a comparison which in the best case halves the number of possible permutations. This gives us a best case of 〖log〗 _2 N! Stirling's approximation shows that 〖log〗 _2 N! ≈N× 〖log〗 _2 N

Selection Sort

Give an algorithm for insertion sort

```
public T[] Sort<T>(T[] a) where T : IComparable<T>
{
    for (int i = 0; i < a.Length; i++)
    {
          int minIdx = i;

          for (int j = i+1; j < a.Length; j++)
                if (a[j].CompareTo(a[minIdx]) < 0) minIdx = j;

          T temp = a[i];
          a[i] = a[minIdx];
          a[minIdx] = temp;
    }
    return a;
}
```

What is the performance?

$\frac{n^2}{2}$ *In the best, worst and average case. Performance is insensitive to input*

Symbol Tables

Hash Tables

Hash Function

Separate Chaining

Overview

Questions -Symbol Tables

Overview

What is the beauty of HashTable?

Can strike a balance between space and time tradeoffs

What is the run time performance of a HashTable?

Amortized constant time

When would we not use a HashTable?

Order is important.

The following operations are also linear and inefficient in hashing

Find maximum/minimum key

Find keys in a range

What is the space of a HashTable

O(N+M) where N is the number of keys and M is the number of buckets

When would we use a BST over a hashtable?

When order operations are important

Hash Tables

What are the two parts of a HashTable?

Hash function to transform keys to array indices

Collision resolution strategy

What are the most common collision resolution strategies?

Separate chaining

Linear probing

What is the beauty of HashTable?

Enables one to strike a reasonable balance between time and space tradeoffs

What are the performance characteristics of HashTables?

Search and insert require amortized constant time

What does what need when creating a hashtable?

Array of Linked Lists

Hash code function

Map hash code to index in the array using the remainder operator

What is the performance of your solution?

If the number of collisions are low then O(1) and if the number of collisions are high tends to O(n)

When would this happen?

Weird data

Bad hash function

Given an alternative?

Chaining with binary search tree

Reduce worst case to O(log n)

Given an another alternative?

Open addressing with linear probing

Give a recommendation for the number of buckets?

A prime number ?

Why?

If the keys are base 10 numbers and the number of buckets are 10^k *only k digits of the keys will be used when mapping to buckets due to the nature of modulo arithmetic*

How would we hash floating point numbers?

We could do modulo hashing on the floats binary representation to take into account all bit.

How would we hash strings?

Treat the string as a N-digit base R integer

What are the characteristics of a good hash function?

Consistent – equal keys provide the same hash value

Efficient to compute

Uniformly distributes the keys

What is the benefit of separate chaining?

If space is not important we can choose M large to get constant time

If space is important we can still get a factor of M improvement by letting M be as large as we can afford

When would one not use a HashTable?

Order is important

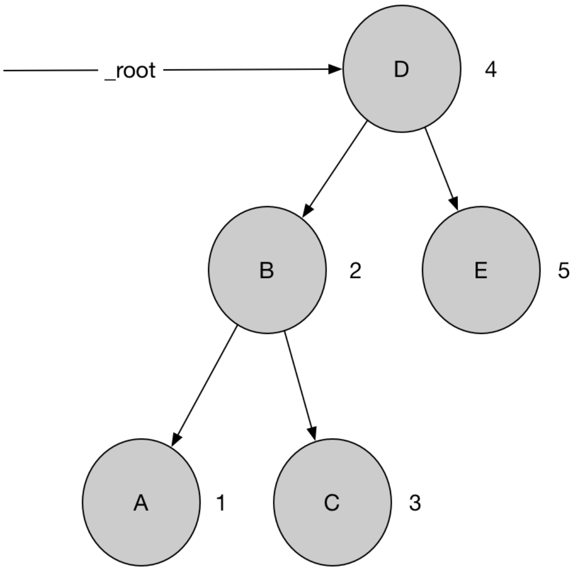
Find maximum/minimum key

Find keys in a range

All these operations take linear time

BINARY SEARCH TREES

Write code to add a node to a binary tree given the root node. Use it to build the following



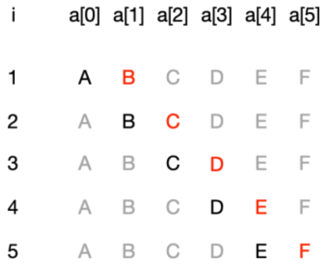
INSERTION SORT

InsertionSort

```
public T[] Sort<T>(T[] a) where T : IComparable<T>
{
    for (int i = 1; i < a.Length; i++)
    {
          for (int j = i; j > 0 && a[j].CompareTo(a[j - 1]) < 0; j--)
          {
                T temp = a[j];
                a[j] = a[j - 1];
                a[j - 1] = temp;
          }
    }
    return a;
}
```

What is the runtime of this algorithm in the best case?

In the best case we have $n - 1$ comparisons



i	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
1	A	B	C	D	E	F
2	A	B	C	D	E	F
3	A	B	C	D	E	F
4	A	B	C	D	E	F
5	A	B	C	D	E	F

What is the runtime of this algorithm in the worst case?

In the worst case the number of comparisons is given by $\frac{n(n-1)}{2}$ The following diagram shows why. We have $1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2}$

i	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
1	F	E	D	C	B	A
2	E	F	D	C	B	A
3	D	E	F	C	B	A
4	C	D	E	F	B	A
5	B	C	D	E	F	A

What is the runtime of this algorithm in the average case?

In the average case we do half the work of the worst case $\frac{n(n-1)}{4}$

Quick Sort

Implement QuickSort

```
public int Paritition<T>(T[] A, int p, int r) where T : IComparable<T>
{
     // Select one element that will form the pivot. We select the
     // last element in the slice we are Partitioning
     var pivot = A[r];

     // We maintain three slices of the array.
     //      A[p..i] has elements <= pivot
     //      A[i+1..j-1] has elements > pivot
     //    A[j..r-1] has elements that can be < or > pivot
     int i = p-1;
     for (int j = p;j <= r-1; j++)
     {
            if (A[j].CompareTo(pivot) > 0)
            {
                   // If the element at index j
                   // if greater than the pivot we
                   // do nothing other than loop around
                   // thus increasing j and adding 1 element
                   // to A[i+1..j-1]
            }
            else
            {
                   // If the element at index j
                   // if less than or equal to the pivot
                   // we increase i which brings in an
                   // element >= pivot into the less that
                   // bucket. But we fix the issue by
                   // exchanges the greater than element
                   i++;
```

```
                Swap(A,i,j);
            }
        }

        // Switch the pivot with the first element greater than x
        // to maintain the loop invariant
        Swap(A,i+1,r);
        return i+1;
    }
```

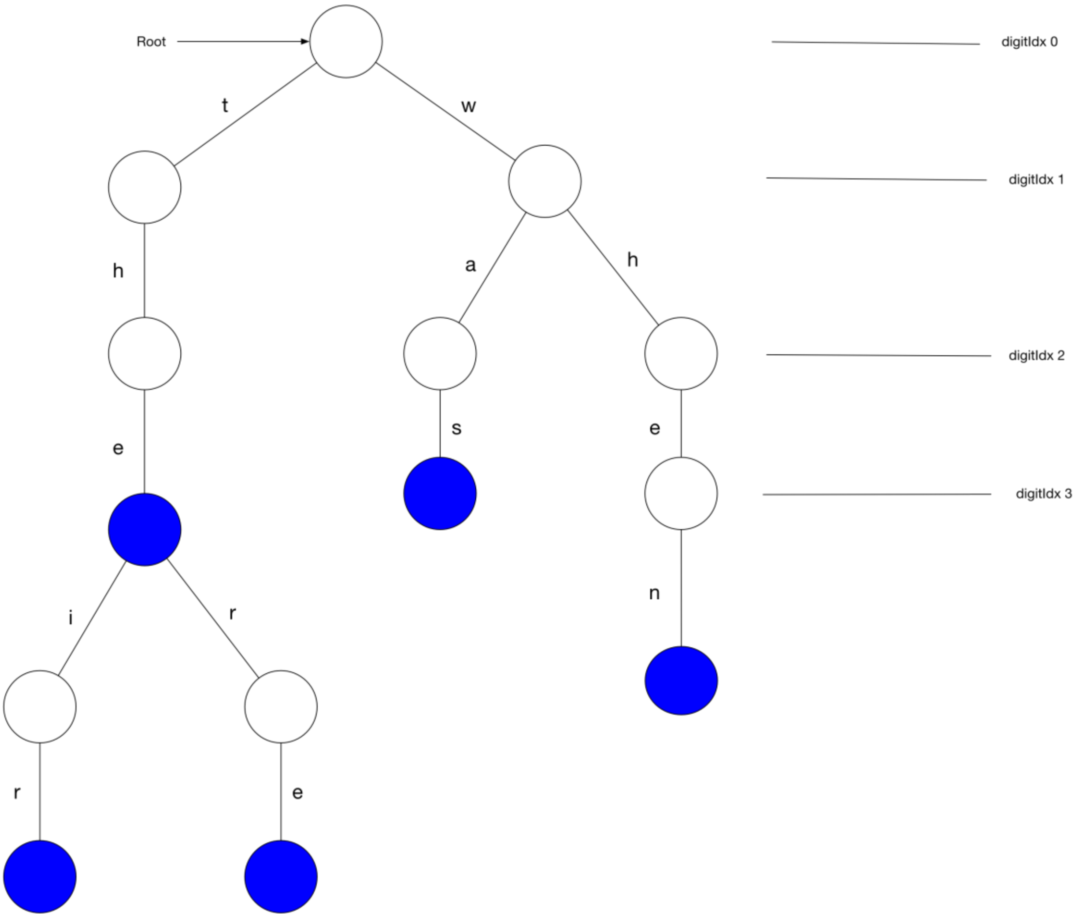
What is the performance

Worst Case O(N^2) and average case NLogN

Arrays and Strings

Tries / Suffix Tree

Each edge represents a digit from the given alphabet. Each vertex with a value represents a complete word. The root vertex represents the empty string.



Inserting Keys

Code to inset nodes can be specified iteratively

```
private void Put(String key, TV value)
{
      Node<TV> current = _root;

      for (int digitIdx = 0; digitIdx < key.Length; digitIdx++)
      {
            int currentDigit = key[digitIdx];

            // Is there an edge representing the currentDigit?
            // Such as edge would be implied by a entry in the
            // current nodes ChildNodes at index currentDigit
            if (current.ChildNodes[currentDigit] == null)
            {
                  // If no such edge exists we create it by
                  // constructing a new vertex and inserting it
                  // into the ChildNodes collection
                  // at index currentDigit
                  current.ChildNodes[currentDigit] =
                        new Node<TV>(_radix);
            }

            // Iterate to the next node
            current = current.ChildNodes[currentDigit];
      }

      // mark last node as leaf by setting a value on it
      current.Value = value;
}
```

Retrieving Keys

Questions -Arrays and Strings

What is the runtime of concatenating n strings of length x characters?

$$O(xn^2)$$

As we add each string we create a new string of length of previous string + x and copy characters. We get a total number of copied characters of

$$x + 2x + 3x + \cdots .. + nx = x(1 + 2 + \cdots + n) = x\frac{n(n+1)}{2}$$

Tries / Suffix trees

What is a trie?

A special form of N-ary tree

What is the special property of a Trie?

All descendants of a node have a common prefix of the string associated with that node

Given string retrieval is O(1) is both a Trie and a HashTable when would one use a TrieWhat general classification of operations are Tries good for?

If we want to quickly find all strings which have a given string as the prefix. This is O(n) in a hash table but much less for a trie.

What is the downside of a Trie?

Rarely space efficient. Each character is a very least a reference which is multiple bytes rather than an ascii character

When would one use a HashTable over Trie?

Just need to do lookups of complete strings

Hashtable requires less space

Likely to be more efficient. Trie nodes more likely to be in non-contiguous memory which is does not lend itself to efficient use of the CPU cache

Hash sets and hashtable are not cache friendly but with hashing you only do one non sequential memory lookup and a hashmap does k non sequential lookups

What operations are good with a Trie

Prefix operations

What can we do specifically?

Find all strings which start with a given prefix

Find all keys that match a wildcard

Find the longest prefix of a given string.

Find the shortest prefix of a given string.

Find the largest common substring

Palindrome

What is the performance of a trie for inserts and search miss?

O(k) where k is the number of keys in the trie.

What is the performance of a trie for search hits?

Approximately $log_k N$

What is the space requirement of a trie?

Each node in the trie has R links. If we have N keys we have at least N different nodes to the lower bound is O(NR) In the worst case every key has a different first character. In such a case the space if O(NRw) where w is the average key length

If we don't need the prefix logic. What is likely to be the better choice, a HashMap or a Trie

Priority Queues

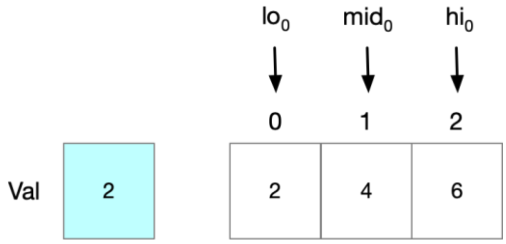
Searching

Binary Search

Consider searching for the value 2 in the array {2,4,6} We start by setting the hi and lo markets to the first and last element in the array and calculate the midpoint as

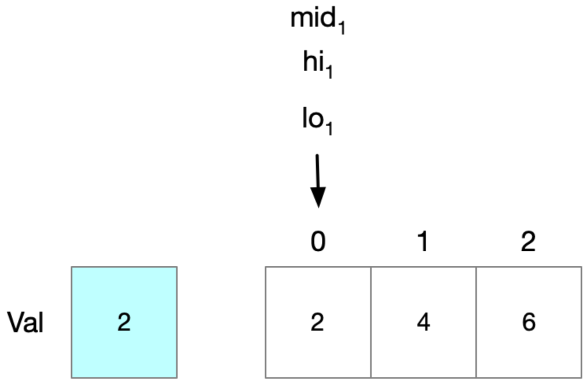
$$mid_0 = lo_0 + \frac{(hi_0 - lo)}{2}$$

Visually we have this



The search value is less than mid so we know it lies between in the closed interval $[lo_0, mid_0 - 1]$ We adjust out bounds and start iteration 1. We calculate a new midpoint.

$$mid_1 = lo_1 + \frac{(hi_1 - lo_1)}{2}$$



Now the search value matches our mid point so we return the index of the mid point. Say instead we had been searching for the value zero which is not in the array. At this point we would say that the value is before mid so we would set hi to be mid-1 which is less that lo. This shows us our two terminating conditions.

- Hit: The element on the mid point matches the search value
- Miss: The hi marker is less than the low marker

```
public int SearchIterative<T>(IList<T> arr, T val)
  where T : IComparable<T>
{
    if (arr == null)
          throw new ArgumentNullException();

    int loIdx = 0;
    int hiIdx = arr.Count - 1;
    while (loIdx <= hiIdx)
    {
          int miIdx = loIdx + (hiIdx - loIdx) / 2;
          int comp = val.CompareTo(arr[miIdx]);

          if (comp == 0)
                return miIdx;

          if (comp > 0)
                loIdx = miIdx + 1;
          else
                hiIdx = miIdx - 1;
    }

    return ~loIdx;
}
```

Questions – Searching

Binary Search

Implement Iterative Binary Search and state its asymptotic run time?

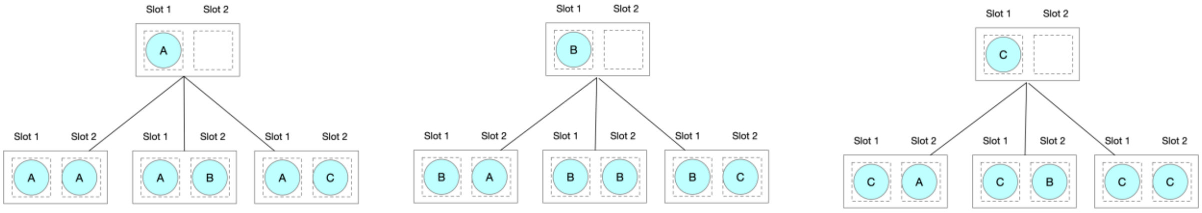
```
public int SearchIterative<T>(IList<T> arr, T val)
  where T : IComparable<T>
{
    if (arr == null)
          throw new ArgumentNullException();

    int loIdx = 0;
    int hiIdx = arr.Count - 1;
    while (loIdx <= hiIdx)
    {
          int miIdx = loIdx + (hiIdx - loIdx) / 2;
          int comp = val.CompareTo(arr[miIdx]);

          if (comp == 0)
                return miIdx;

          if (comp > 0)
                loIdx = miIdx + 1;
          else
                hiIdx = miIdx - 1;
    }

    return ~loIdx;
}
```

The runtime is O(logN)

Combinatorics

K-Strings

Consider the set $S = \{A, B, C\}$ If we use this set to form two-character strings we have the following situation For each of the three possible values of the first slot we have 3 possible values for the second slot.



We have 3^2 permutations. In full generality there are n^k ways of forming different k-strings over a Set of size n . The following code shows how we can generate the k-strings very simply using recursion

```
public static void GenerateKStrings<T>(T[] kString, T[] set, int
position,
    Action<T[]> visit)
{
    if (position == kString.Length)
    {
          visit(kString);
          return;
    }

    for (int kStringIdx = 0; kStringIdx < set.Length; kStringIdx++)
    {
          kString[position] = set[kStringIdx];
          GenerateKStrings(kString, set, position + 1, visit);
    }
}
```

A slightly more complex iterative algorithm is given as follows

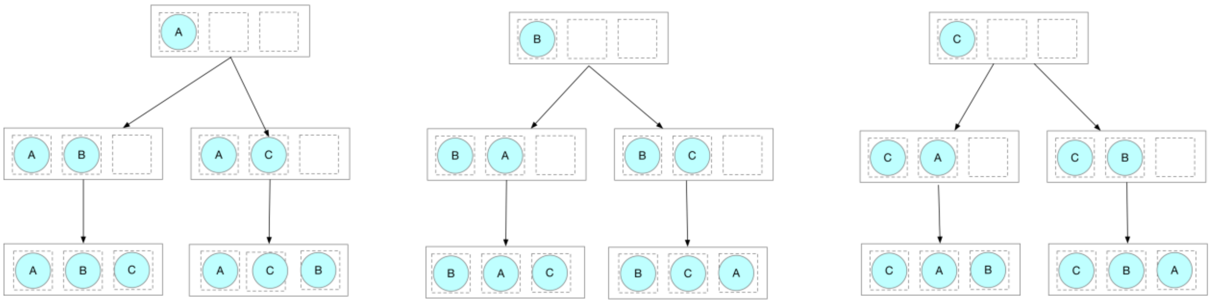
```
public static IEnumerable<T[]> GenerateKStrings<T>( T[] S,int k)
{
    // Holds a  k digit number where each digit is of base equal to
    // the number of elements in S. So if there are two character in S
    // the digits in this number are binary.
    //
    // Each digit forms a index into the set S telling us exactly which
    // element of S forms the character at the correspondong location
    // in the current kstring. So if we had k=3 and S{'a','b'} then
    // a seqIndices of {0,1,1} would correspond to the k-string of
    // {'a','b','c'}
    int[] seqIndices = new int[k];

    while (true)
    {
           // Process the current value. Convert indices to elements
           T[] kstring = new T[k];
           for (int i = 0; i < k; i++)
                  kstring[i] = S[seqIndices[i]];

           yield return kstring;

           // In this algorithm we treat  the indices array as a
           // n digit number where the base of each digit is determined by the
           // number of elements in that digits corresponding event array from
           // events. Moving to the next n-tuple is then a  case of
           // incrementing the  n-digit number held in seqIndices. To
           // this we need to take care of overflow which is what
           // the following loop condition does.
           int j = 0;
           while (j < k && seqIndices[j] == S.Length - 1)
           {
                  seqIndices[j] = 0;
                  j++;
           }

           // If j is greater than the last element in seqIndices
           // we have overflowed off the end of seqIndices.
           // In this case the work of this algorithm is done
           // and we have visited all n-permutations
           if (j == k)
                  break;

           seqIndices[j]++;
    }
}
```

The runtime of this algorithm is clearly n^k

Permutations

From an Introduction to Algorithms

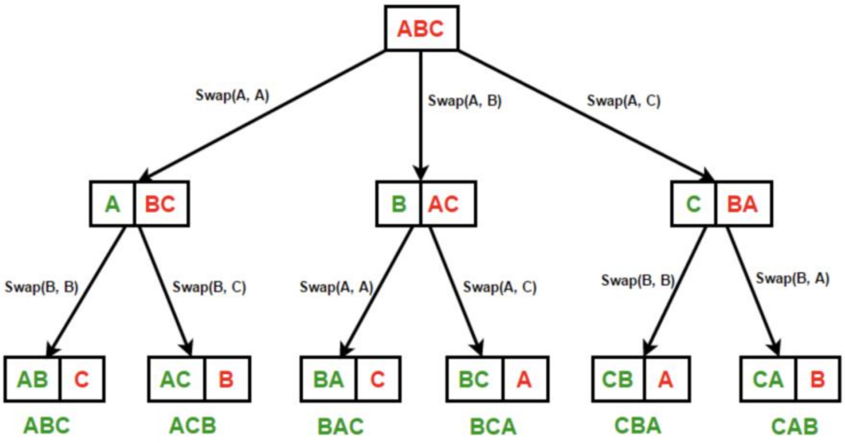
“A permutation of a finite set S is an ordered sequence of the elements of S, with each element appearing exactly once. “ So we can permutate a 3 element set as follows.



In general, there are n! permutations of a Set of n elements. One simple algorithm is as follows.

```
public void GeneratePermutations<T>(HashSet<T> set, T[] permutation, int
permIdx, Action<T[]> visitFunc) where T : IComparable<T>
{
    if(permIdx == permutation.Length)
    {
          visitFunc(permutation);
          return;
    }

    foreach (var element in set)
    {
          permutation[permIdx] = element;
          var clonedSet = new HashSet<T>(set);
          clonedSet.Remove(element);

    GeneratePermutations(clonedSet,permutation,permIdx+1,visitFunc);
    }
}
```

Another recursive algorithm based on swaps is given as



```
public static void GeneratePermutations<T>(T[] S, int k, Action<T[]>
visit)
{
    if (k==S.Length-1)
    {
          visit(S);
          return;
    }

    for (int i = k; i < S.Length; i++)
    {
          Swap(S,k,i);
          GeneratePermutations(S,k+1,visit);
          Swap(S,k,i);
    }
}
```

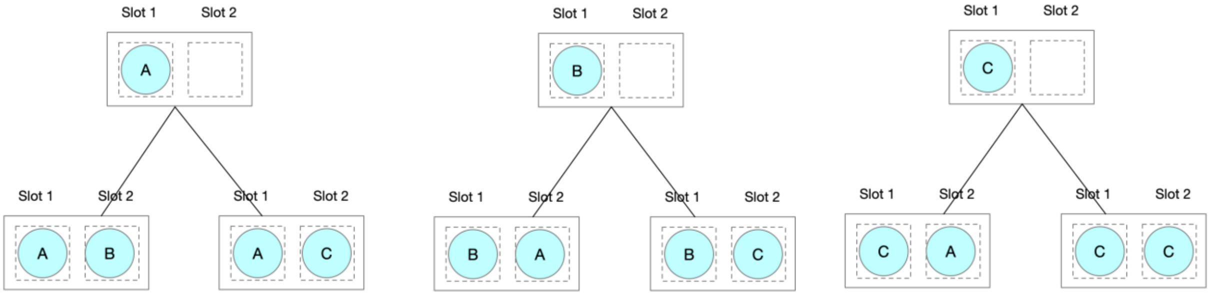
A more complex and significantly faster algorithm is given by Heap’s algorithm

```
public void Generate(int N, int[] arr, Action<int[]> processPerm)
{
      if (N == 1) processPerm(arr);

      for (int c = 0; c < N; c++)
      {
            // For the current value of arr[N-1]
            // lock it down and permutate the array
            // arr[0..N-2]
            Generate(N - 1, arr, processPerm);

            // If N is even permutate arr[N-1] with arr[c]
            if (N % 2 == 0)
                  Swap(arr, c, N - 1);
            // If N is odd permutate arr[N-1] with arr[0]
            else
                  Swap(arr, 0, N - 1);
      }
}
```

K-Permutations

From an Introduction to Algorithms

“A k-permutation of S is an ordered sequence of k elements of S, with no element appearing more than once in the sequence. (Thus an ordinary permutation is a n-permutation of an n-set)”



We have three choices for the first slot. But for each of those choices we only have two choices for the second slot as once value from the set have been used up. So we have 3 × 2 ways of taking 3 objects 2 at a time without repetition. In full generality we have

$$P_k^n = n \times (n-1) \times (n-2) \times \ldots \times (n-r+1)$$

Ways of taking n objects r at time without repetition. R must be less than or equal to n. This can be expressed as $P_k^n = \frac{n!}{(n-r)!}$. More formally this gives the number of different ordered arrangements of an r element subset of a n set

We show why in the following section

Proof

$$n! = n \times (n-1) \times \ldots \times (n-r+1) \times (\boldsymbol{n}-\boldsymbol{r}) \times (\boldsymbol{n}-\boldsymbol{r}-\mathbf{1}) \times\times \mathbf{2} \times \mathbf{1} \qquad (\mathbf{1})$$

$$(n-r) \times (n-r-1) \times (n-r-2) \times \ldots \times 2 \times 1 = (n-r)! \qquad (2)$$

Substituting (2) into (1)

$$n! = n \times (n-1) \times (n-2) \times \ldots \times (n-r+1) \times (n-r)! \qquad (3)$$

And re-arranging

$$n \times (n-1) \times (n-2) \times \ldots \times (n-r+1) = \frac{n!}{(n-r)!}$$

Permutation of Multi-Sets

If we have a set rather than a multi-set we can permutate using the following algorithm

```
public IEnumerable<T[]> GeneratePermutations<T>(T[] multiSet) where T : IComparable<T>
{
     // To be clear on our terminology. We assume the given initialPermutation
     // is the permutation with lowest lexicographical value. As such we expect
     // a[0] <= a[1] <= ... <= a[n-1] Put another way we expect the values in the
     // permutation to be in weakly increasing (non-decreasing) order from index 0
     // through to index n-1. For the purposes of this method we will refer to the element
     // with the highest value of j as the rightmost element and the value with the lowest
value
     // of j as the left most element
     //
     //     Left/MSB  {1,2,3,4} Right/LSB
     //     Index      0,1,2,3
     //
     // For example initialPermutation might hold {1,2,3,4} and we assume index 0
     // hold the 'most significant digit' and index 3 holds the 'least significant digit'
     T[] a = multiSet;
     int n = multiSet.Length;

     while (true)
     {
            // Take a shallow copy of the current permutation and yield return
            // it before we make any modifications
            T[] b = (T[])a.Clone();
            yield return b;

            // Find the value of index j such that we have visited all permutations of
            // a[0],a[1],...a[j] We obtain this by finding the highest index j such that
            // a[j] < a[j+1]
            //
            // Example: If we say that a={1,2,3} then we are finding the highest value of j
            // such that the value at position j is greater than the element at position
j+1 In
            // our case that occurs at j=1
            //
            // {1,2,3}
            //    |
            //    j
            var j = n - 2;
            while (j >= 0 && a[j].CompareTo(a[j + 1]) >= 0) j--;

            // If there is no such j then we are already on the
            // lexicographically highest and therefore the last
            // permutation. In this case we break out of the while loop
            // thereby terminating the method
            if (j == -1)
                   break;

            // If we have visited all permutations {a[0],...[aj]} then
            // the way to move to the next permutation lexicographically
            // is to swap a[j] with the smallest element greater than
            // a[j] whose index is greater than j. As the elements to
            // the right of a[j] are sorted in decreasing order from left
            // to right, the first element greater than a[j] when walking from
            // rightto left is the smallest value greater than a[j]
            var l = n - 1;
            while (a[j].CompareTo(a[l]) >= 0) l -= 1;

            // swap
            Swap(a, j, l);


            // At the moment, everything to the right of a[j] is sorted in
            // decreasing order As we have just increased a[j] we need to
            // reverse a[j+1]..a[n] so we have the next lexicographical element
            for (int lo = j + 1, hi = n - 1; lo < hi; lo++, hi--)
                   Swap(a, lo, hi);
     }
}
```

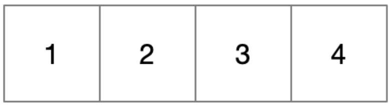
The permutations of a multi-set where e.g. {1,2,2,4} The number of permutation is

$$\frac{4!}{2!}$$

Permutation of Multiset algorithm

Iteration 1

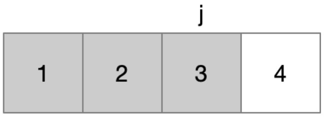
Visit Permutation



Initial permutation

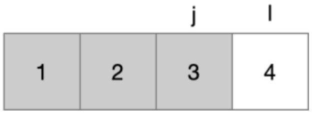
Find j

We want to find the smallest index j such that we have visited every permutation starting with a_0 ... a_j I We achieve this by setting $j = n - 1$ and decrementing j until $a_j < a_{j+1}$ Once this condition is met we know we have visited every permutation beginning with a_0 ... a_j In this specific case we have $j = 3$ and we have visited every permutation beginning with {1,2,3} namely the single permutation{1,2,3}{4}

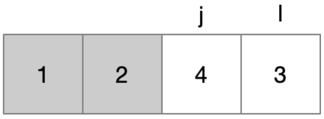


Increase a_j

We know from the previous step that we have visited ever permutation beginning with a_0 ... a_j. We want to find the smallest element greater than a_j that can legitemately follow a_0 ... a_{j-1}in a permutation. We achieve this by setting $l = n$ and then decreasing l until $a_j < a_l$ Because the tail is sorted in decreasing order we know $a_{j+1} \geq ... \geq a_n$ so the first element walking back from $a_n$that is greater than a_j is also the lowest possible value that is greater than a_j



Swap $a_j \leftrightarrow a_l$



Reverse

We know that everything after a_j is in decreasing order. But to be lexigographic we need it to be in increasing order so we reverse $a_{j+1}..a_n$. In this case we have a single element so no reversing is needed.

ITERATION 2

Visit Permutation

1	2	4	3

Find j

We want to find the smallest index j such that we have visited every permutation starting with $a_0 \dots a_j$ IWe achieve this by setting $j = n - 1$ and decrementing j until $a_j < a_{j+1}$

	j		
1	2	4	3

Once this condition is met we know we have visited every permutation beginning with $a_0 \dots a_j$ In this specific case we have $j = 2$ and we have visited every permutation beginning with $\{1,2\}$ namely the $\{1,2\}\{3,4\}$ and $\{1,2\}\{4,3\}$

Increase a_j

We know from the previous step we have visited all permutation beginning with $\{1,2\}$ so the key now it to increase $a_2$by the smallest amount possible. We know that $a_{j+1} \geq \dots \geq a_n$ in our case $\{4,3\}$ so the first element moving from highest to lowest index greater than $a_2$is also the smallest element greater than a_2

	j		l
1	2	4	3

Swap $a_j \leftrightarrow a_l$

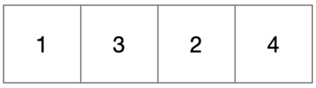
	j		l
1	3	4	2

Reverse

No we know everything after $\{1,3\}$is in decreasing order. As we have increased $a_2$we want to reverse everying after it so we end up with the next lexicographical element.

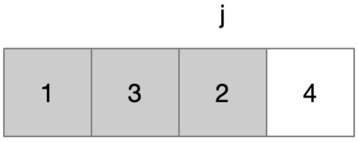
Iteration 3

Visit Permutation



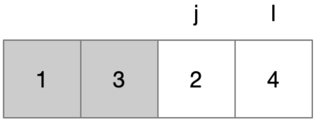
Find j

We want to find the smallest index j such that we have visited every permutation starting with a_0 ... a_j IWe achieve this by setting $j = n - 1$ and decrementing j until $a_j < a_{j+1}$ We have visited the single permutation starting {1,3,2} namely {1,3.2}{4}

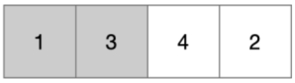


Increase a_j

We want to find the smallest value greater than a_j



Swap $a_j \leftrightarrow a_l$



Reverse

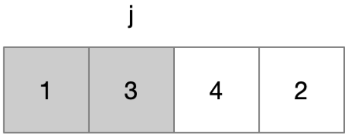
There is only a single element in position a_{j+1}so reversing does nothing

Iteration 4

Visit Permutation

1	3	4	2

Find j

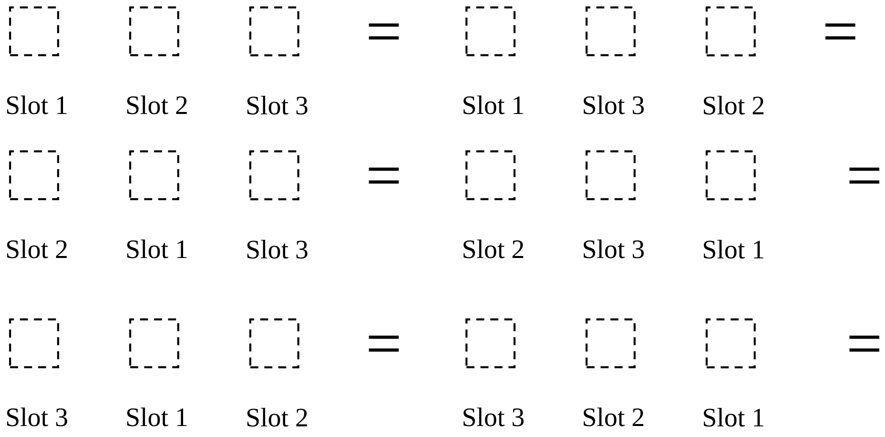


	j		
1	3	4	2

Increase a_j

Combinations

Unlike permutations, where order is important, with combinations we are only concerned that we selected something. One way to visualize this is that the order of slots is unimportant so if we have three slots then



Because we consider all permutations of the same things equal, then the numbers of combinations equals the number of permutations divided by the number of permutations of slots. We can formulate the problem generally as

- The number of ways of selecting r items from a total of n where order in unimportant

$$\binom{n}{r} = \frac{n!}{(n-r)!\,r!} \equiv {}^{n}C_{r} \equiv {}_{n}C_{r} \equiv C(n,r)$$

More formally a k-combination of a set S is a subset of k distinct elements of S. Every k combination has exactly k! permutations of elements. So we obtain the number of k-combinations by dividing the number of k-permutations by k!

$$\left(\binom{n}{r}\right)\left(\binom{n+r-1}{r}\right) =$$

The following code shows how we might generate combinations

```
public void GeneratePermutations(int[] set,
                                             int[] combination,
                                             int combinationIdx,
                                             int firstSetIdx,
                                             Action<int[]> visit)
{
    if (combinationIdx == combination.Length)
    {
           visit(combination);
           return;
    }

    for (int setIdx = firstSetIdx; setIdx < set.Length ; setIdx++)
    {
           combination[combinationIdx] = set[setIdx];

           GeneratePermutations(set, combination,combinationIdx + 1, setIdx + 1
                  ,visit);
    }
}
```

Properties of Combinatorial Coefficients

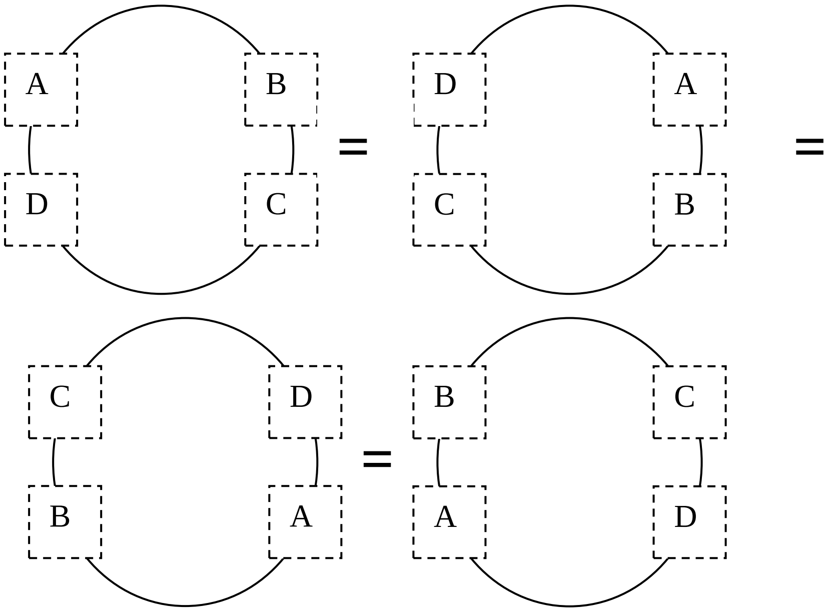
1. ${}^nC_n = {}^nC_0$ Because ${}^nC_n = \frac{n!}{(n-n)!} = \frac{n!}{0!n!} = 1$ and ${}^nC_0 = \frac{n!}{n!0!} = 1$

2. ${}^nC_{n-r} = {}^nC_r$ Because ${}^nC_{n-r} = \frac{n!}{[n-(n-r)]!(n-r)!} = \frac{n!}{(n-r)!r!} = {}^nC_r$

3. ${}^nC_r + {}^nC_{r+1} = {}^{n+1}C_{r+1}$ Because **Insert Proof**

Circular Permutations

With a circular permutation we consider two different permutations where each entry has the same element to its right/left as identical



From this we can see that the number of circular permutations of n items taken r at a time is given by

$$\frac{n!}{r(n-r)!}$$

Binomial Theorem

Overview – A recurrence relation for the co-efficient

Expressions of the form$(1+x)^n$, where n is a positive integer are known as binomial expressions. Multiplying out a binomial expansion gives us a binomial expansion.

$(1+x)^1 = 1+x$

$(1+x)^2 = 1+2x+x^2$

$(1+x)^3 = 1+3x+3x^2+x^3$

$(1+x)^4 = 1+4x+6x^2+x4^3+x^4$

Notice the emerging pattern! The co-efficient of x^2 in the expansion of $(1+x)^4$is equal to the sum of the co-efficient of x^2 and the co-efficient of x in the expansion of $(1+x)^3$. In general the co-efficient of x^min the expansion of $(1+x)^n$is equal to the sum of the co-efficient of $x^m$and x^{m-1}in the expansion of $(1+x)^{n-1}$. We can easily see why this is by looking at the following example

$(1+x)^5 = (1+x)(1+x)^4$

$1+4x+6x^2+4x^3+x^4$

$1+x$

$1+4x+6x^2+4x^3+x^4$

$x+4x^2+6x^3+4x^4+x^5$

$1+5x+10x^2+10x^3+5x^4+x^5$

A CLOSED FORM SOLUTION FOR BINOMIAL CO-EFFICIENTS

The coefficients of the binomial expansion $(1+x)^n$ are given by

$$1+{}_1^nCx^1+{}_2^nCx^2+...+{}_{n-1}^nCx^{n-1}+{}_n^nCx^n$$

It is worth spending a little time looking at why this might be. Consider the expansion of

(1.1). $(1+x)^3=(1_1+x_1)(1_2+x_2)(1_3+x_3)=$

(1.2) $1_1.1_2.1_2+1_1.1_2.x_3+1_1..x_2.1_1+1_1.x_2x_3+x_1.1_2.1_3+x_1.1_2.x_3+x_1x_2.1_3+x_1x_2x_3$

(1.3) $1+x_3+x_2+x_2x_3+x_1+x_1.x_3+x_1x_2+x_1x_2x_3=$

(1.4) $1+(x_3+x_2+x_1)+(x_1.x_3+x_1x_2+x_2x_3)+(x_1x_2x_3)=$

(1.5) $1+3x+3x^2+x^3$

Look at line (1.4) and notice that the number of ways of obtaining a unit power of x is the number of ways of selecting one item from the set (x_3,x_2,x_1) The ways of obtaining a square power of x is the number of ways of selecting two items from (x_3,x_2,x_1) The ways of obtaining a cube power of x is the number of ways of selecting three items from (x_3,x_2,x_1) And not forgetting the unit term, the number of ways of obtaining 1 is the number of ways of selecting zero x's from three things

Since the 1's have no effect the co-efficient of the x^r term is the number of combinations of n x's taken r at a time $+{}_r^nCx^r$

Extending the solution to (a+b)

Now we know how to obtain the coefficients of $(1+x)^n$ we can extend the methodology to the expansion of $(a+b)^n$ by noting that.

$$(a+b) = a\left(\frac{a+b}{a}\right).$$

Also

$$\left(\frac{a+b}{a}\right)^n = \frac{(a+b)^n}{a^n}$$

So

$$(a+b)^n = a^n\left[1+\frac{b}{a}\right]^n$$

Binomial expansion of E

Binomial expansion of E is given by

$$\left(1+\frac{1}{n}\right)^n = 1+n\left(\frac{1}{n}\right)+\frac{n(n-1)}{2!}\left(\frac{1}{n}\right)^2+\frac{n(n-1)(n-2)}{3!}\left(\frac{1}{n}\right)^2+,\dots\dots,+\frac{1}{n^n}$$

$$=1+\left(\frac{n}{n}\right)+\frac{n(n-1)}{2!n^2}+\frac{n(n-1)(n-2)}{3!n^4}+,\dots\dots,+\frac{1}{n^n}$$

$$=1+1+\frac{1-\frac{1}{n}}{2!}+\frac{\left(1-\frac{1}{n}\right)\left(1-\frac{2}{n}\right)}{3!}+,\dots\dots,+\frac{1}{n^n}$$

In the limit $\underset{n\to\infty}{Lim}\left(\frac{1}{n}\right)=0$ then the above expression tends to

$$=1+1+\frac{1-0}{2!}+\frac{(1-0)(1-0)}{3!}+,\dots\dots=\frac{1}{0!}+\frac{1}{1!}+\frac{1}{2!}+,\dots\dots=\sum_{r=0}^{\infty}\frac{1}{r!}$$

Questions – Combinatorics

Give an expression for the number of ways of taking n objects r at a time with repetition

$$n^r$$

Write code to provide all permutations of n things taken n at a time with repetition

Give an expression for the number of ways of taking n objects r at a time without repetition

$$\frac{n!}{(n-r)!}$$

Give a proof

$$n! = n \times (n-1) \times \ldots \times (n-r+1) \times \boldsymbol{(n-r)} \times \boldsymbol{(n-r-1)} \times\times \boldsymbol{2} \times \boldsymbol{1} \tag{\mathbf{1}}$$

$$(n-r) \times (n-r-1) \times (n-r-2) \times \ldots \times 2 \times 1 = (n-r)! \tag{2}$$

Substituting (2) into (1)

$$n! = n \times (n-1) \times (n-2) \times \ldots \times (n-r+1) \times (n-r)! \tag{3}$$

And re-arranging

$$n \times (n-1) \times (n-2) \times \ldots \times (n-r+1) = \frac{n!}{(n-r)!}$$