Algorithms

And data structures

Contents

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.

$$\Theta(g(n)) = \left\{ \begin{matrix} f(n): there\ exists\ positive\ constants\ c_1, c_2, n_0 such\ that \\ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) for\ all\ n\ \geq n_0 \end{matrix} \right\}$$

$g(n)$ 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++)                    c1n
	{
		T key = a[j];                                 c2(n − 1)
		int i = j - 1;                                c3(n − 1)


		while (i >= 0 && Less(key, a[i]))             c4 ∑_{j=1}^{j=n−1} tj
		{
			a[i + 1] = a[i];                      c5 ∑_{j=1}^{j=n−1}(tj − 1)
			i--;                                  c6 ∑_{j=1}^{j=n−1}(tj − 1)
		}

		a[i + 1] = key;                               c7(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?

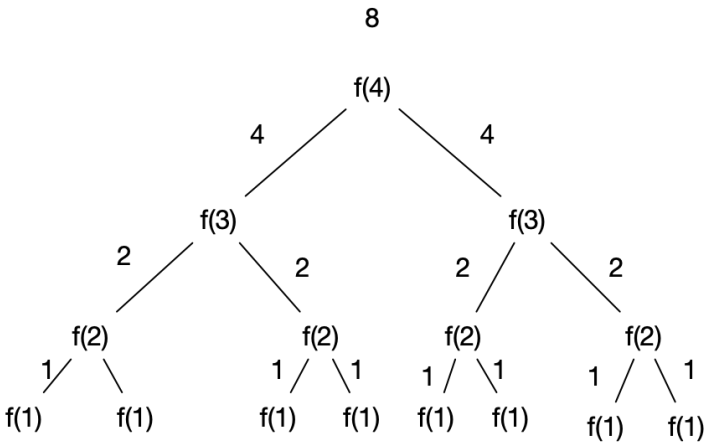
Because $\log_a x = \frac{\log_b x}{\log_b a} = \log_b x \frac{1}{\log_b a} =$

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);
}
```

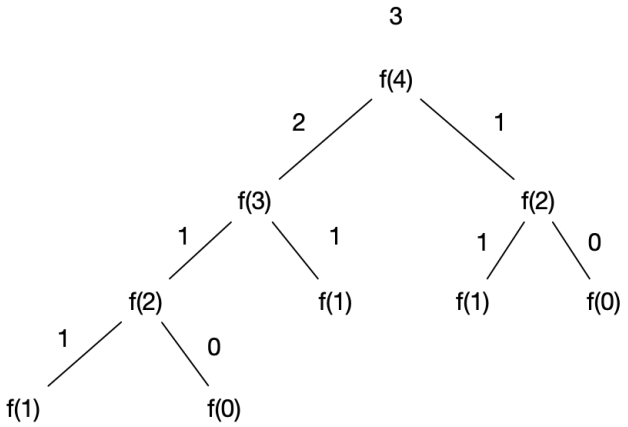
Look at the call graph of the specific case of Function(4). We get the total run time by summing the number of calls at each level. 1+2+4+8. As we move down from one level to the other each level has double the number of calls as the level before. In our case we have $1 + 2 + 4 + 8 = 2^0 + 2^1 + 2^2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



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 $O(x)$

Data Structures

Cheat Sheet

	Add	RemoveAt(int)
Array List	O(1)	O(N)
Row Header Two		

Symbol Tables

ArrayList

When to Use

When we perform many more reads than writes. When we need to frequently retrieve elements by index.

Description

In .NET, the List<T> class is backed by arrays. Appending is an efficient, constant time, O(1) operation as there is usually space on the end to add the element. Because the data is stored contiguously in memory, retrieval by index is also an efficient, constant time operation.

Inserting an element in the middle and deleting elements are slow, linear, O(N) operation as we need to move all elements of the array after the insertion/deletion point. If the array is sorted, then searching using a binary search is a logarithmic O(log(n)) operation.

Backing a list with arrays supports constant time appending and indexing. Inserting into the middle and deleting items are both inefficient O(N) operations.

```
public interface IArrayList<T>
{
    void Add(T element); ❶

    void RemoveAt(int index); ❷

    void AddRange(IEnumerable<T> elements); ❸

    void Insert(int index, T element); ❹

    T this[int index] { get; set; }

    long Count {get;}
}
```

❶ **Add**	O(1)
❷ **RemoveAt(int)**	O(N)
❸ **AddRange(IEnumerable<T>)**	O(M) where M is the number of elements being added
❹ **Insert(int, T)**	O(N)
❺ **this[int]**	O(1) constant time.
❻ **Count**	O(1) constant time as we keep track of the size as items are added and removed.

Linked List

When to Use

When we frequently need to insert elements into the middle of the list.

Description

In .NET LinkedList<T> is backed by a doubly linked list. Inserting and deleting elements is fast and efficient constant time O(1) operation if we have a reference to the node before or after it. If we need to find the insertion point we must perform a slow linear traversal (O(N)).

Priority Queue

API

Name	Example
Add	*Insert the element into the correct location in the queue*
Dequeue	*Remove and return the maximum/minimum element. Whether the element is the maximum or minimum is dependent on how the PriorityQueue is setup.*

HEAP (HEAP-ORDERED BINARY TREE)

Why Use

When we need to have keys in order but not full order. Typically, we just need to have the largest or smallest item at a time. A priority queue support two operations

- Read highest priority O(1)
- Remove highest priority O(1)
- Insert element O(1)

An efficient implementation uses a heap ordered binary tree that allows us to perform efficient logarithmic time removal and insertion. A binary tree is heap ordered if the key in each node is larger than or equal to the keys in that nodes two children. The largest key is found at the root. The time to add/remove is spent reheapifying.

Versus Binary search Tree

The binary search tree supports ordered traversal in O(N) but this is O(nlogn) for binary heap

Killer Property

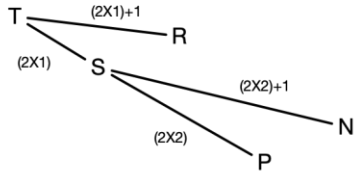
While inserting elements into a heap ordered binary tree has worst case performance of $O(log\ N)$, in the average case inserting random elements is $O(1)$. This is better than a binary search tree which has average case insert time of $O(log\ N)$. For proof of this result see below.

In cases where we do not need to keep all the elements in total order this feature makes the heap ordered binary tree preferable to a standard binary search tree.

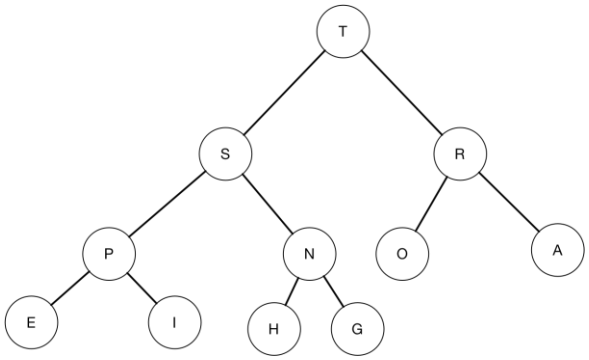
Overview Of Binary Heap

A binary heap is a collection that has the property that each node has a value that is larger/smaller than its two child nodes. The root node then has the highest/smallest. Whether the root has the largest or smallest depends on whether the heap is in maximum or minimum configuration. The children of a node with position k are in positions $2 \times k$ and$(2k + 1)$. Similarly, the parent of an element in position k is in position $\lfloor k/2 \rfloor$. The following shows an example

-	T	S	R	P	N	O	A	E	I	H	G
0	1	2	3	4	5	6	7	8	9	10	11



If we add or remove items or adjust priorities, we need to rehepify to restore heap order

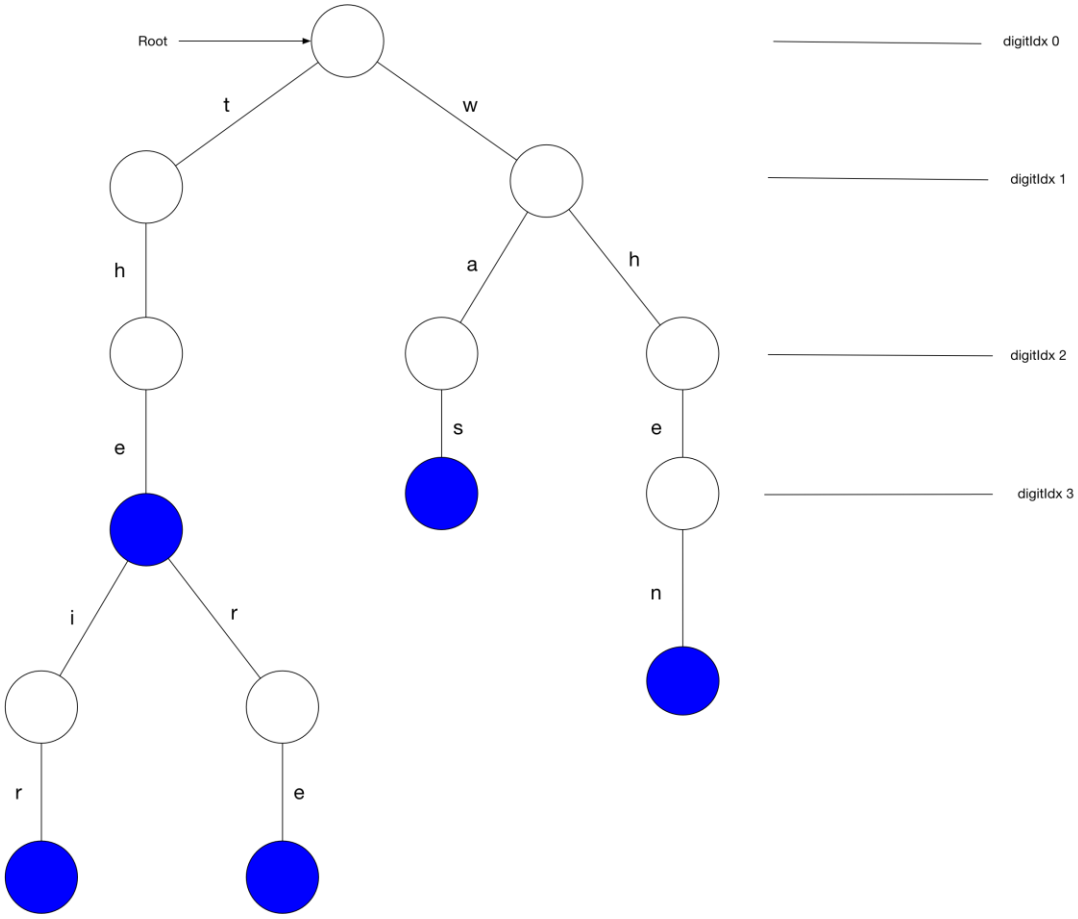


Proof that insert has average case performance of O(1)

$\frac{1}{2} \times 1 + \frac{1}{4} \times 2 + \cdots + \frac{1}{2^N} \times N = 2?????,$

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

Bag

Queue

Stack

Hash Table

Binary Search Tree

Questions – Data Structures

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

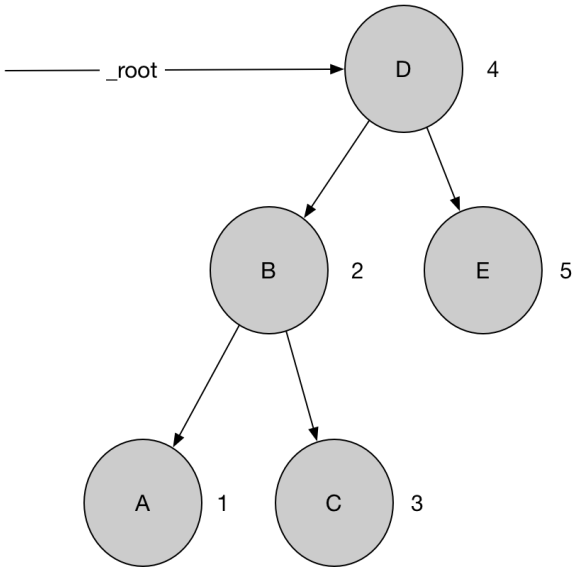
Find maximum/minimum key

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



Given a binary tree write code to return the number of nodes on each level

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

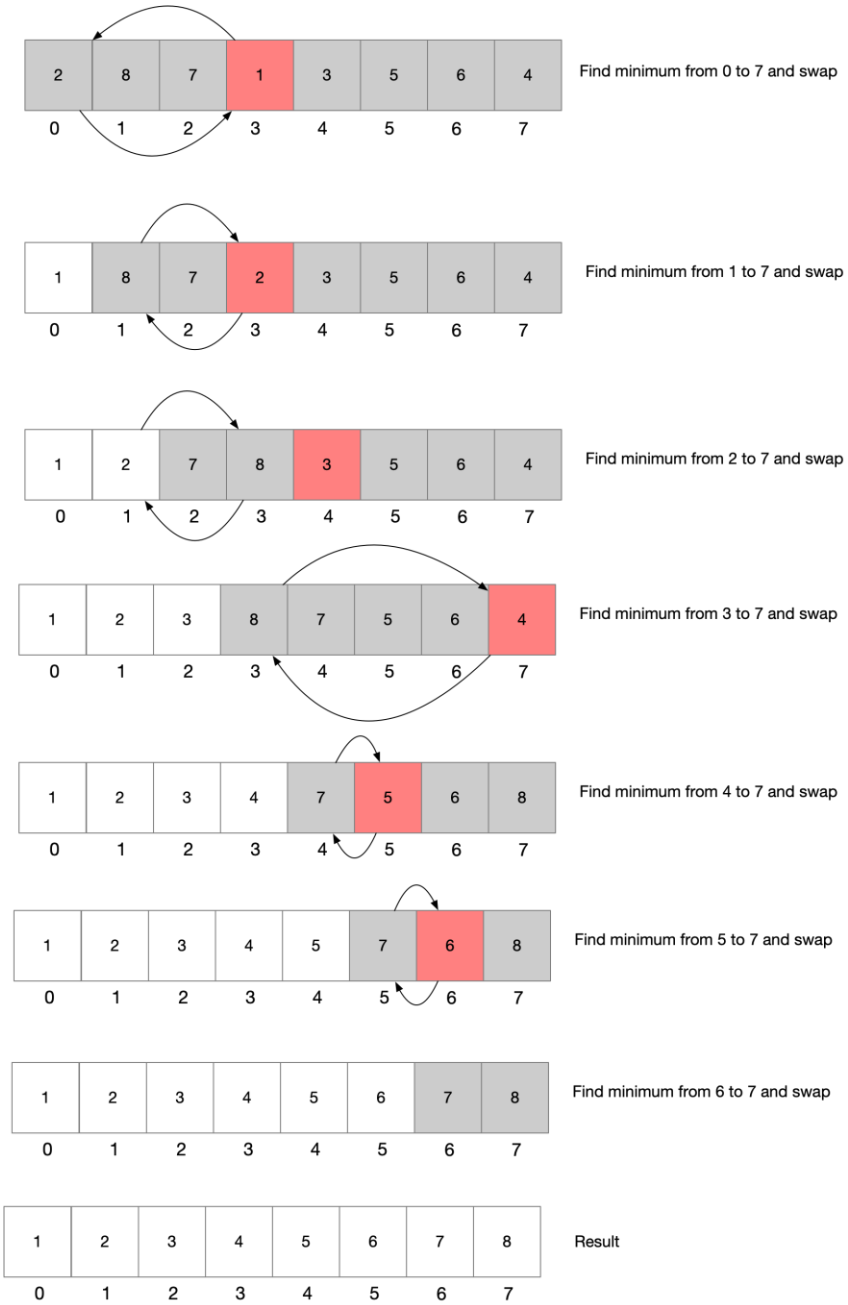
> **STABLE SORT**
>
> A stable sort keeps elements with the same key in the same order relative to each other in the sorted output

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;
}
```

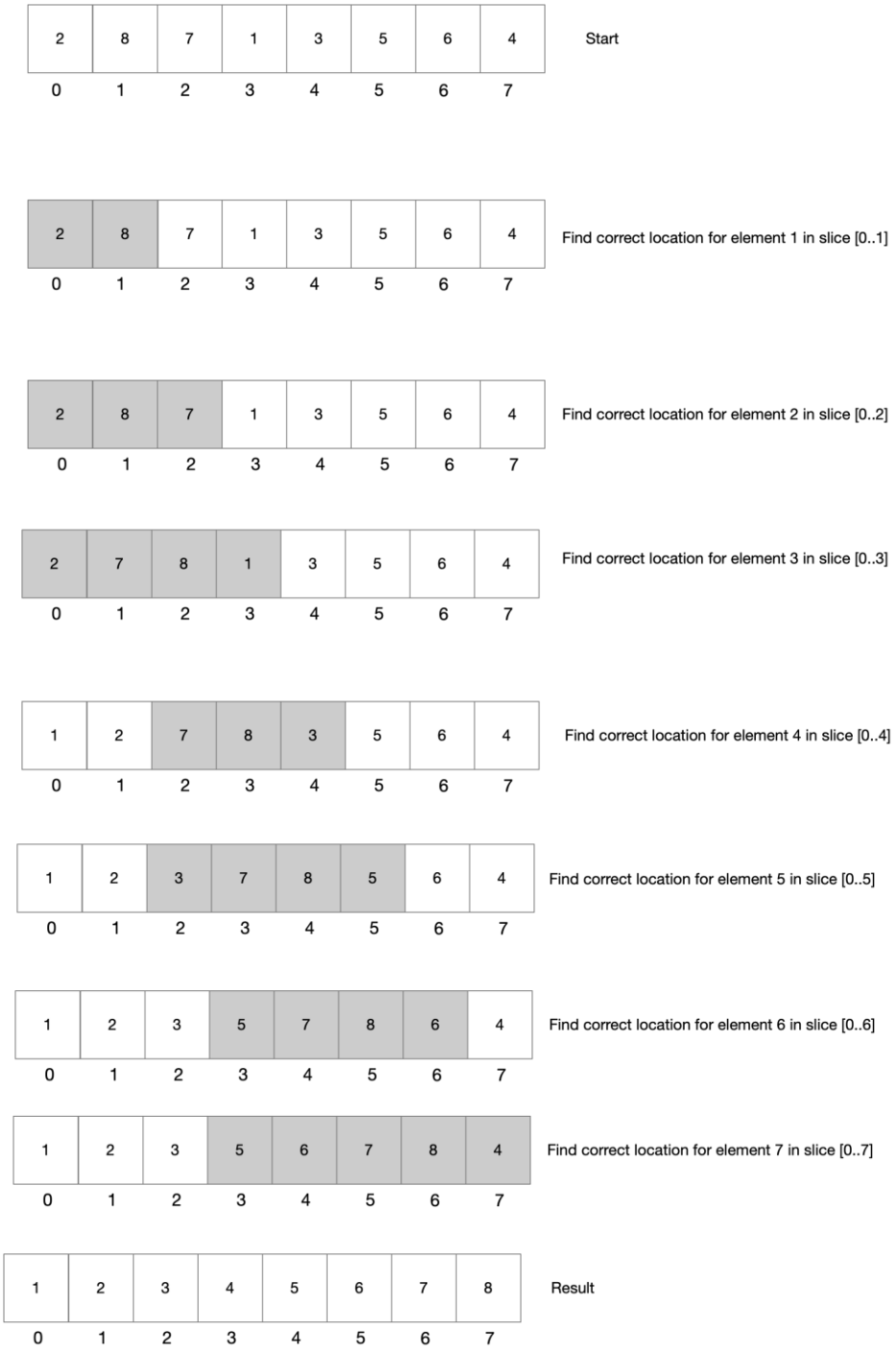
The performance is easily analysed by looking at the two loops. The outer loop runs from 0 to (n-1) giving n iterations. Each iteration of the inner loop runs from i+1 to n-1 giving us

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

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



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;                    c1

	for (int i = 1; i < a.Length; i++)                         c2(n)
	{
		for (int j = i; j > 0 && Less(a[j], a[j - 1]); j--)    c3 ∑_{j=1}^{j=n-1} tj
		{
			Switch(a, j - 1, j);                               c4 ∑_{j=1}^{j=n-1} tj

		}
	}
}
```

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

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

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

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

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

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

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

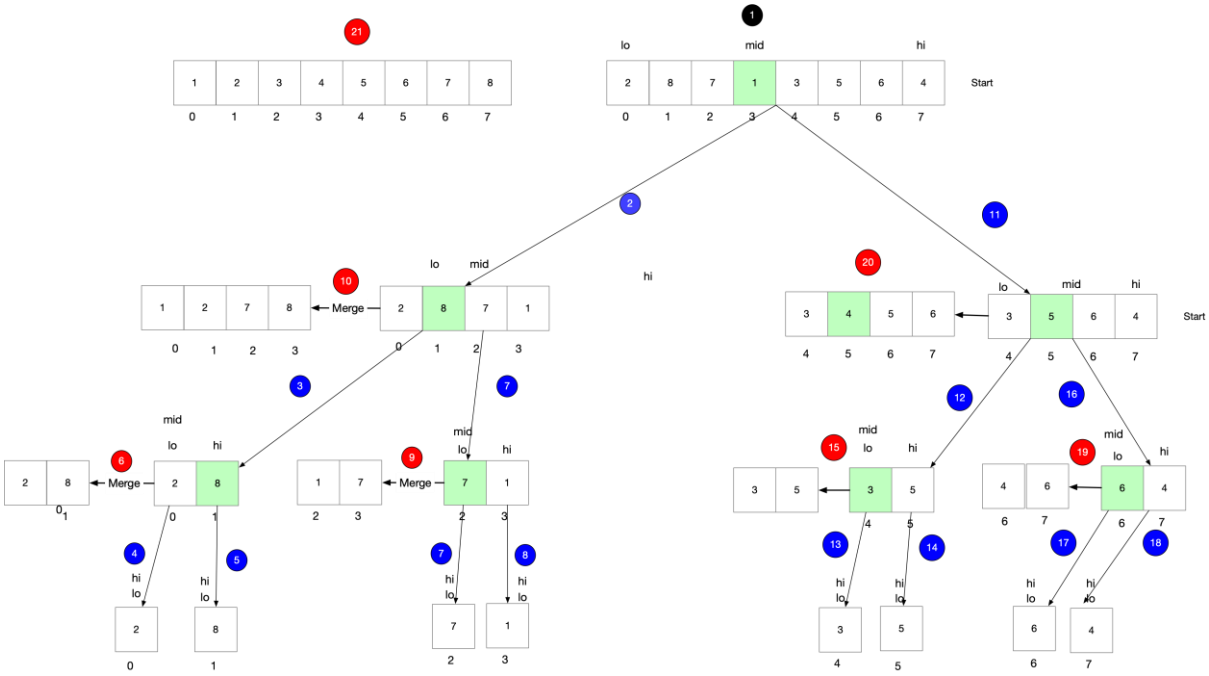
Proof by induction

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

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

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

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*

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

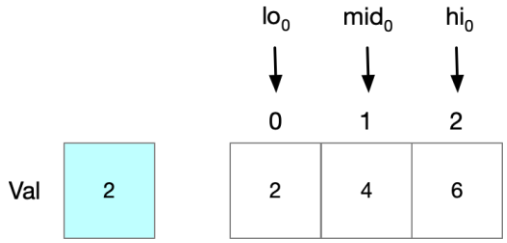
Searching

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

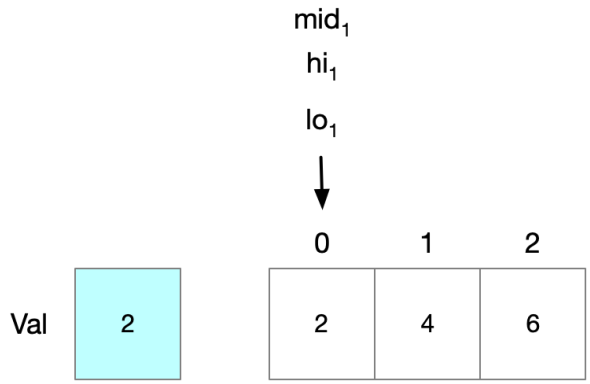
$$mid_0 = lo_0 + \frac{(hi_0 - lo)}{2}$$

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;
}
```

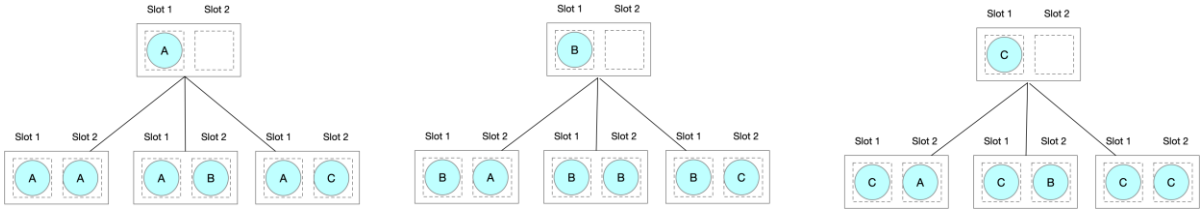
The runtime is O(logN)

Combinatorics

K-Strings (n^k)

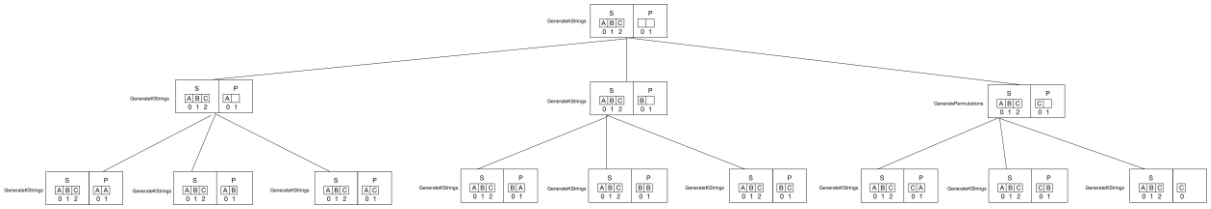
DESCRIPTION

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 .Order is important so we can think of k-strings as permutations with repetition. The following code shows how we can generate the k-strings very simply using recursion.

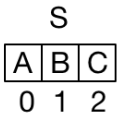
ALGORITHM ONE.

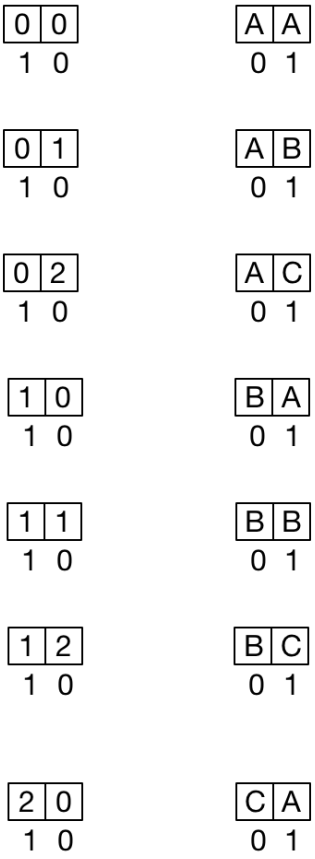


```
public IEnumerable<List<T>> GenerateKStrings<T>( List<T> S, int k)
{
    var kStrings = new List<List<T>>();
    GenerateKStrings(S, new List<T>(new T[k]),0, k => kStrings.Add(k));
    return kStrings;
}

public static void GenerateKStrings<T>(List<T> kString, List<T> S, int
stringIdx, Action<List<T>> visit)
{
    if (stringIdx == kString.Count)
    {
          visit(kString);
          return;
    }

    for (int setIdx = 0; setIdx < S.Count; setIdx++)
    {
          var clone = new List<T>(kString);
          clone[stringIdx] = S[setIdx];
          GenerateKStrings(kString, S, stringIdx + 1, visit);
    }
}
```

ALGORITHM TWO

This algorithm is based on incrementing integers.





```
public static IEnumerable<T[]> GenerateKStrings<T>( T[] S,int kStringLength)
{
    // Holds a k-digit number where each digit is of a base equal to
    // the number of elements in the set S. So if there are two
    // character in S,the digits in this number are binary.
    //
    // Each digit forms a index into S telling us exactly which
    // element of the 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[kStringLength];

    while (true)
    {
        // Generate the current k-string by using the elements
        // of seqIndices to index into S.
        T[] kString = new T[kStringLength];
        for (int i = 0; i < kStringLength; i++)
            kString[i] = S[seqIndices[i]];

        // Return the k-string.
        yield return kString;

        // In this algorithm we treat  the indices array as an
        // n-digit number where the base of each digit is determined by the
        // number of elements in S.
        // 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 < kStringLength && 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 == kStringLength)
            break;

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

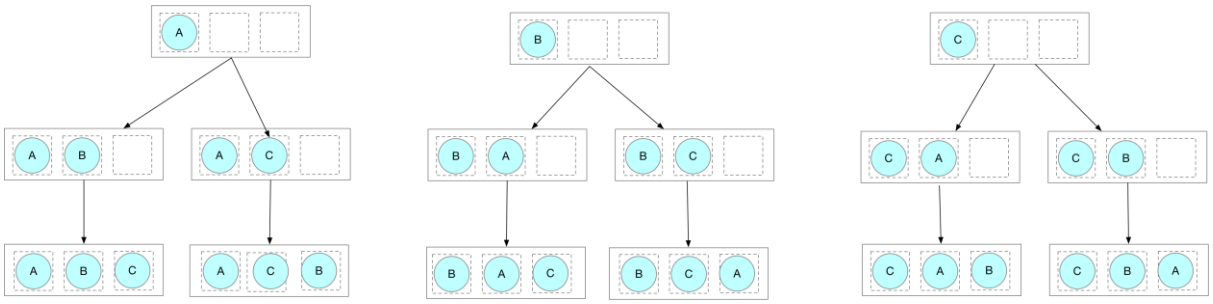
The runtime of this algorithm is clearly n^k

Ordinary Permutations ($n!$)

DESCRIPTION

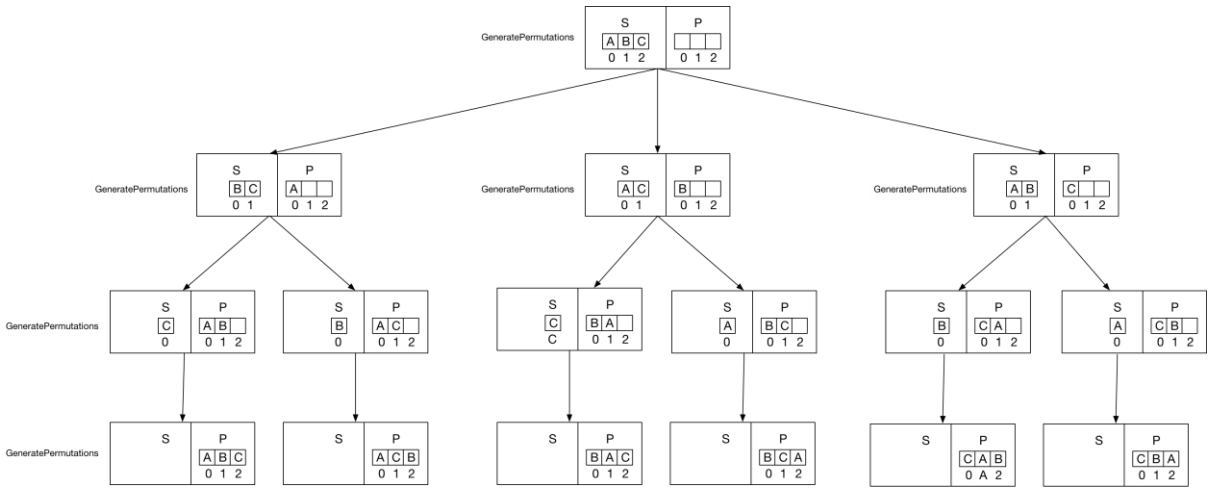
From an Introduction to Algorithms

"A permutation of a finite set S is an ordered sequence of the elements of S, with each element appearing exactly once. " With permutations duplicates are not permitted. There are $n!$ permutations of a set of n elements.



ALGORITHM ONE – SIMPLE RECURSIVE

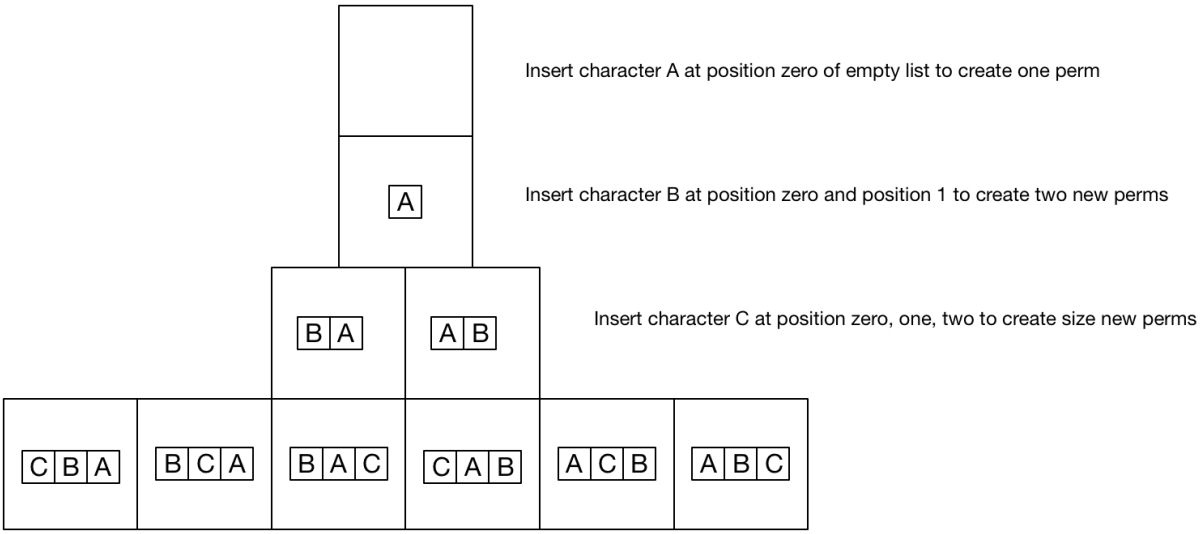
We are using a simple recursive algorithm. At each level we remove the elements of the set used already. We can clearly see the factorial nature




```
public IEnumerable<List<T>> GeneratePermutations<T>(List<T> S) where T :
IComparable<T>
{
    var permutations = new List<List<T>>();
    GeneratePermutations(S,new List<T>(new T[S.Count()]),0,
          permutation => permutations.Add(permutation));
    return permutations;
}

public void GeneratePermutations<T>(List<T> S, List<T> P, int idx,
    Action<List<T>> f) where T : IComparable<T>
{
    if(idx == P.Count())
          f(P);

    foreach (var element in S)
    {
          var permutation = new List<T>(P);
          permutation[idx] = element;

          var clonedSet = new List<T>(S);
          clonedSet.Remove(element);

          GeneratePermutations(clonedSet,permutation,idx+1,f);
    }
}
```

ALGORITHM TWO – ITERATIVE BASED ON INSERTS BETWEEN EXISTING ELEMENTS

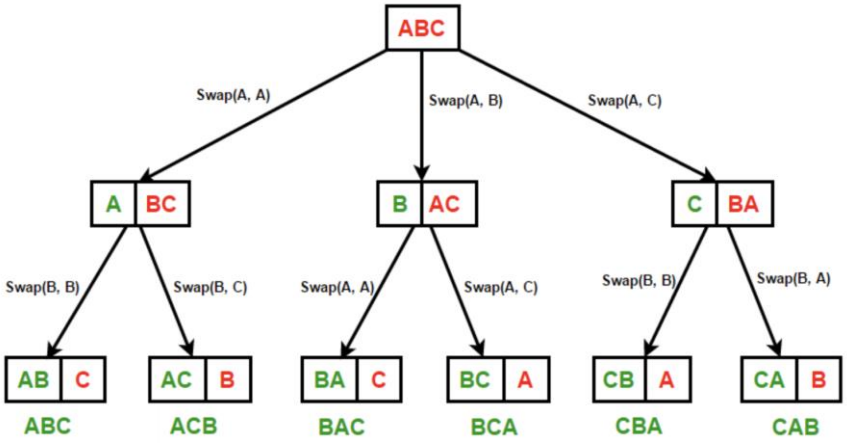
The idea behind this simple iterative algorithm is that we start with the empty set and add each character to every position until we get the desired length.




```
public IEnumerable<List<T>> GeneratePermutations<T>(List<T> S) where T :
IComparable<T>
{
    // These are the temporaty lists we use to build up the permutations.
    // We seed it with the empty list
    var temporaryPermutations = new List<List<T>>() { new List<T>()};

    // The actual results. Al
    var results = new List<List<T>>();

    for (int setIdx = 0; setIdx < S.Count; setIdx++)
    {
        // The number of temporary elements at this iteration
        int temporaryElementCount = temporaryPermutations.Count;

        // The set element to be added to every tempory elemtn in every pos.
        T setElement = S[setIdx];

        for (int tempListIdx = 0; tempListIdx < temporaryElementCount;
            tempListIdx++)
        {
            var sourcePerm = temporaryPermutations[tempListIdx];

            for (int insertIdx = 0; insertIdx <=
sourcePerm.Count();insertIdx++)
            {
                var newPerm = new List<T>(sourcePerm);
                newPerm.Insert(insertIdx,setElement);

                if (newPerm.Count == S.Count())
                {
                    results.Add(newPerm);
                }

                temporaryPermutations.Add(newPerm);
            }
        }
    }

    return results;
}
```

ALGORITHM THREE - SWAPS

Another recursive algorithm based on swaps is given as




```
public IEnumerable<List<T>> GeneratePermutations<T>(List<T> S) where T :
IComparable<T>
{
    var permutations = new List<List<T>>();
    GeneratePermutations(S, 0,permutation => permutations.Add(new
List<T>(permutation)));
    return permutations;
}

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

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

}

private static void Swap<T>(List<T> arr, int idx1, int idx2)
{
    T temp = arr[idx1];
    arr[idx1] = arr[idx2];
    arr[idx2] = temp;
}
```

Algorithm Four – Heaps Algorithm

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);
	}
}
```

ALGORITHM FIVE – KNUTH

```
public IEnumerable<List<T>> GeneratePermutations<T>(List<T> initialPermutation) 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'
      List<T> a = initialPermutation;
      int n = initialPermutation.Count;

      while (true)
      {
            // Take a shallow copy of the current permutation and yield return
            // it before we make any modifications
            List<T> b = new List<T>(a);
            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 right
            // to 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);
      }
}
```

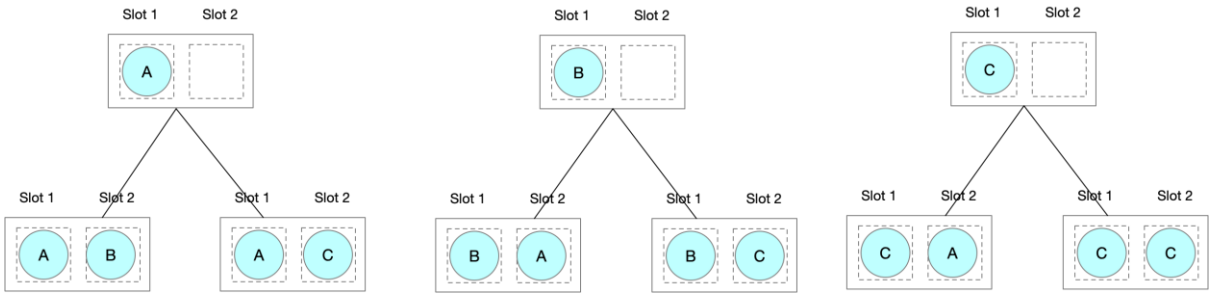
```
private void Swap<T>(List<T> arr, int idx1, int idx2)
{
     T temp = arr[idx1];
     arr[idx1] = arr[idx2];
     arr[idx2] = temp;
}
```

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

From an Introduction to Algorithms

"A k-permutation of a set 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)"

So, we can create a 2-permutation of a set of 3 elements as follows.



We have three choices for the first slot. But for each of those choices we only have two choices for the second slot as one value from the set has 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 objects 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

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

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

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

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

ALGORITHM ONE.

The first algorithm mis recursive. We call the GeneratePermutations with an empty T[] of size k. This algorithm will work for all k-permutations including the special permutation case where k is equal to the size of the set S.

```
public void GenerateKPermutations<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);
            GenerateKPermutations(clonedSet,permutation,permIdx+1,visitFunc);
     }
}
```

Permutation of Multi-Sets

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

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

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

ALGORITHM ONE.

ALGORITHM TWO.

```
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);
      }
}
```

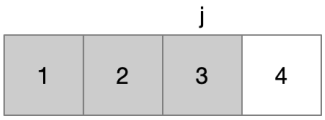
ITERATION 1

Visit Permutation

Initial permutation

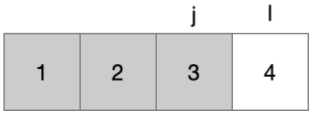
Find j

We want to find the smallest index j such that we have visited every permutation starting with $a_0 \ldots a_j$ 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 $a_0 \ldots a_j$ 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}

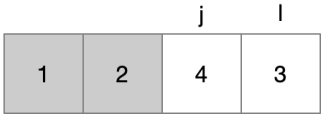


Increase a_j

We know from the previous step that we have visited ever permutation beginning with $a_0 \ldots a_j$. We want to find the smallest element greater than a_j that can legitemately follow $a_0 \ldots a_{j-1}$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 $a_{j+1} \geq \ldots \geq a_n$ 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.

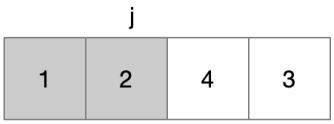
ITERATION 2

Visit Permutation

1	2	4	3

Find j

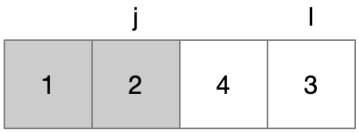
We want to find the smallest index j such that we have visited every permutation starting with $a_0 \dots a_j$ IWe achieve this by setting $j = n - 1$ and decrementing j until $a_j < a_{j+1}$



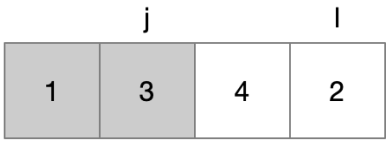
Once this condition is met we know we have visited every permutation beginning with $a_0 \dots a_j$ In this specific case we have $j = 2$ and we have visited every permutation beginning with $\{1,2\}$ namely the $\{1,2\}\{3,4\}$ and $\{1,2\}\{4,3\}$

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



Swap $a_j \leftrightarrow a_l$



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

1	3	2	4

Find j

We want to find the smallest index j such that we have visited every permutation starting with $a_0 \ldots a_j$ 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}

		j	
1	3	2	4

Increase a_j

We want to find the smallest value greater than a_j

		j	l
1	3	2	4

Swap $a_j \leftrightarrow a_l$

1	3	4	2

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

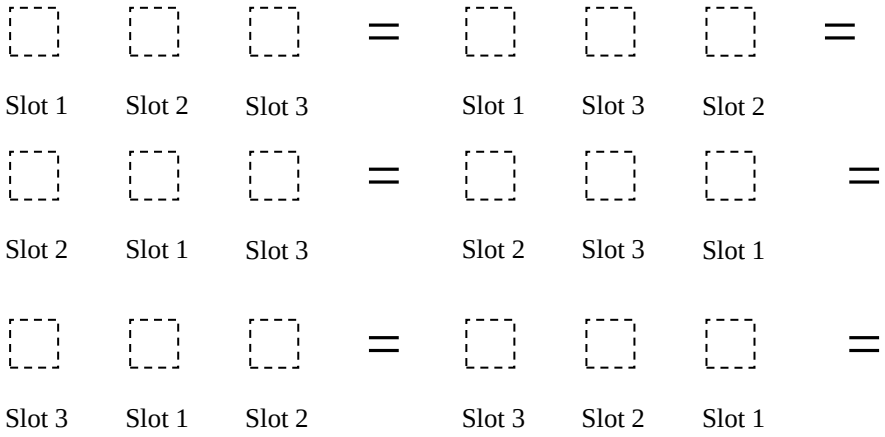
1	3	4	2

Increase a_j

K-Permutation of Multi-Sets

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);
     }
}
```

Subsets

Consider the situation where we want to generate all subsets of a given set. Say the set is $S = \{A, B, C\}$ The approach we take is to consider all 3-strings of the set $\{0,1\}$ where each index specifies whether that element of the set is present in the subset.

$\{0,0,0\} = [\quad]$

$\{0,0,1\} = [C]$

$\{0,1,0\} = [B]$

$\{0,1,1\} = [B, C]$

.

.

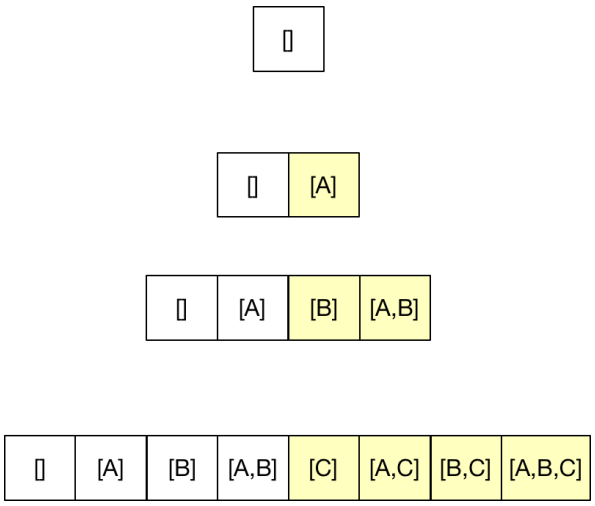
$\{1,1,1\} = [A, B, C]$

ALGORITHM ONE.

```
public static void GenerateSubstrings<T>(T[] set)
{
    var flags = new bool[set.Length];

    GenerateKStrings(flags, new bool[] {false,true},0, (perm) =>
    {
          var subString = new List<T>();

          for (int i = 0; i < perm.Length; i++)
          {
                if (perm[i]) subString.Add(set[i]);
          }

          System.Console.WriteLine(subString);
    });
}


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

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

ALGORITHM TWO.

This algorithm works by noticing we can generate all subsets by starting with the empty set and then adding element by element as follows



The code is then as follows

```
public List<T[]> GenerateSubsets<T>(T[] set)
{
     List<T[]> subsets = new List<T[]>();

     // Seed the subsets collection with the empty set
     subsets.Add(new T[0]);

     for (int setIdx = 0; setIdx < set.Length; setIdx++)
     {
            // We only want to walk the elements that existed in the
            // subsets collection before this iteration. For this reason
            // we need to store the count and not check it on each iteration
            // of the inner loop (where we are adding to the list).
            int currentSubsetCount = subsets.Count;

            for (int subSetIdx = 0; subSetIdx < currentSubsetCount; subSetIdx++)
            {
                  // Copy the subset at index subSetIdx and add the
                  // a new element
                  var sourceSubset = subsets[subSetIdx];
                  var newSubset = new T[sourceSubset.Length+1];
                  Array.Copy(sourceSubset,newSubset,sourceSubset.Length);
                  newSubset[sourceSubset.Length] = set[setIdx];

                  // Add the new subset to the subsets collection
                  subsets.Add(newSubset);
            }
     }

     return subsets;
}
```

Properties of Combinatorial Coefficients

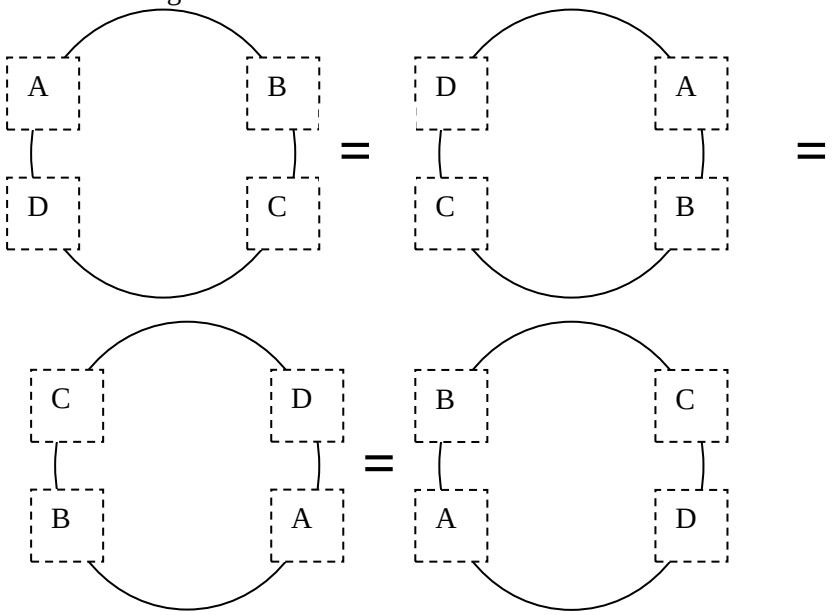
1. ${}^nC_n = {}^nC_0$ Because ${}^nC_n = \frac{n!}{(n-n)!} = \frac{n!}{0!n!} = 1$ and ${}^nC_0 = \frac{n!}{n!0!} = 1$

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

$1+{}^n_1Cx^1+{}^n_2Cx^2+\ldots+{}^n_{n-1}Cx^{n-1}+{}^n_nCx^n$

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 $+{}^n_r Cx^r$

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

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

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

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

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

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

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

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

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

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

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

Dynamic Programming Problems

Whereas divide and conquer can be used with disjoint subproblems, dynamic programming applies when the subproblems overlap. When there are overlapping subproblems a naive divide and conquer implementation inefficiently solves identical subproblems multiple times. A dynamic programming approach prevents this by storing the subproblem results in a table.

Dynamic programming is often used to solve optimisation problems. In an optimisation problem we assign values to each possible solution of a problem. We then aim to find an optimal solution among all the possible solutions. The optimal solution is the solution with the maximum or minimum value. In order to specify a dynamic programming solution we need to

- Specify what an optimal solution looks like
- Define the value recursively
- Calculate the value. Usually we use a bottom up approach
- Construct the optimal solution from the calculated information

The distinction between steps 3 and 4 is subtle. Step 3 gives us the value of the optimal solution. We only need to do step 4 if, in addition to the value, we also need the optimal solution itself.

0/1 Knapsack

Unbounded Knapsack

In order to use dynamic programming, the problem under consider must have

1. Optimal substructure
2. Overlapping subproblems

Optimal substructure means a problems solution contains within it solutions to subproblems. Overlapping subproblems occur when the same subproblem occurs over and over. A problem would not have overlapping subproblems when each step of the recursion generates new problems.

Questions – Dynamic Programming Problems

MAX SUB-ARRAY PROBLEM

Consider where we want to find the maximum subarray of an array as follows.

```
int[] array = new int[]{ 3, -1, -1, 10, -3, -2, 4 };
```

Brute force

We can use a brute force algorithm as follows. This technique creates all combinations of two array indices i and j. When we add in the inner loop that sums between the two indices, we have an algorithm that is very clearly cubic $O(n^3)$

```
public int MaxSubarraySum(int[] array)
{
    int max = int.MinValue;

    for (int i = 0; i < array.Length; i++)
    {
          for (int j = i+1; j < array.Length; j++)
          {
                int sum = 0;
                for (int idx = i; idx <=j; idx++) sum += array[idx];

                if (sum > max)
                      max = sum;
          }
    }
     return max;
}
```

A quick glance at this code shows the running time is $O(n^3)$. Can we do any better? One way to improve the algorithm is to use the divide and conquer technique.

Divide and Conquer

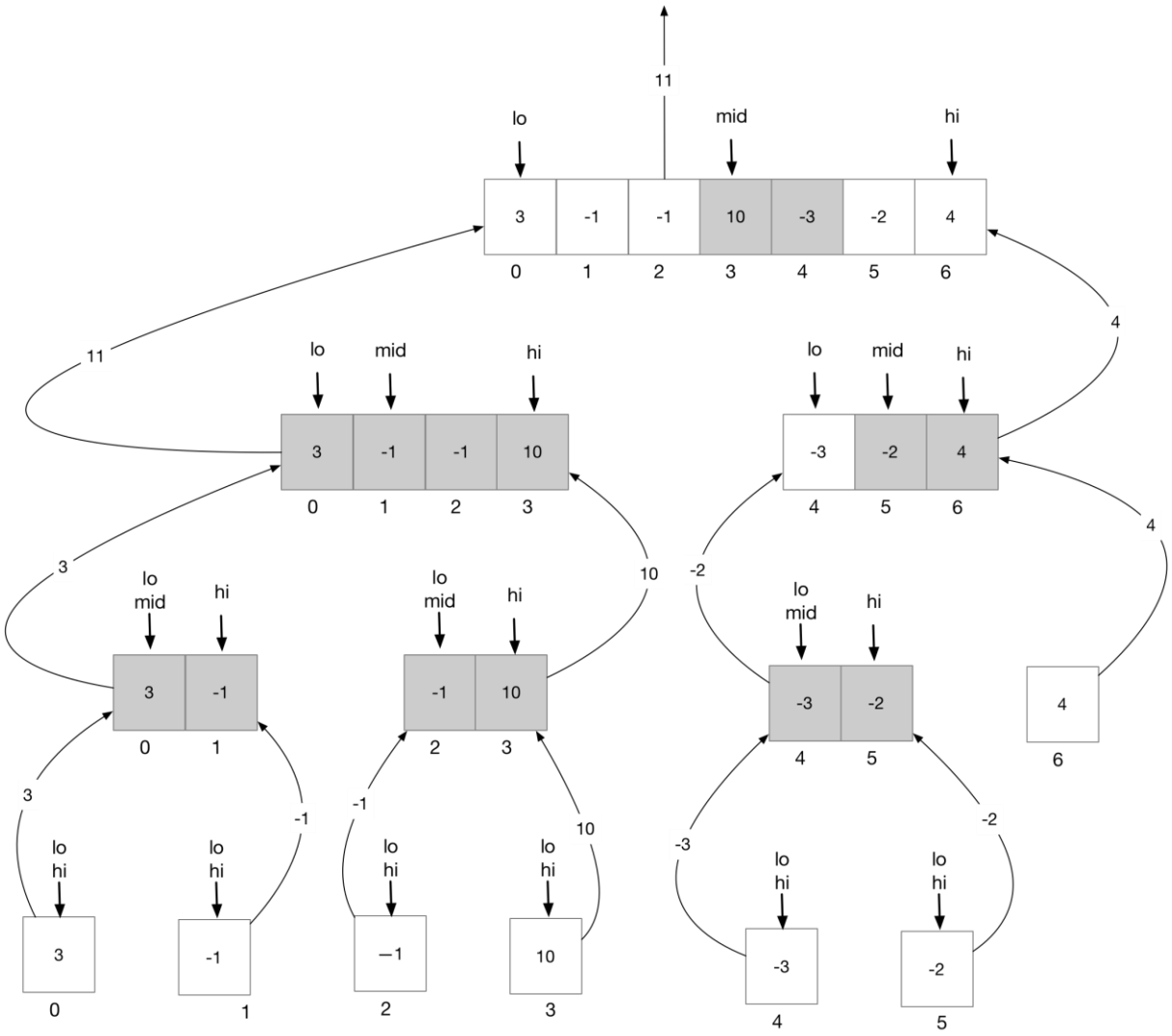
We use a recursive algorithm to break the problem down into half at each stage. Consider the input as follows.

```
int[] array = new int[]{ 3, -1, -1, 10, -3, -2, 4 };
```

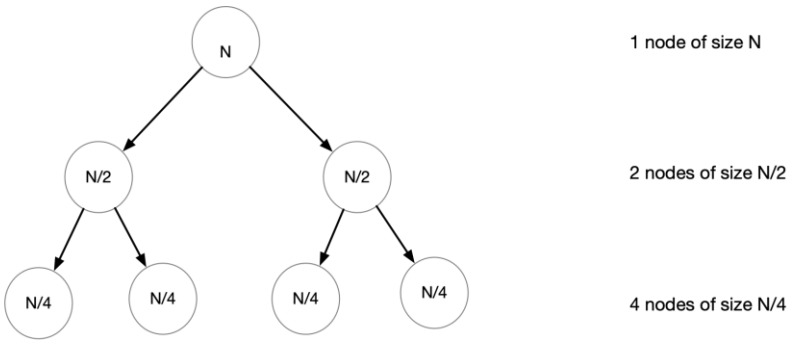
At each stage we divide the array in two and recursively calculate the maximum subarray in the left and right half. We also calculate the maximum subarray that crosses the middle line

```
public int MaxSubarraySum(int[] arr, int lo, int hi)
{
    // The recursion bottoms out here
    if (lo == hi) return arr[hi];

    // Divide
    int mid = (lo + hi ) / 2;

    // Conquer
    int maxLeft = MaxSubarraySum(arr,lo,mid);
    int maxRight = MaxSubarraySum(arr,mid+1,hi);

    // Combine
    int maxCrossing = CalcMaxCrossingArray(arr,lo,mid,hi);
    return Math.Max(Math.Max(maxLeft,maxRight),maxCrossing);
}

public int CalcMaxCrossingArray(int[] arr, int lo,int mid, int hi)
{
    // Calculate the largest array fragment that
    // in the sub-array arr[lo..mid] that also includes
    // arr[mid]
    int leftSum = int.MinValue;
    int sum = 0;
    for (int i = mid; i >= lo; i--)
    {
          sum += arr[i];
          if (sum > leftSum) leftSum = sum;
    }

    // Calculate the largest array fragment that
    // in the sub-array arr[mid+1..hi] that also includes
    // arr[mid+1]
    int rightSum = int.MinValue;
    sum = 0;
    for (int j = mid + 1; j <= hi; j++)
    {
          sum += arr[j];
          if (sum > rightSum) rightSum = sum;
    }

    // The largest fragment that crosses the midpoint
    // is then the sum of left and right
    int maxCrossing = leftSum + rightSum;
    return maxCrossing;
}
```

Graphically we have as follows.



The following recursion diagram helps us analyse the runtime.



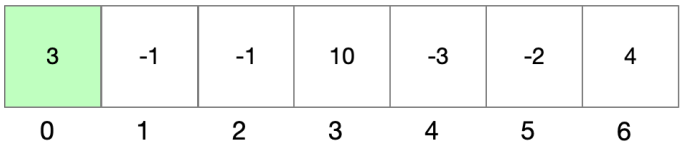
If we assume N is a power of 2 then we see that we have $\log_2 n$ levels. At each level l there are 2^lnodes each of size $\frac{N}{2^l}$ so each level in total carries out $2^l \frac{N}{2^l}$ = N comparisons. To the total work over the whole recursion tree is $N \log_2 N$

Dynamic Programming

There is a very clever way we can convert our original cubic approach to a linear approach. Consider the following array. First we calculate the maximum subarray that ends at index 0 This is obviously just the single value 3. Similarly the maximum subarray of any array in A[0..0] is also just 3 so currentSum and bestSum are the same

currentSum=3

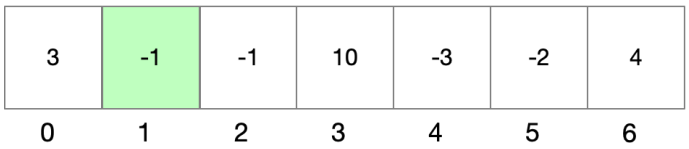
bestSum=3



Now we want to find the best subarray that ends at index 1. There are two choices. Either the best subarray that ends at index 0 + A[1] or a single element subarray consisting of A[1]. The answer is of course Max[0..0] + A[1]. This is worse that the bestSum so far seem so we don't update the best

currentSum=2

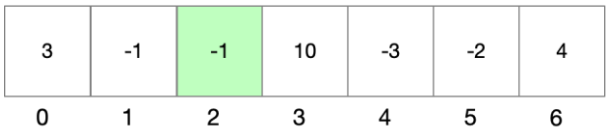
bestSum=3



Now we want to find the best subarray that ends at index 2. There are two choices. Either the best subarray that ends at index 1 + A[2] or a single element subarray consisting of A[2].

currentSum=1

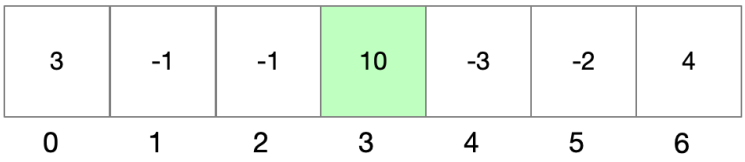
bestSum=3



Now we want to find the best subarray that ends at index 3. There are two choices. Either the best subarray that ends at index 2 + A[3] or a single element subarray consisting of A[3]. The sum is best so we choose the sum. Also this better than the best sum so we update that too

currentSum=11

bestSum=11



The code is as follows

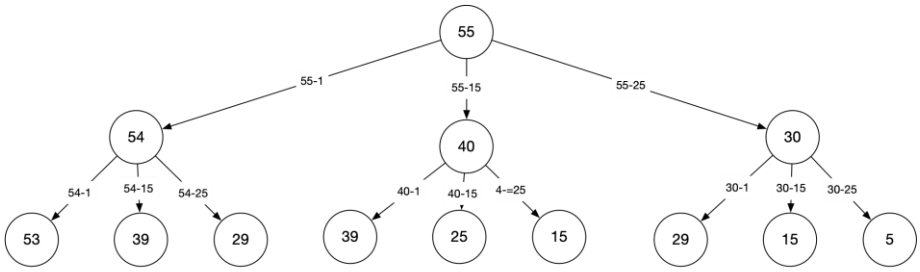
```
public int MaxSubarraySum(int[] array)
{
    var sums = new int[array.Length];

    // The current sum is the best sum
    // of any subarray that ends at
    // ends at index j
    int currentSum = array[0];

    // The sum of the largest subarray
    // anywhere in array[0..j] This is
    // obviously the best value seen so
    // far of all values of currentSum
    int bestSum = array[0];

    for (int i = 1; i < array.Length; i++)
    {
          // What is the best sum ending at index i ? It
          // is one of two things.
          //  1) The best sum ending at i-1 + array[i]
          //  2) A single element array[i]
          currentSum = Math.Max( currentSum + array[i],array[i]);

          // If the best sum ending at i is better than the best
          // sum seen anywhere in arr[0..i-1] we update the best sum
          bestSum = Math.Max(currentSum,bestSum);
    }

    return bestSum;
}
```

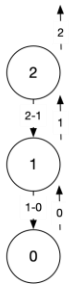
MINIMUM COINS

You are given an array of integers representing coins and a target monetary amount calculate the minimum number of coins that can be used to generate that amount. For example, given the array {1,15,35} and the amount 55 you would return 7 coins. This is because we can solve the problem with one 35p coin, one 15p coins and five 1p coins.

Greedy Algorithm

We can formulate a greedy algorithm as follows.






```
public int MinCoins(int[] coins, int amount)
{
    // Terminating case that causes
    // recursion to bottom out.
    if (amount == 0) return 0;

    int minCoins = int.MaxValue;

    // Assume we have a one penny coin so we also have a solution
    for (int i = 0; i < coins.Length; i++)
    {
          int newAmount = amount - coins[i];
          if (newAmount >=0)
          {
                minCoins =
                      Math.Min(minCoins, MinCoins(coins, newAmount));
          }

    }

    return minCoins +1;
}
```

Memoization

We can massively improve the performance of our top down greedy algorithm by adding memoization.

```
public int MinCoins(int[] coins, int amount, IDictionary<int,int> memo)
```

```
{
    // Terminating case that causes
    // recursion to bottom out.
    if (amount == 0) return 0;

    int minCoins = int.MaxValue;
    // Assume we have a one penny coin so we also have a solution
    for (int i = 0; i < coins.Length; i++)
    {
          int newAmount = amount - coins[i];

          if (newAmount >=0)
          {
                int requiredCoins;
                if (!memo.TryGetValue(newAmount,out requiredCoins))
                {
                      memo[newAmount] =  MinCoins(coins, newAmount,memo);
                }

                minCoins = Math.Min(minCoins,requiredCoins);
          }
    }

    return minCoins +1;
}
```

On the input new int[] { 1,15, 35},55 we reduce the number of calls from ~5600 to 56

Dynamic Programming

We can solve the same problem using bottom up dynamic programming as follows

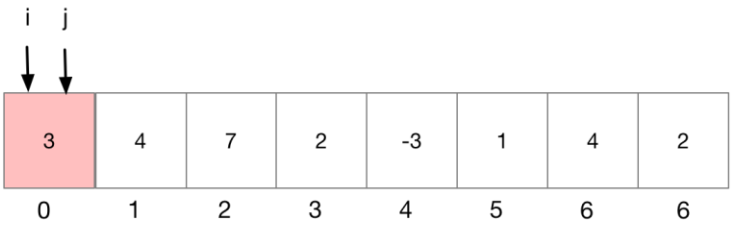
The following algorithm shows the approach. The running time of this algorithm is $O(NM)$ where N is the amount and M is the number of coins.

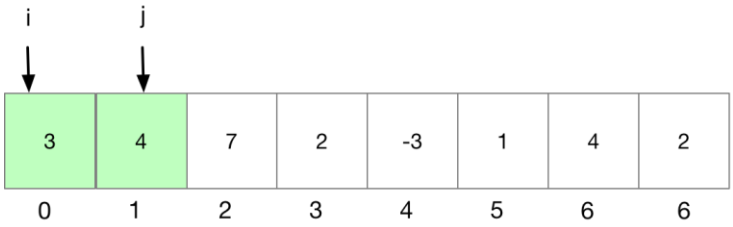
```
static int MinCoins(int[] coins,
                            int totalAmount)
{
    int numCoins = coins.Length;
    // table[i] will be storing
    // the minimum number of coins
    // required for i value. So
    // table[V] will have result
    int[] table = new int[totalAmount + 1];

    // Base case (If given
    // value V is 0)
    table[0] = 0;

    // Initialize all table
    // values as Infinite
    for (int i = 1; i <= totalAmount; i++)
          table[i] = int.MaxValue;

    // Compute minimum coins
    // required for all
    // values from 1 to V
    for (int amount = 1; amount <= totalAmount; amount++)
    {
          // We have to find the minimum number of coins with
          // which we can represent amount
          for (int coinIdx = 0; coinIdx < numCoins; coinIdx++)
          {
                int coin = coins[coinIdx];

                // The current coin is only considered if it is less
                // that or equal to the amount we need to achieve
                if (coin <= amount)
                {
                      // What is the minimum number of coins needed to
                      // represent amount-coin? Since this amount is
                      // by definition less than amount it must already
                      // be in the table. We just look it up
                      int numCoinsForAmountLessCoin
                            = table[amount - coin];

                      // Only update this entry if they was no better
                      if (numCoinsForAmountLessCoin + 1 < table[amount])
                        table[amount] = numCoinsForAmountLessCoin + 1;
                }
          }
    }
    return table[totalAmount];
}
```

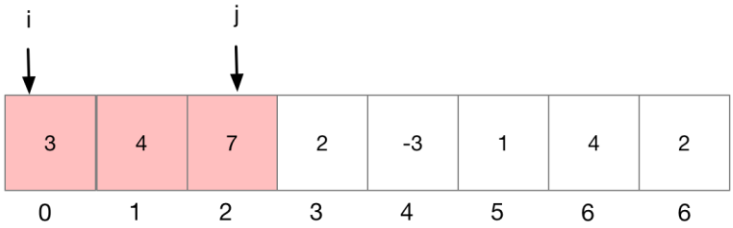
SUBARRAY SUM

Greedy

Count the number of contiguous subarrays that sum to k. We can use a brute force solution considering all combinations of indices as follows








```
public static int Fragments(int[] nums, int k)
{
    int count = 0;
    for (int i = 0; i < nums.Length; i++)
    {
          for   (int j=i ; j< nums.Length;j++)
          {
                int sum=0;
                for   (int idx=i; idx<=j;idx++)
                {
                      sum += nums[idx];
                }

                if (sum==k) count++;
          }
    }
    return count;
}
```

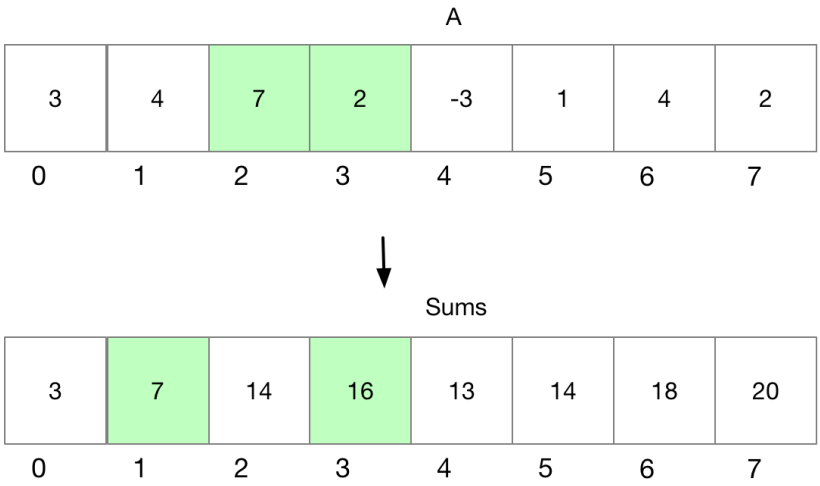
The number of sub-arrays is the number of all combinations of the indices i and j. This is basically 1+2+…+n-1+n which is equal to $\frac{n(n+1)}{2}$ For each of these we need to calculate the sum.

~First Optimisation

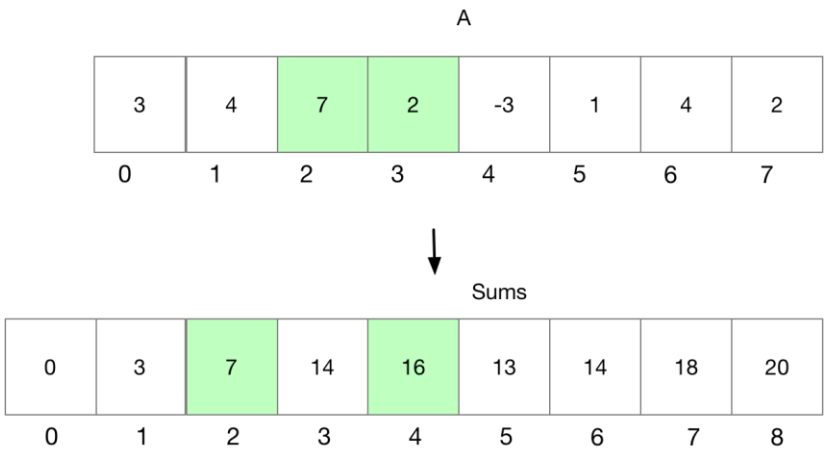
The first optimisation is to make use of the following

$$\sum_{l=i}^{j} A[l] = \sum_{l=0}^{j} A[l] - \sum_{l=0}^{i-1} A[l]$$

In the following example $Sum[2..3] = Sum[0..3] - Sum[0..1] = 16 - 7 = 9$



We make one further modification to deal with the case where i zero. We add one more entry to the array and shift as follows



The code becomes.

```
public static int Fragments(int[] nums, int k)
{
    int count =0;

    // one pass to initialize the sums
    int[] sums = new int[nums.Length+1];
    for (int i = 0; i < nums.Length; i++) sums[i+1] = sums[i] + nums[i];

    for (int i = 0; i < nums.Length; i++)
    {
          for(int j=i; j<nums.Length;j++)
          {
                if (sums[j+1]-sums[i] == k) count++;
          }
    }

    return count;
}
```

Dynamic Programming

We noted that

$$\sum_{l=i}^{j} A[l] = \sum_{l=0}^{j} A[l] - \sum_{l=0}^{i-1} A[l]$$

Let $\sum_{l=0}^{j} A[l] = y$ and $\sum_{l=0}^{i-1} A[l] = x$

If for any value of j there exists i such that $x = k - y$ then $\sum_{l=i}^{j} A[l] = k$ The algorithm then keeps track of how many value of i give each value

```
public static int Fragments(int[] a, int k)
{
    int fragCount=0;

    // Map each running Sum[0..i] with the number of values of
    // i for which that value has occured.
    IDictionary<int,int> valueCounts = new Dictionary<int, int>();

    // Special case
    valueCounts[0] = 1;

    // Hold the total A[0..j]
    int total =0;
    for (int l = 0; l < a.Length; l++)
    {
          int y = total += a[l];
          int x = y-k;

          // If Sum(a[0..i]) = x for n different values of i
          // we increment the fragment coint by n
          if (valueCounts.TryGetValue(x,out int n))
                fragCount = fragCount + n;


          // add Sum(a[0..i]) = y
          if (!valueCounts.TryGetValue(y,out int yCount )) yCount = 0;

                valueCounts[y] = yCount+1;
    }

    return fragCount;
}
```

The space required is a dictionary with at most N entries. The runtime performance of O(N)

LONGEST COMMON SUBSEQUENCE

A subsequence of some sequence $X = \langle x_1, x_2, \dots x_m \rangle$ is formed by removing zero or more elements of X. We say a sequence $Z = \langle z_1, z_2, \dots z_k \rangle$ is a subsequence of X is there exists a strictly increasing sequence of indices of X given by $\langle i_1, i_2, \dots i_k \rangle$ that maps values in x to values in Z. For example, if $X = \langle A, B, C, B, D, A, B \rangle$ then $Z = \langle B, C, D, B \rangle$is a subsequence and the indices are $\langle 2,3,5,7 \rangle$.

Given two sequences $X = \langle x_1, x_2, \dots x_m \rangle$ and $Y = \langle y_1, y_2, \dots y_n \rangle$ then a third sequence $Z = \langle z_1, z_2, \dots z_k \rangle$ is a common subsequence of X and Y if it is a subsequence of X and Y. Given $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$ then $\langle B, C, A \rangle$ is a common subsequence. It is not however the longest common subsequence.

The longest common subsequence problem takes two sequences $X = \langle x_1, x_2, \dots x_m \rangle$ and $Y = \langle y_1, y_2, \dots y_n \rangle$ and aims to find a longest common subsequence. Since there are 2^msubsequence of X (subsets of a set) we can code up a brute force algorithm that takes each subsequence of X and checks if it is a subsequence of Y. If it then it makes a note of it if its longer that the longest so far seen subsequence. Such an algorithm would have exponential running time. We can do better

Given a sequence $X = \langle x_1, x_2, \dots x_m \rangle$ we define the i'th prefix of X as $X_i = \langle x_1, x_2, \dots x_i \rangle$. So given $X = \langle A, B, C, B, D, A, B \rangle$ then $X_3 = \langle A, B, C \rangle$ We can now specify the solution of finding the LCS in terms of smaller instances of the same problem using the following information

$X = \langle x_1, x_2, \dots x_m \rangle$ and $Y = \langle y_1, y_2, \dots y_n \rangle$be two sequences. Furthermore, let sequence $Z = \langle z_1, z_2, \dots z_k \rangle$ be any longest common subsequence of X and Y.

1. If $x_m = y_n$then If $z_k = x_m = y_n$ and $Z_{k-1}$is a LCS of $X_{m-1}$and Y_{n-1}
2. If $x_m \neq y_n$then $z_k \neq x_m$ implies that Z is a LCS of $X_{m-1}$and Y
3. If $x_m \neq y_n$then $z_k \neq y_n$ implies that Z is a LCS of $X$and Y_{n-1}

If we let $c[i,j]$ be the length of the longest common subsequence of X_i and Y_j we get the following recursive algorithm

$$c[i,j] = \begin{cases} 0, & If\ i = 0\ or\ j = 0 \\ c[i-1, j-1], & If\ i,j > 0\ and\ x_i = y_i \\ \max(c[i.j-1], c[i-1,j]), & If\ i,j > 0\ and\ x_i \neq y_i \end{cases}$$

Priority Queues

FILE MERGE PATTERNS

We are given an array of integers representing the sizes of files to be merged. The merges occur two files at a time. The problem is to find the optimum merge strategy. For example, if we are given the array [2,3,4] the optimal merge strategy has a cost of 14.

At each step we want to merge the two smallest files to create a single new file leaving us with N-1 files. Then we merge the two smallest files again. At each step we merge the two smallest until we are left with one file.

The trick with the problem is to put all the file sizes onto a priority queue setup in minimum configuration. Then we continually remove the two smallest elements, add them, and push them back onto the queue until we are left with one element. The algorithm is as follows.

```
static int CalculateOptimumMergeCost(int size, int[] files)
{
    PriorityQueue<int> pq
          = new PriorityQueue<int>((x,y)=> y.CompareTo(x),size);

    for (int i = 0; i < size; i++)
          pq.Add(files[i]);

    // Keep track of the total cost
    int totalCost = 0;

    while (pq.Count > 1)
    {
          // Remove the two smallest file sizes from the
          // the heap and keep a track of their sizes
          int smallestFileSize = pq.Dequeue();
          int secondSmallestFileSize = pq.Dequeue();

          // Merge them into a new file
          int mergedFileSize = smallestFileSize+secondSmallestFileSize;

          // Add the cost of the merge to the running count
          totalCost += mergedFileSize;

          // Push the new file on the heap ready for the next
          // round of merging
          pq.Add(mergedFileSize);
    }

    return totalCost;
}
```

Space and Time

The space required by the heap is O(N). We have at most N elements on it. The time complexity is 2N-1 inserts and 2(N-1) dequeue. Each insert is on average O(1) giving us O(N) in total. Each dequeue is O(LogN) giving us a total runtime performance of O(NLogN)

Arrays and String Problems

Questions -Arrays and String Problems

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

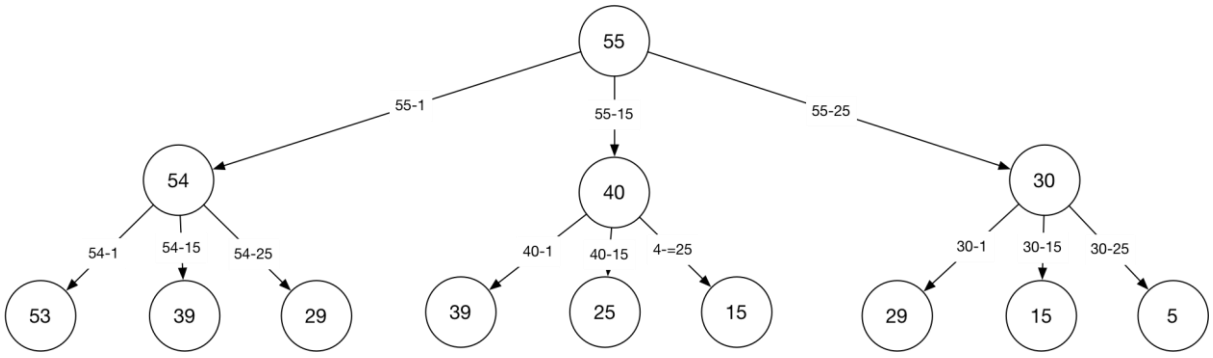
Questions -Recursion

Given an array of integers representing a set of coins and a currency amount find the minimum number of coins that sum to that currency amount. For example, five {1,15,25} and 55 we might find the minimum is 1 x 25, 1 x 15, 10 x 1

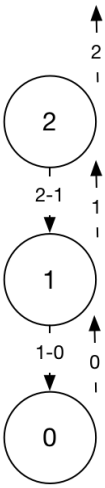
The following code is a basic greedy recursive algorithm to solve this problem

```
public int MinCoins(int[] coins, int amount)
{
    if (amount == 0) return 0;

    int minNumber = int.MaxValue;

    for (int i = 0; i < coins.Length; i++)
    {
          if (amount - coins[i] >= 0)
          {
                int minCoins = MinCoins(coins,amount-coins[i]);
                if (minCoins < minNumber)
                      minNumber = minCoins;
          }
    }

    return minNumber+1;
}
```

The idea is that at each stage we take an amount and recursively call ourselves to find the minimum number of coins if we use each type of coin to reduce the amount. We start as follows. We start by calling `MinCoins(new int[] { 1,15, 35},55)` *We then calculate the minimum number of coins to represent 55 by adding one to minimum of calling itself recursively with each coin value.*



The terminating case is then as follows


2
2
2-1
1
1
1-0
0
0

Techniques

Divide and Conquer

Many problems can be solved recursively. At each step the code solves related subproblems using recursion. At each level of the recursion we

1. **Divide** the problem into several smaller instances of the same kind
2. **Conquer** Solve the subproblems recursively. If however the subproblems are small enough we just solve the in a simple non-recursive manner. At this stage we say the recursion has bottomed out
3. **Combine** the solutions of the subproblems into the solution of the problem

We describe situations where the subproblems are large enough to solve recursively as the recursive case. When the problems are small enough to solve non-recursively, we say we are in the base case. At this stage we say the recursion has bottomed out.

The divide and conquer method works well when the problem can be broken down into disjoint subproblems

.NET Collections

We consider three kinds of collections provided by the .NET Framework.

- Generic Collections – The basic strongly typed collections
- Read-only collections
- Concurrent Collections – Optimised for concurrent add and remove
- Immutable Collections

We will not look at the non-generic collections as they are superseded by the generic collections.

Generic Collections

Symbol Tables

Dictionary<K, V>

The Dictionary is implemented using a Hashtable and as such does not support ordering or retrieval by integer index. Retrieval by key, insertion and deletion are fast constant time operation

Operation	Performance	Notes
Retrieve by Key	O(1)	
Insertion	O(1)	There is no concept of order and consequently no distinction between sequential and random insert
Deletion	O(1)	

Sorted Dictionary<K, V>

The SortedDictionary is implemented using a red-black tree and as such it keeps elements in order by key. Insertion, retrieval and deletion are all logarithmic operations. Strangely the .NET implementation does not support retrieval by integer index even though a red black tree could provide this as a logarithmic time operation.

Operation	Performance	Notes
Retrieve by Key	O(log N)	
Insertion Random	O(log N)	
Insertion Sequential	O(log N)	
Deletion	O(log N)	

SortedList<K, V>

The SortedList is implemented as a List and a Hashtable. As such it can provide fast constant time lookup by key and integer index. The compromise is increased storage and slow linear time deletion. Random insertion is also a slow operation.

Operation	Performance	Notes
Retrieve by Key	O(1)	
Retrieve by Index	O(1)	
Insertion Random	O(N)	
Insertion Sequential	O(1)	
Deletion	O(N)	

KeyedCollection

Combines a list and a dictionary to provide O(1) indexed retrieval and amortized O(1) key lookup. Internally holds a list and a dictionary. The keys must be a property of the items stored in the collection

```
void Main()
{
    var keyedCollection = new ProductCollection()
    {
          new Product() {Id=3,Name="Product3"},
          new Product() {Id=4,Name="Product4"},
    };

    // Lookup by key
    WriteLine(keyedCollection[3].ToString());
```

```
    // Lookup by index
    WriteLine(keyedCollection.ElementAt(1).ToString());
}

public class ProductCollection : KeyedCollection<int, Product>
{
    protected override int GetKeyForItem(Product item) => item.Id;
}

public class Product
{
    public int Id { get; set; }

    public string Name { get; set; }

    public override string ToString() => $"{Id}, {Name}";
}
 override string ToString() => $"{Id}, {Name}";
}
```

ReadOnlyCollection<T> / ReadOnlyDictionary<T>

The read only collections provide read-only wrappers or proxies around collections. This enables a type to provide clients with a read-only interface to a collection that it can still add to. Any changes to the input collection are visible through the read only wrapper.

Concurrent Collections

The concurrent collection types are optimised for concurrent add and remove operations. Internally they use low locking and lock-free synchronization strategies. In some scenarios the concurrent collection performs significantly better than protecting non-concurrent collections' add and remove operations with external locks

In order to determine when to use the concurrent collections and when to use the non-concurrent collections with external locks protected the add and remove operations it is necessary to carry out performance analysis on representative scenarios and loads.

SCENARIOS

Pure Producer-Consumer	Any single thread is either adding to removing elements. No single thread does both
Mixed Producer-Consumer	Single threads are both adding and removing elements

There are four Concurrent collections

- ConcurrentStack<T>
- ConcurrentQueue<T>
- ConcurrentBag<T>
- ConcurrentDictionary<T>

The stack, queue and bag are internally backed by linked lists which are more appropriate for low lock multithreaded implementation. The downside of this choice is increased memory usage.

Internally the concurrent queue and concurrent stack are completely lock free. They use a combination of compare and swap operations and memory barriers to synchronize. The downside of this is of course that they can spin under contention.

The concurrent dictionary is completely lock free for reads but uses locks for writes. As such it is optimized for reads.

Immutable Collections

Immutable collections are collections whose internal structure never changes. By internal structure we mean the number of elements and their relative order. Methods such as `ImmutableStack.Push` creates a new object from the existing stack. The new collection shares much of the memory of the original collection with some changes to reflect the difference. The original collection is completely unchanged. The designers of the immutable collections had the following goals

- Reuse as much memory as possible
- Avoid copying
- Reduce pressure on GC
- Offer similar operations to mutable collections with competitive performance

We will now look at the different types of immutable collections.

IMMUTABLESTACK<T>

The simplest immutable collection. Internally implemented as a linked list. The immutable stack is a pointer to the top element of the linked list. We can push and pop elements without changing the stack.

```
IImmutableStack<char> s1 = ImmutableStack.Create(new char[]
{'a','b','c'});

var s2 = s1.Pop();
var s3 = s2.Pop();

WriteLine(s1.Peek());
WriteLine(s2.Peek());
WriteLine(s3.Peek());
```

IMMUTABLELIST<T>

The immutable list uses a balanced binary tree internally. As such addition, removal, insertion or lookup are all O(log N). If we need to perform M operations, we get O(M log N). We can improve this for some operations by using builders and freezing.

GARBAGE COLLECTION

One of the biggest problems with immutable collections is that when we create a new immutable collection as a diff on a source object often the source object is no longer referenced and becomes available for garbage collection. Garbage collection freezes all other threads when it runs, and hence frequent garbage collection runs can cause performance problems.

One of the worst causes of such garbage generation occurs when initializing large immutable collections. The overhead of such operations can be mitigated by using builders as follows

```
ImmutableList<char> s2 = ImmutableList.Create<char>();
ImmutableList<char>.Builder builder = s2.ToBuilder();

builder.Add('a');
builder.Add('z');

s2 = builder.ToImmutable();
```

An immutable collection disallows element level assignments and maintains at all times its structure which is defined as the number of elements and there relative order. At the same time immutable

collections provide an API to support mutations. If we push an element onto an immutable stack, we end up with two stacks: one without the new element and one with it.