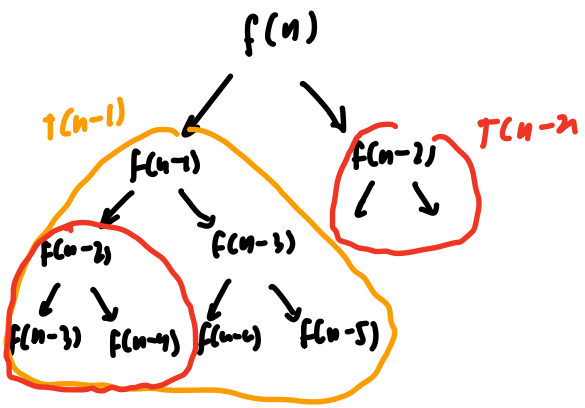
Runtime of algorithms:

```
long fib(long n)
{
    if (n==1||n==2){
        return 1;
    }
    return fib(n-1)+fib(n-2);
}
```

Assume running time for fib is a function $T(n)$

$T(n) = T(n-1)+T(n-2)$ since $fib(n) = fib(n-1)+fib(n-2)$

Base case runtime : $T(n) = 1$ $1 \leq n \leq 2$



$T(n-1) > T(n-2)$

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

$< 2T(n-1)$

$< 2^2 T(n-2)$

$< 2^3 T(n-3)$

$\vdots$

$< 2^k T(n-k)$

$\vdots$

$< 2^{n-2} T(n-(n-2))$ ← $T(2)$

$= 2^{n-2} T(2)$

$= 2^n/4$ ← worst case

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

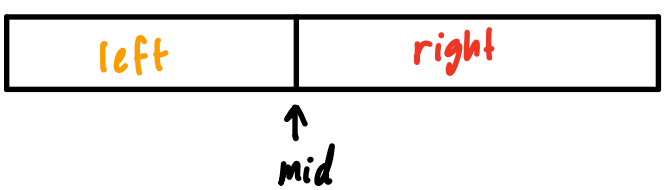
Searching

① Linear Search (Checks all elements until element is found, returns position if found, else returns -1)

```
long search (long n, const long arr[n], long q) {
    for(long i=0; i<n; i+=1){
        if(list[i]==q){
            return i;
        }
    }
    return -1;
}
```

Worst case: Element is not in array (we iterate through n elements)
O(n)

Recursive Implementation



① Base case: middle element == q
element not in list

② Wishful thinking: We can split the array in two halves, searching the left and the right

Runtime analysis:

Worst case: search left half (n/2)
+
search right half (n/2)

$T(n) = 2T(\frac{n}{2}) + 1$ ← additional checks

$T(1) = 1$ ← base case

③ recursion:

```
if (i>j){
    return -1; //elem not in list
}
if (arr[mid]==q){
    return mid; //q is middle elem
}
```

$$T(n) = 2T\left(\frac{n}{2}\right) + 1$$
$$= 2^2 T\left(\frac{n}{4}\right) + 2 + 1$$
$$= 2^3 T\left(\frac{n}{8}\right) + 4 + 2 + 1$$
$$= 2^k T\left(\frac{n}{2^k}\right) + (2^k - 1)$$
$$\vdots$$
$$= 2^{\log_2 n} T(1) + n + \frac{n}{2} + \dots$$
$$\leq nT(1) + 2n$$
$$\approx O(n)$$

for $\frac{n}{2^k} = 1$, $2^k = n$, $k = \log_2 n$

```
long pos = search (arr, i, mid-1)
if (pos >= 0){
    return pos; // if elem found in left
}
return search (arr, mid+1, j);
// else search right
```

$$n + \frac{n}{2} + \frac{n}{4} + \dots + 1$$
$$S_n = \frac{n\left(1 - \left(\frac{1}{2}\right)^n\right)}{1 - \frac{1}{2}}$$
$$\lim_{n \to \infty} S_n = 2n$$

Binary search (Sorted Array)

We search if the element is the midpoint, we can return the midpoint.

If arr[mid] > q, we can search the right half only (as the input arr is sorted)

If arr[mid] < q, we can search the left half only

Cuts size of search in half for each iteration

```
long search(const long list[n], long i, long j, long q) {
    if (i > j) {
        return -1;
    }
```

```
long mid = (i+j)/2;
if (list[mid]==q){
    return mid;
}
if (list[mid]>q){
    return search(list, i, mid-1);   //left
}
return search(list, mid+1, j)        //right
}
```

$T(n) = T(\frac{n}{2}) + 1$ we search only one side

$T(1) = 0$

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

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

$\vdots$

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

$= T(1) + \log_2 n$ as $k = \log_2 n$ when $\frac{n}{2^k} = 1$

$= 1 + \log_2 n$

Sorting

Counting sort (Analysis):

initialising frequency array - $O(MAX)$

loop to count frequency - $O(n)$

loop to store elements in out[] - $\sum_{i=0}^{MAX} f_i \; n$ - $O(n)$

$\sum_{i=0}^{MAX} f_i$: Total time all numbers appear

total runtime = $O(2n + MAX) = O(n + MAX)$ ..

unsure if $MAX > n$ or $n > MAX$

Bubble sort (swaps pairs that are out of order until array is sorted)

```
void swap (long a[], long i, long j){
        long tmp = a[i];
        a[i] = a[j];
        a[j] = temp;
}

void bubble_pass (long last, long a[]){
     for (size_t i=0; i<last; i+=1){
          if (a[i] > a[i+1]){
               swap (a, i, i+1);
          }
     }
}
void bubble_sort (size_t n, long a[n])
   {

        for (size_t last = n-1; last > 0; last -= 1){
            bubble_pass (last, a)
        }
   }
```

O(last)

$O(last \times last) = O(last^2) \approx O(n^2)$