Design and Analysis of Algorithms

Time & Space Tradeoff

To measure performance.

Sujeet Patil
Shreya Shetty
Bharvi Sharma
Abhijith Pillai
Romil Shah

Table of Contents

TIME-SPACE TRADE OFF

A space-time or time-space tradeoff in computer science is a case where an algorithm or program trades increased space usage with decreased time. Here, space refers to the data torage consumed in performing a given task (RAM, HDD, etc), and time refers to the time consumed in performing a given task (computation time or response time).

A tradeoff is a situation where one thing increases and another thing decreases. It is a way to solve a problem in:

- Either in less time and by using more space, or
- In very little space by spending a long amount of time.

Efficiency of Algorithms

We measure an algorithm’s efficiency using the time and space (memory) it takes for execution.

The best algorithm is the one that completes its execution in the least amount of time using the least amount of space

But often, in reality, algorithms have to tradeoff between saving space or time.

Time Complexity

Time Complexity of an algorithm is the representation of the amount of time required by the algorithm to execute to completion, commonly expressed using asymptotic notations like Big O - O(n) etc.

Types of Time Complexities:

1. Linear Time Complexity: O(n)
2. Logarithmic Time Complexity: O(log n)
3. Quadratic Time Complexity: O(n²)
4. Exponential Time Complexity: O(2^n)

Space Complexity

The space complexity of an algorithm or a computer program is the amount of memory space required to solve an instance of the problem as a function of characteristics of the input. It is the memory required by an algorithm until it executes completely.
It is composed of two different spaces; Auxiliary space and Input space.

It is composed of two different spaces.

1

Auxilary Space

2

Input Space

Understanding Asymptotic Analysis

Evaluation of the performance of an algorithm in terms of input size

1st Case

```
int i = 1 to N
N = N + N
print N
```

2nd Case

```
return N * N
```

3rd Case

```
T(n)=n^2+2n+8.
```

Types of Analysis

Worst Case

Best Case

Average Case

Types of Analysis

Worst Case

- Defines the input for which algorithm takes longest time to execute.
- Algorithm runs slower.
- Algorithm executes maximum number of steps on input data of size n.

Types of Analysis

Best Case

- Defines the input for which algorithm takes lowest time to execute.
- Algorithm runs fastest.
- Algorithm executes least number of steps on input data of size n.

Types of Analysis

Average Case

- Provides a prediction about the running time of the algorithm.
- Assumes that the input is random.
- Algorithm performs average number of steps on input data of size n.

Asymptotic Notations

Big - O Notation

Omega - Ω Notation

Theta - Θ Notation

Asymptotic Notations

Big - O Notation

This notation gives the tight upper bound of the given algorithm / function f(n). It is represented as

```
f(n) = O(g(n))
```

It means, for larger values of n, the upper bound of function f(n) is a function g(n).
Here upper bound means, value of f(n) cannot exceed g(n) after a particular value of n.

Asymptotic Notations

Big - O Notation

Some Big - O Examples:

```
f(n) = n2 + 1
```

n2 + 1 ≤ 2n2, for all n ≥ 1
Therefore, n2 + 1 = O(n2),
with c = 2 and n0 = 1

```
f(n) = 2n3 - 2n2
```

2n3 - 2n2 ≤ 2n3, for all n ≥ 1
Therefore, 2n3 - 2n2 = O(2n3),
with c = 2 and n0 = 1

Asymptotic Notations

Omega - Ω Notation

This notation gives the tight lower bound of the given algorithm / function f(n). We can represent it as

```
f(n) = Ω(g(n))
```

It means, for larger values of n, g(n) is the lower bound of function f(n).
Here lower bound means, rate of growth of f(n) is always greater than or equal to the rate of growth of function g(n) after a particular value of n i.e. n0.

Asymptotic Notations

Omega - Ω Notation

Some Ω Examples :

```
f(n) = 5n2
```

such that:
0 ≤ cn ≤ 5n2 => c = 1 and n0 = 1
Therefore, 5n2 = Ω(n2)

Ω Notation can also be defined as
Ω(g(n)) = { f(n) : there exists positive constants n0 and c such that 0 ≤ cg(n) ≤ f(n) for all n ≥ n0 }.

Asymptotic Notations

Theta - Θ Notation

This notation gives a range of upper bound and lower bound and determines whether the upper bound and lower bound of the given algorithm are same.

For example, assume `f(n) = 10n + n`

its tight upper bound is O(n) and the lower bound is Ω(n).

Asymptotic Notations

Theta - Θ Notation

Some Θ Examples :

```
Prove n ≠ Θ(n2)
```

c1n2 ≤ n ≤ c2n2,
only holds for n ≤ 1 / c1
Therefore, n ≠ Θ(n2)

```
Prove 6n3 ≠ Θ(n2)
```

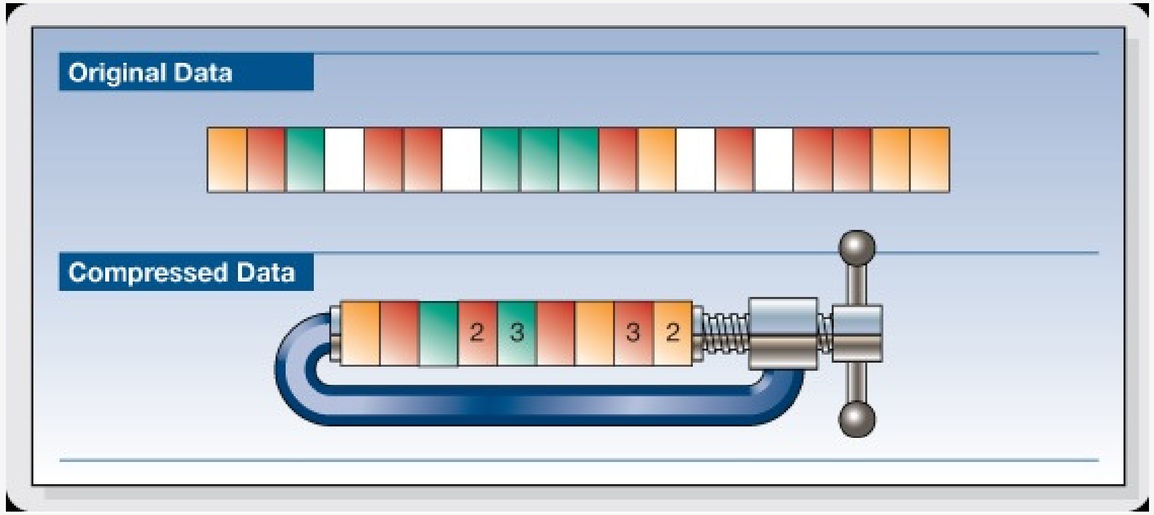
c1n2 ≤ 6n3 ≤ c2n2,
only holds for n ≤ c2 / 6
Therefore, 6n3 ≠ Θ(n2)

TYPES OF TIME-SPACE TRADE OFF

1. Compressed vs. uncompressed data
2. Re-rendering vs. stored images
3. Smaller code vs. loop unrolling
4. Lookup tables vs. recalculation

1.Compressed vs. uncompressed data

A space–time tradeoff can be applied to the problem of data storage. If data is stored uncompressed, it takes more space but access takes less time than if the data were stored compressed (since compressing the data reduces the amount of space it takes, but it takes time to run the decompression algorithm).



2. Re-rendering vs. stored images

In this case storing only the SVG source of a vector image and rendering it as a bitmap image every time the page is requested would be trading time for space; more time used, but less space.

Storing an image in the cache is faster than re-rendering but requires more space in memory; more space used, but less time.

3.Smaller code vs. loop unrolling

Larger code size can be traded for higher program speed when applying loop unrolling. This technique makes the code longer for each iteration of a loop, but saves the computation time required for jumping back to the beginning of the loop at the end of each iteration.

Code without Loop unrolling

```c
// This program does not uses loop unrolling.
#include<stdio.h>

int main(void)
{
    for (int i=0; i<5; i++)
        printf("Hello\n"); //print hello 5 times

    return 0;
}
```

Code with Loop unrolling

```c
// This program uses loop unrolling.
#include<stdio.h>

int main(void)
{
    // unrolled the for loop in program 1
    printf("Hello\n");
    printf("Hello\n");
    printf("Hello\n");
    printf("Hello\n");
    printf("Hello\n");

    return 0;
}
```

4.Lookup tables vs. recalculation

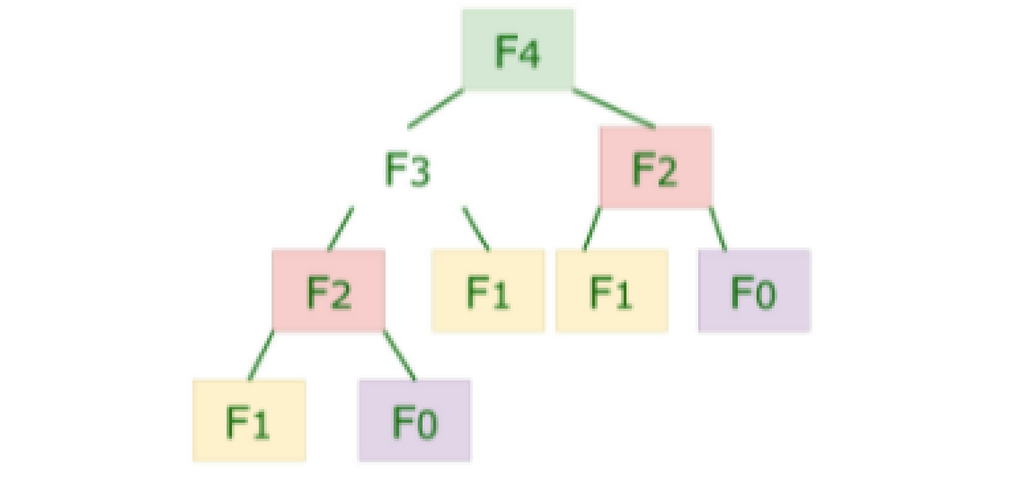
A common situation is an algorithm involving a lookup table: an implementation can include the entire table, which reduces computing time, but increases the amount of memory needed, or it can compute table entries as needed, increasing computing time, but reducing memory requirements.

Time Complexity: $O(2^n)$
Auxiliary Space: O(1)

```cpp
#include <iostream>
using namespace std;

// Function to find Nth Fibonacci term
int Fibonacci(int N)
{
    // Base Case
    if (N < 2)
        return N;

    // Recursively computing the term
    // using recurrence relation
    return Fibonacci(N - 1) + Fibonacci(N - 2);
}

// Driver Code
int main()
{
    int N = 5;

    // Function Call
    cout << Fibonacci(N);

    return 0;
}
```

```cpp
#include <iostream>
using namespace std;

// Function to find Nth Fibonacci term
int Fibonacci(int N)
{
    int f[N + 2];
    int i;

    // 0th and 1st number of the
    // series are 0 and 1
    f[0] = 0;
    f[1] = 1;

    // Iterate over the range [2, N]
    for (i = 2; i <= N; i++) {

        // Add the previous 2 numbers
        // in the series and store it
        f[i] = f[i - 1] + f[i - 2];
    }

    // Return Nth Fibonacci Number
    return f[N];
}

// Driver Code
int main()
{
    int N = 5;

    // Function Call
    cout << Fibonacci(N);

    return 0;
}
```

Using Dynamic Programming to reduce the complexity by memoization of the overlapping subproblems.



Time Complexity: O(n)
Auxiliary Space: O(n)

Example

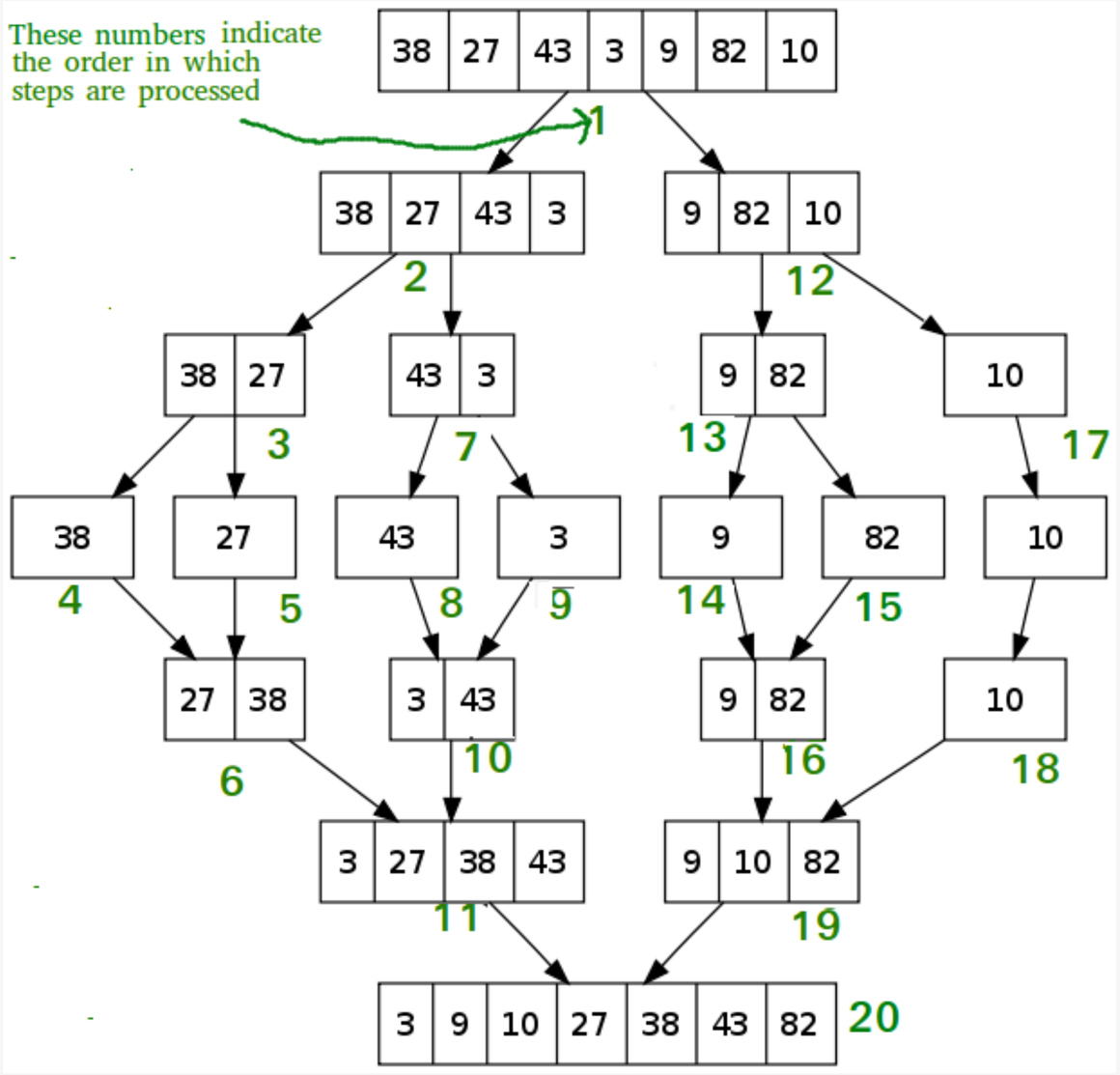
Merge sort VS Binary Search

Merge Sort

Merge Sort is a Divide and Conquer algorithm. It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves.

MergeSort(arr[], l, r)
If r > l
1. Find the middle point to divide the array into two halves:
middle m = l+ (r-l)/2
2. Call mergeSort for first half:
Call mergeSort(arr, l, m)
3. Call mergeSort for second half:
Call mergeSort(arr, m+1, r)
4. Merge the two halves sorted in step 2 and 3:
Call merge(arr, l, m, r)

Merge Sort


These numbers indicate the order in which steps are processed
38 27 43 3 9 82 10
1
38 27 43 3
9 82 10
2
12
38 27
43 3
9 82
10
3
7
13
17
38
27
43
3
9
82
10
4
5
8
9
14
15
27 38
3 43
9 82
10
6
10
16
18
3 27 38 43
9 10 82
11
19
3 9 10 27 38 43 82
20

Binary Search

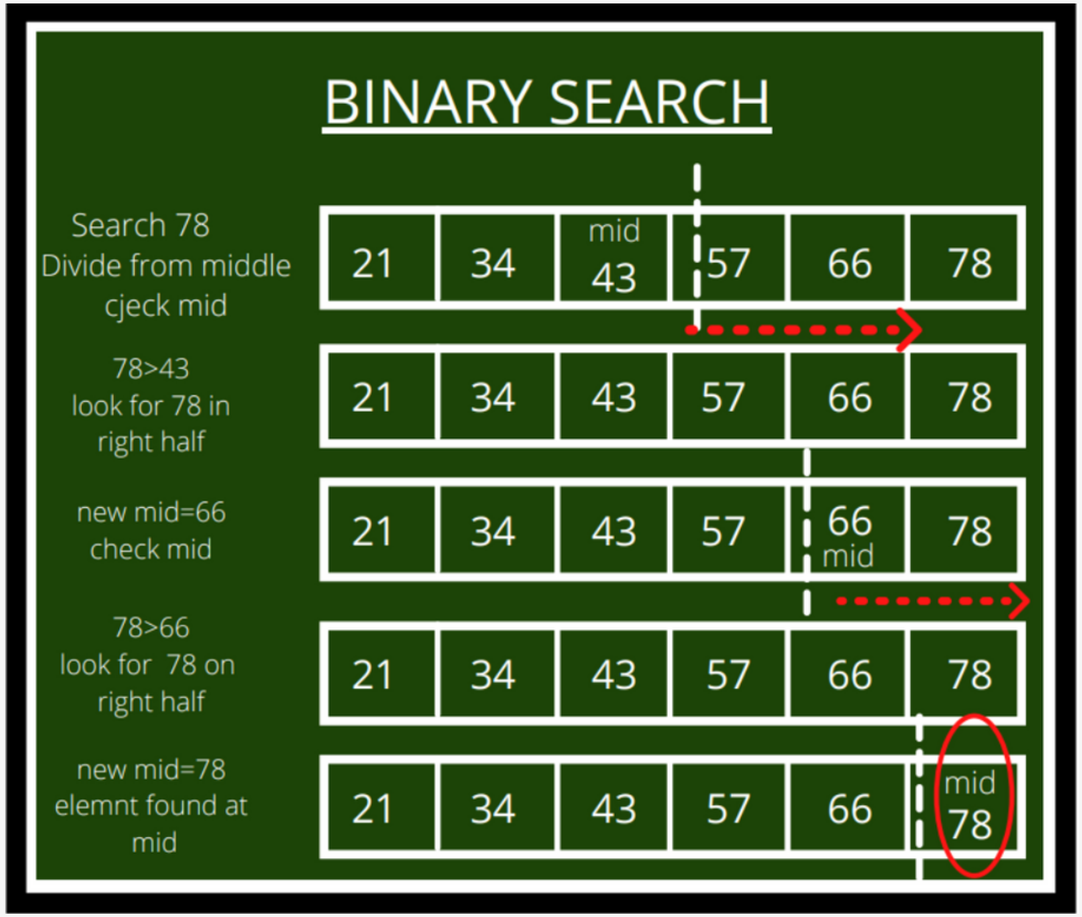
Binary Search Algorithm is one of the widely used searching techniques. It can be used to sort arrays. This searching technique follows the divide and conquer strategy. The search space always reduces to half in every iteration.
Binary Search Algorithm is a very efficient technique for searching but it needs some order on which partition of the array will occur.

1. We are given an input array that is supposed to be sorted in ascending order.
2. We take two variables which will act as a pointer i.e, beg, and end.
3. Beg will be assigned with 0 and the end will be assigned to the last index of the array.
4. Now we will introduce another variable mid which will mark the middle of the current array. That will be computed as (low+high)/2.
5. If the element present at the mid index is equal to the element to be searched, then just return the mid index.
6. If the element to be searched is smaller than the element present at the mid index, move end to mid-1, and all RHS will get discarded.
7. If the element to be searched is greater than the element present at the mid index, move beg to mid+1, and all LHS will get discarded.

Binary Search

Iterative Approach:

```
binarySearch(arr, size)
    loop until beg is not equal to end
   midIndex = (beg + end)/2
   if (item == arr[midIndex] )
      return midIndex
   else if (item > arr[midIndex] )
      beg = midIndex + 1
   else
      end = midIndex - 1
```



Merge Sort

The time complexity of merge sort of is O(nlogn) in all cases i.e.

Best Case : Ω(n*logn)

Worst Case : O(n*logn)

Average Case : Θ(n*logn)

The Space Complexity of merge sort of is:

Best Case : Ω(n)

Worst Case : O(n)

Average Case : Θ(n)

Binary Search

Binary Search has O(logn) time complexity for :

Best case : Ω(1)

Worst case O(logn)

Average case : Θ(logn)

The Space Complexity of Binary search for:

Best Case : Ω(1)

Worst Case : O(1)

Average Case : Θ(1)

Space or Time

Approach 1

take more space
but takes less
time to complete
execution

Approach 2

take less space
but takes more
time to complete
execution.

Thank you!