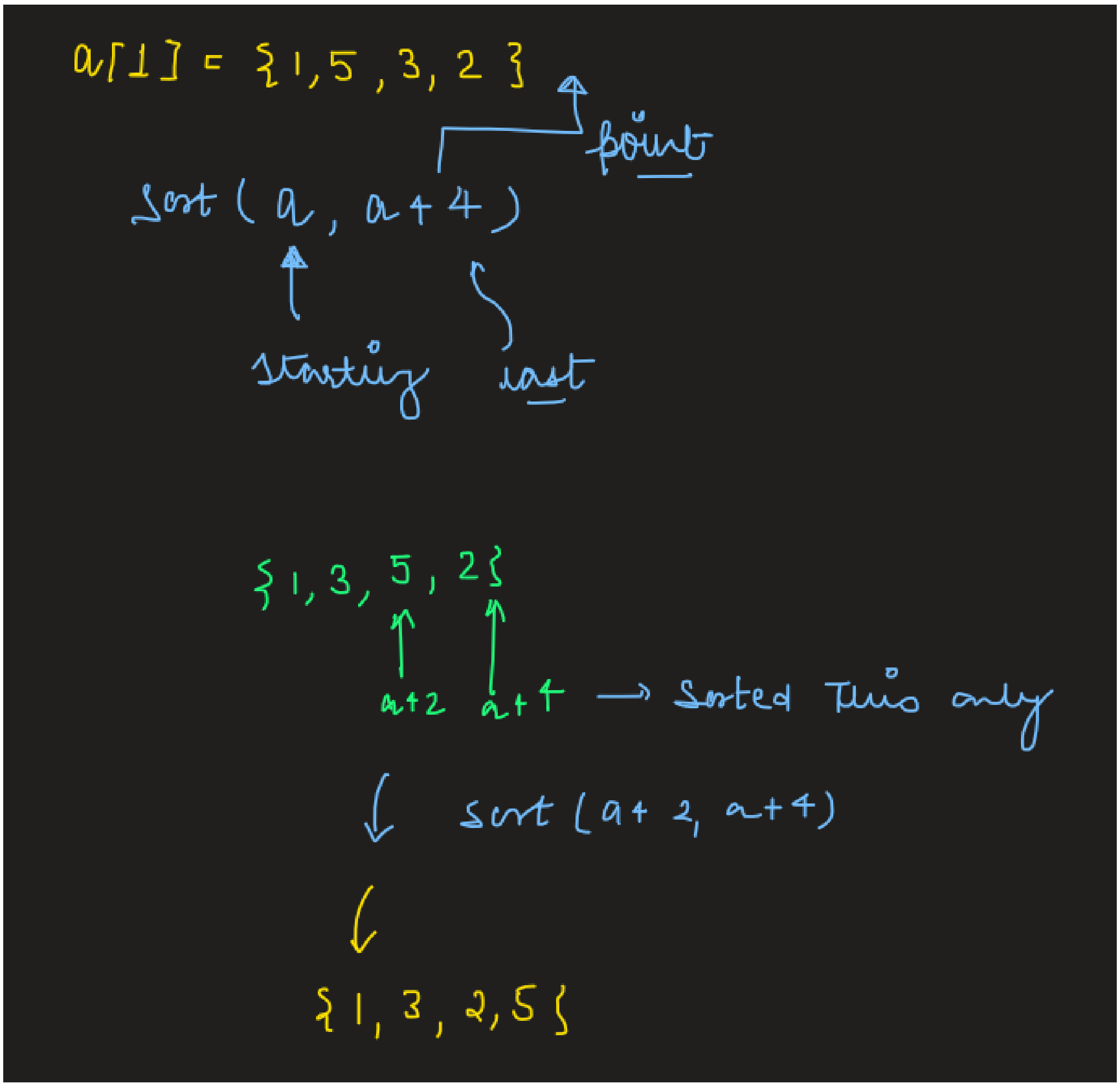
3️⃣

Algorithm

An algorithm is **a procedure used for solving a problem or performing a computation** Algorithms act as an exact list of instructions that conduct specified actions step by step in either hardware- or software-based routines. Algorithms are widely used throughout all areas of IT.

```
#include <bits/stdc++.h>
using namespace std;

bool comp(pair<int, int> p1, pair<int, int> p2)
{
      if(p1.second <p2.second)
      {
            return true;
      }
      if(p1.second>p2.second)
      {
            return false;
      }

       // they are same

      if(p1.first>p2.first)
      {
            return true;
      }
      return false;
```

```
}
void explainExtra()
{
	int n;
	int a[n];
	vector<int> v;
	sort(a, a + n);
	sort(v.begin(), v.end());

	sort(a + 2, a + 4);

	// sort(a, a + n, greater <int> ); // sorted in descending order

	pair<int, int> a[] = {{1, 2}, {2, 1}, {4, 1}};

	// Sort it according to second element
	// if second element is same, then sort
	// it according to first element but in descending

	// (My way)
	sort(a, a + n, comp); // comp --> self written comparator


	// {{4,1}, {2,1, {1,2}}

	int num = 7;
	int cnt = __builtin_popcount(num);   // 3


	long long num = 165786578687;
	int cnt = __builtin_popcountll(num);

	string s = "123";
	sort(s.begin(), s.end());
	do
	{
		/* code */
		cout << s << endl;
	} while (next_permutation(s.begin(),s.end()));

	// {1,10,5,6}
	int maxi = *max_element(a, a + n);// return 10
}
int main()
{
	explainExtra();
	return 0;
}
```



a[1] = {1, 5, 3, 2}

sort(a, a+4)

point

starting

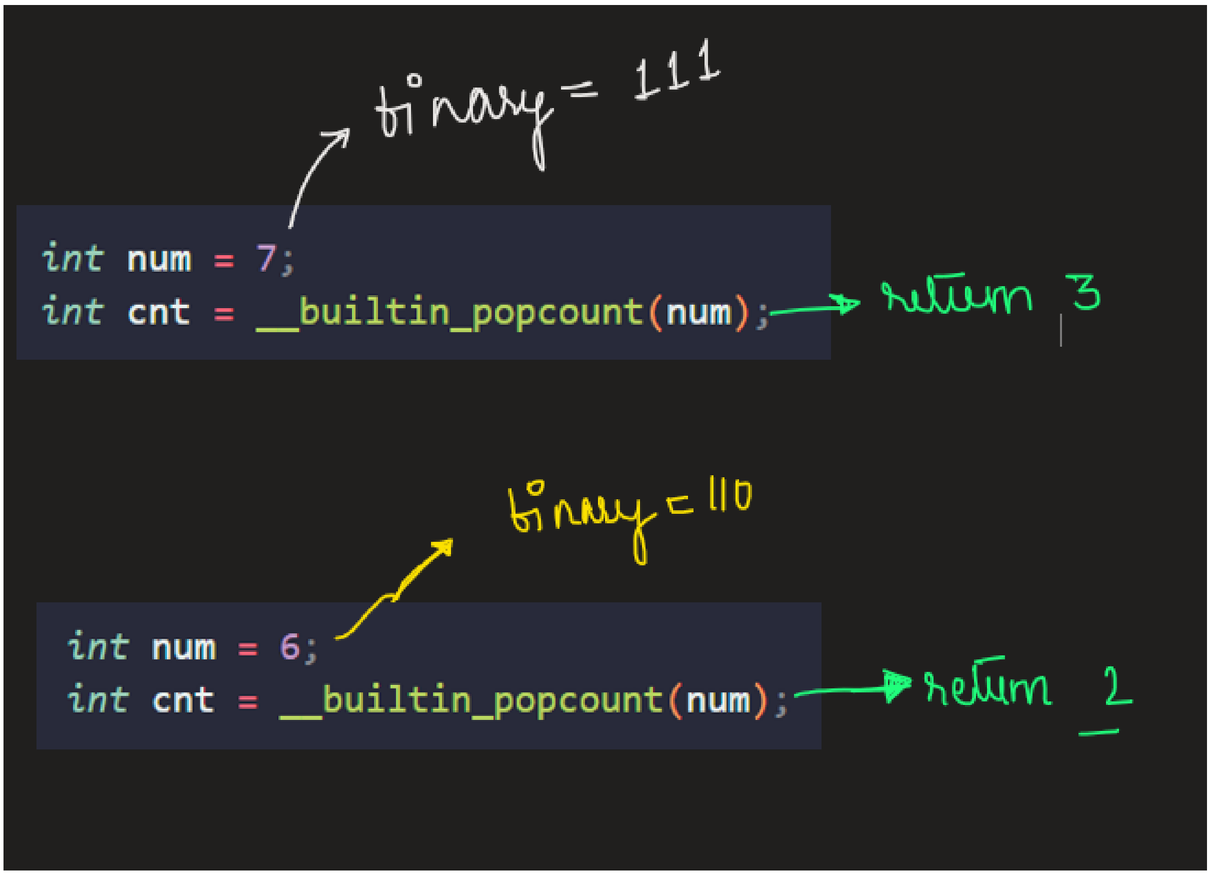
last

{1, 3, 5, 2}

a+2 a+4 → Sorted this only

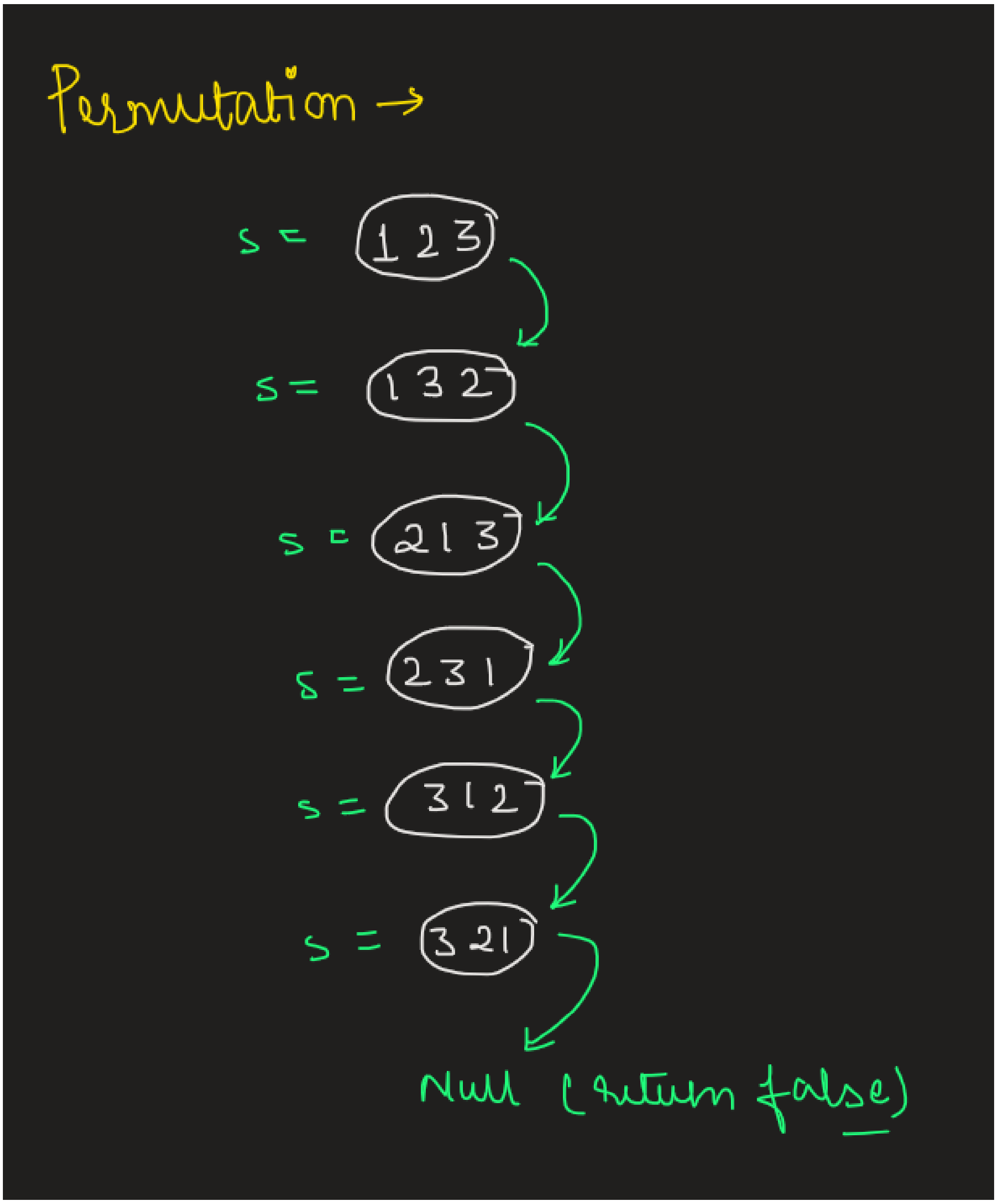
sort(a+2, a+4)

{1, 3, 2, 5}



```
int num = 7;
int cnt = __builtin_popcount(num);
```

```
int num = 6;
int cnt = __builtin_popcount(num);
```


Permutation →
s = 123
s = 132
s = 213
s = 231
s = 312
s = 321
Null (return false)

sort() in C++ STL

Sorting is one of the most standard operations used very frequently while writing programs. Writing the complete sorting algorithm might be time consuming and hence STL provides us with a standard inbuilt function to sort any container very easily.

Benefits of using sort():

- Ease of implementation, omits writing lengthy code to implement different sorting algorithms.
- Need not to worry about time complexity, it's one of the most efficient functions with a N*logN time complexity and uses a mix of quicksort and mergesort in its internal implementation.

Syntax:

The sort() function in STL accepts two mandatory parameters: begin, and end, and sorts the range with-in the container in ascending order by default.

```
sort(begin, end)
```

begin: It is an iterator pointing to the first element of a container.

end: It is an iterator pointing to element just after the last element of the container.

Example 1:

```
int arr[] = {4,2,1}

sort(arr, arr+3);

Output: arr[] = {1,2,4}
```

Example 2:

```
vector<int> vec = {4,2,1}

sort(vec.begin(), vec.end());

Output: vec = {1,2,4}
```

Code For Array

```
#include<iostream>
#include<vector>
#include<algorithm>

using namespace std;

int main() {

    int arr[] = {4,2,1};

    sort(arr, arr+3);

    cout<<arr[0]<<" "<<arr[1]<<" "<<arr[2];

    return 0;
}

Output:

1 2 4
```

Code For Vector

```
#include<iostream>
#include<vector>
#include<algorithm>

using namespace std;

int main() {

    vector<int> vec = {4,2,1};

    sort(vec.begin(), vec.end());

    cout<<vec[0]<<" "<<vec[1]<<" "<<vec[2];

    return 0;
}

Output:
```

```
1 2 4
```

Can we also sort a container in descending order using sort()?

Yes, by using comparators. The role of a comparator in most functions is to compare between two elements before performing an operation.

sort() function accepts an optional third parameter which is a comparator that allows us to define a custom comparison check between two elements while sorting them. In order to sort the container in descending order, we need to put the greater element first while comparing between two elements.

In STL, we already have a comparator defined to do this which is called **greater()**. We just need to pass this greater<container_data_type>() as the third parameter to sort function as shown in the below codes and it will sort the comparator in descending order.

Code for Array: —

```
#include<iostream>
#include<vector>
#include<algorithm>

using namespace std;

int main() {

    int arr[] = {4,2,1};

    sort(arr, arr+3, greater<int>());

    cout<<arr[0]<<" "<<arr[1]<<" "<<arr[2];

    return 0;
}
Output:

4 2 1
```

Code for Vector —

```
#include<iostream>
#include<vector>
#include<algorithm>

using namespace std;

int main() {

    vector<int> vec = {4,2,1};

    sort(vec.begin(), vec.end(), greater<int>());

    cout<<vec[0]<<" "<<vec[1]<<" "<<vec[2];

    return 0;
}
Output:

4 2 1
```

Can we also sort the container in a custom order – defined by a custom container?

Yes, we can. We can write a custom comparator to compare between any two elements however we wish to do it and it can then be used to sort the entire container accordingly.

But wait, isn't sorting is of just two types ascending and descending? Why do we need a custom sorting algorithm?

Before we answer that, can you think of using the sort() function with a vector of pairs where the vector needs to sorted according to the second element of all pairs? or Say an array of structures?

This is where we need comparators. To sort custom data types defined by users based on custom parameters.

Let's understand this by the example of vector of pairs where the *vector needs to sorted in ascending order of second element of all pairs.*

Below is a valid comparator for this:

```
bool sortbysec(const pair<int,int> &a,const pair<int,int> &b)
{
    return (a.second < b.second);
}
```

The above function accepts two **pairs** “a” and “b”, and returns true if second element of first pair is smaller than second element of second pair and if not then it returns false.

Code:

```
#include<iostream>#include<vector>#include<algorithm>using namespace std;

bool sortbysec(const pair<int,int> &a,const pair<int,int> &b)
{
    return (a.second < b.second);
}

int main() {

    vector<pair<int, int> > vec = {{10,3}, {20, 1}, {30, 2}};

    sort(vec.begin(), vec.end(), sortbysec);

    for(int i=0; i<3; i++)
    {
        cout<<vec[i].first<<" "<<vec[i].second<<"\n";
    }

    return 0;
}
```

Output:

```
20 1
30 2
10 3
```

min_element() in C++ STL

Problem Statement: Given a vector find the minimum element of the vector.

Example:

```
Example 1:Input: arr = {3,1,9,5,2}
Output: 1
Explanation: 1 is the minimum element.

Example 2:Input: arr = {10,40,22,5,2}
Output: 2
Explanation: 2 is the minimum element.
```

The minimum element can be found using the STL function ***min_element()**.

Syntax:

```
*min_element(first index,last index);
```

Code:

C++ Code

```
#include<bits/stdc++.h>using namespace std;

int main(){
    vector<int>v {4,2,5,9,1};
    cout<<"The elements in the vector are: ";
    for(int i=0;i<v.size();i++)
    {
        cout<<v[i]<<" ";
    }
    cout<<endl;

    cout<<"The minimum element is: "<<*min_element(v.begin(),v.end());
}
```

Output:

The elements in the vector are: 4 2 5 9 1The minimum element is: 1

max_element() in C++ STL

Problem Statement: Given a vector find the maximum element of the vector.

Example:

```
Example 1:Input: arr = {3,1,9,5,2}
Output: 9
Explanation: 9 is the maximum element.

Example 2:Input: arr = {10,40,22,5,2}
Output: 40
Explanation: 40 is the maximum element.
```

The maximum element can be found using the STL function ***max_element()**.

Syntax:

```
*max_element(first index,last index);
```

Code

```
#include<bits/stdc++.h>using namespace std;

int main(){
    vector<int>v {4,2,5,9,1};
    cout<<"The elements in the vector are: ";
    for(int i=0;i<v.size();i++)
    {
        cout<<v[i]<<" ";
    }
    cout<<endl;

    cout<<"The maximum element is: "<<*max_element(v.begin(),v.end());
}
```

Output:

The elements in the vector are: 4 2 5 9 1

The maximum element is: 9

next_permutation in C++ STL

next_permutation in STL is a built-in function which as the name suggests returns the next lexicographically greater permutation of the elements in the container passed to it as an argument.

Does it accept any parameters?

Yes, next_permutation() accepts two iterators (begin and end) of a container(example, an array or vector) as parameters and modifies the container to store the next lexicographically greater permutation of elements within the range [begin, end)

Does it return anything?

Yes, it returns true if a next lexicographically greater permutation is possible, otherwise, it returns false.

Syntax:

```
next_permutation(begin, end);

where, begin is a iterator pointing to 1st element
of the container.
and, end is an iterator pointing to just after the
last element of the container.
```

Example 1:

```
int arr[] = {1,2,3};

next_permutation(arr, arr+3);
```

Example 2:

```
vector<int> vec = {1,2,3};

next_permutation(vec.begin(), vec.end());
```

Implementation in Array

```
#include<iostream>
#include<vector>
#include<algorithm>

using namespace std;

int main() {
    int arr[] = {1,3,2};

    next_permutation(arr,arr+3);//using in-built function of C++

    cout<<arr[0]<<" "<<arr[1]<<" "<<arr[2];

    return 0;
}
Output:

2 1 3
```

Implementation in Vector

```
#include<iostream>
#include<vector>
#include<algorithm>

using namespace std;

int main() {
    vector<int> vec = {1,3,2};

    next_permutation(vec.begin(), vec.end());

    cout<<vec[0]<<" "<<vec[1]<<" "<<vec[2];

    return 0;
}
Output:

2 1 3
```

__builtin_popcount() in STL C++ : Count Set bits

Before we describe this function, can you think of a solution to the problem: “**Count Set bits in a given integer?**“, In simple words, count the number of bit positions in binary representation of an integer which is set to 1?

How will you solve this problem?

```
// Here is a solution

int count_setbits(int N)
{
   int cnt=0;

   while(N>0)
   {
       cnt+=(N&1);
       N=N>>1;
   }

   return cnt;
}
```

Above is a simple solution to the problem.

The solution works as follows:

- Until the given number is greater than zero, that is, until all bits in the number is not set to 0.
    - Keep on doing bitwise AND with the number 1, which has only first bit 1.
    - and, keep on shifting the bits by 1 place to right in the input number.
    - Simultaneously, keep on calculating count.

What if we could have solved this using a single line of code?

Yes, it’s possible using the function __builtin_popcount() in STL.

The function takes an unsigned integer as input parameter and returns the number of set bits present in that integer.

Syntax:

```
__builtin_popcount(int num);
```

Note: This function only works for unsigned or positive integers.

Code:

```
#include<iostream>#include<vector>#include<algorithm>using namespace std;

int main() {

    int n = 7;

    cout<<__builtin_popcount(n);

    return 0;
}
```

Output:

```
3
```

What if the integer is of the type "long long" and not fits in the range of int?

In that case, there is a separate function with slight variation in function name but serves exactly the same purpose. The function is __builtin_popcountll(). Notice the ll in the end of function name.

Below is the implementation of the above function:

Code:

```
#include<iostream>#include<vector>#include<algorithm>using namespace std;

int main() {

    long long n = 77777777777777;

    cout<<__builtin_popcountll(n);

    return 0;
}
```

Output:

```
23
```

Video Link —

https://www.youtube.com/watch?v=RRVYpIET_RU&t=1372s

Special Thanks — Striver