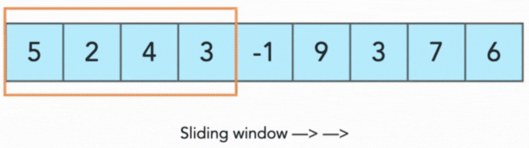


Sliding Window

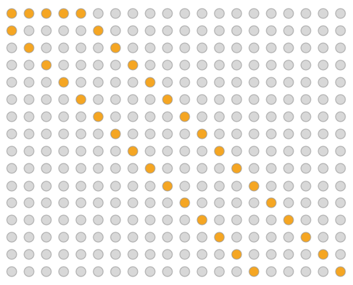
Tags	Algorithm Algorithm-Design Problem-Solving approach concept
Created time	@November 4, 2022 12:46 PM
Created time 1	@November 4, 2022 12:46 PM
Last edited time	@November 8, 2022 7:23 PM
Status	Completed
URL	
URL 1	

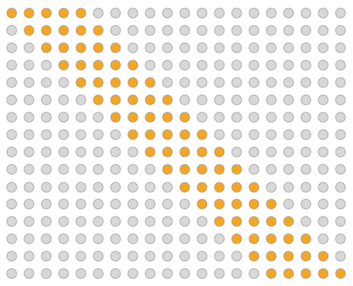
Window Sliding Technique is a computational technique which aims to reduce the use of nested loop and replace it with a single loop, thereby reducing the time complexity.

Where is it's useful? - it is used to in problem when we have to compute something in a contiguous sub array.

how does it works?

it used two pointer starting with first pointer start iteration doing calculation . when we archive a distance 'K' mostly given in problem it self. we start increasing second pointer as well so it always maintain 'K' distance. and before every time we increase second pointer we delete the calculation of the last element so we always have data of elements within the first and second pointer. the distance between both pointer is called the “window” and every time we are increasing pointers we are actually “sliding the window” over the array/list. it is mainly of two type that can be called fixed sized window and variable sized window. as the names suggests first type works on fixed sized window that is given in problem set. and in some advance problem we are given task to find a contiguous sub-array that meet a certain quality so we have to return the window that matches the quality.





Example questions (Fixed sized window) :-

▼ Max Sum of Sub-array

▼ Description -

Given an array of integers of size ‘n’. Our aim is to calculate the maximum sum of ‘k’ consecutive elements in the array.

```
Input  : arr[] = {100, 200, 300, 400}
k = 2
Output : 700
```

```
Input  : arr[] = {1, 4, 2, 10, 23, 3, 1, 0, 20}
k = 4
Output : 39
We get maximum sum by adding sub-array {4, 2, 10, 23}
of size 4.
```

```
Input  : arr[] = {2, 3}
k = 3
Output : Invalid
There is no sub-array of size 3 as size of whole
array is 2. .
```

▼ Code -

```
int maxSum(vector<int> &vec,int K)
{
  int ans{0},curSum{0},back{0},front{0};//declaration of variable
  while(front<vec.size())// loop from 0th index to the last index
  {
    curSum += vec[front];//adding front of the window
    if(front+1-back == K)//checking if we hit window size
    {
      ans = max(ans,curSum);
      curSum-=vec[back];//removing last element of window before sliding
      back++;//sliding window step 1
    }
    front++;//sliding window step 2
  }
  return ans;
}
```

▼ First negative integer in every window of size k🔗

▼ Description

Given an array **A[]** of size **N** and a positive integer **K**, find the first negative integer for each and every window(contiguous sub-array) of size **K.**

▼ Code

```
vector<long long> printFirstNegativeInteger(long long int A[],
                                             long long int N, long long int K){
  vector<long long> answer;
  list<long long> stk;
  int front{0},back{0};
  while(front<N)
  {
    if(A[front]<0)stk.push_back(A[front]);
    if(front+1-back==K)
    {
      if(stk.empty())answer.push_back(0);
      else answer.push_back(stk.front());
      if(stk.front()==A[back++])stk.pop_front();
    }
    front++;
  }
  return answer;
}
```

▼ Count Occurrences of Anagrams🔗

```
int search(string pat, string txt)
    {
      int K = pat.size();
      unordered_map<char,int> hash1;
      for(auto&&i:pat)hash1[i]++;
      int count = 0;
      int front = 0;
      int back = 0;
      unordered_map<char,int>hash2;
      while(front<txt.size())
      {
        hash2[txt[front]]++;
        if(front+1-back==K)
        {
          if(hash1==hash2)count++;
          hash2[txt[back]]--;
          if(hash2[txt[back]]==0) hash2.erase(txt[back]);
          back++;
        }
        front++;
      }
      return count;
    }
```

▼ Sliding Window Maximum🔗

```
vector<int> Solution::slidingMaximum(const vector<int> &A, int B) {
  vector<int> answer;
  int macs = INT_MIN;
  int back = 0;
  int front = 0;
  while(front < A.size())
  {
    macs = max(macs,A[front]);
    if(front+1-back==B)
    {
      answer.push_back(macs);
      back++;
    }
    front++;
    if(macs==A[back-1])
    {
      macs=INT_MIN;
      for (size_t i = back; i < front; i++)
      {
        macs = max(macs,A[i]);
      }

    }
  }
  return answer;
}
```

Example questions (variable sized window) :-

▼ Longest Sub-array of sum K

▼ Description -

Given an array containing N positive integers and an integer K. Your task is to find the length of the longest Sub-Array with sum of the elements equal to the given value K.

```
For Input:
1
7 5
4 1 1 1 2 3 5
your output is:
4 .
```

▼ Explanation -

start with two pointer(index pointer int variable not address variable) first and last and then keep two variable for sum and one for answer(initialise it with INT_MIN so when we check max() first time it return true) start a keep incrementing first and add array[first] variable until sum is equal to K when it does store it in ans after checking max with previous ans. then check if sum is bigger then K subtract A[back- -] until it is smaller again. in the end return ans.

▼ Code -

```
int longestOfK(vector<int> &vec, int K)
{
  int f{0},l{0},sum{0},ans{0};
  while (f < vec.size())
  {
    sum += vec[f];

    if (sum == K)
      ans = max(ans, f - l + 1);

    while (sum > K)
      sum -= vec[l++];

    f++;
  }
  return ans;
}
```

▼ Longest K unique characters sub-string 🔗

▼ Description -

Given a string you need to print the size of the longest possible substring that has exactly K unique characters. If there is no possible substring then print -1.

▼ Code -

```
int longestKSubstr(string& str,int k)
    {
      int ans = -1;
      int i{0},j{0},uc{0};
      unordered_map<char,int> hash;
      while(i<str.size())
      {
        hash[str[i]]++;
        if(hash[str[i]]==1)uc++;
        if(uc==k)ans=max(ans,i+1-j);
        while(uc>k){
          hash[str[j]]--;
          if(hash[str[j]]==0)uc--;
          j++;
        }
        i++;
      }
      return ans;
    }
```

▼ Longest Sub-string Without Repeating Characters 🔗

▼ Description -

Given a string s, find the length of the longest sub-string without repeating characters.

▼ Code -

```
  int lengthOfLongestSubstring(string s) {
    int front{0},back{0},ans{0},uc{0};
    unordered_map<char,int> hash;
    while(front<s.size())
    {
      hash[s[front]]++;
      if(hash[s[front]]==1)uc++;
      if(uc==front+1-back) ans = max(ans, uc);
```

```
        while(uc<front+1-back)
        {
          hash[s[back]]--;
          if(hash[s[back]]==0)uc--;
          back++;
        }
        front++;
      }
      return ans;
      }
```

▼ Minimum Window Sub-string 🔗

▼ Code

```
string minWindow(string s, string t) {
        map<char,int> hash;
        for(auto&&i:t)hash[i]++;
        int uc = hash.size();
        int front{0},back{0},ans{INT_MAX},ansIdx{0};
        while(front<s.size()){
            if(hash.find(s[front]) != hash.end()){
                hash[s[front]]--;
                if(hash[s[front]]==0)uc--;
            }
            while (uc==0)
            {
                if(ans>front+1-back){
                    ans=front+1-back;
                    ansIdx=back;
                }
                if(hash.find(s[back]) != hash.end()){
                    hash[s[back]]++;
                    if(hash[s[back]]==1)uc++;
                }
                back++;
            }
            front++;

        }
        return ((ans==INT_MAX)?"":s.substr(ansIdx,ans));
    }
```