MODUL 5 – AN INTRO TO ALGORITHM

DASAR PEMROGRAMAN 2019

TIM ASDOS DASAR PEMROGRAMAN 2019

INFORMATIKA ITS

DAFTAR ISI

1. SEARCHING

Searching merupakan hal mendasar yang sering digunakan untuk pada pemrograman. Seringkali permasalahan menuntut kita untuk melakukan pencarian suatu item/elemen tertentu pada suatu domain, misalkan array.

1.1. Linear Search

Algoritma searching paling mudah untuk dipahami dan diimplementasikan adalah *Linear Search* atau biasa juga disebut dengan *Sequential Search*. *Linear search* bekerja dengan melakukan pengecekkan kepada semua elemen yang ada.

Secara garis besar, cara kerja *linear search* adalah :

- Memeriksa item satu per satu.
- Apabila ditemukan, maka "ketemu".
- Jika sampai akhir belum ditemukan, maka item yang dicari tidak ada.

Implementasi *Linear Search*

Kode Program 1.1.1 Implementasi Linear Search

```
1. int linearSearch(int arr[], int n, int item)
2. {
3.     int i;
4.     for (i = 0; i < n; ++i) {
5.         if (item == arr[i]) return 1;
6.     }
7.     return -1;
8. }
```

1.2. Binary Search

Kasus terburuk dari penggunaan *linear search* adalah melakukan N kali perbandingan pada data (N adalah banyaknya data). Teknik *linear search* tidak akan efisien apabila jumlah datanya sangat besar. Lalu apakah ada cara yang lebih efisien?

Binary Search adalah salah satu cara yang lebih efisien daripada *linear search*. Jika pada linear search kasus terburuknya adalah melakukan N perbandingan,

maka dalam *binary search* kasus terburuknya adalah hanya melakukan sebanyak $\log N$ perbandingan saja.

Salah satu hal yang perlu diperhatikan dalam *binary search* adalah, data yang akan dicari harus **terurut** (*ascending* atau *descending*). Properti ini mutlak harus terpenuhi agar *binary search* bisa bekerja.

Bagaimana *binary search* bekerja ?

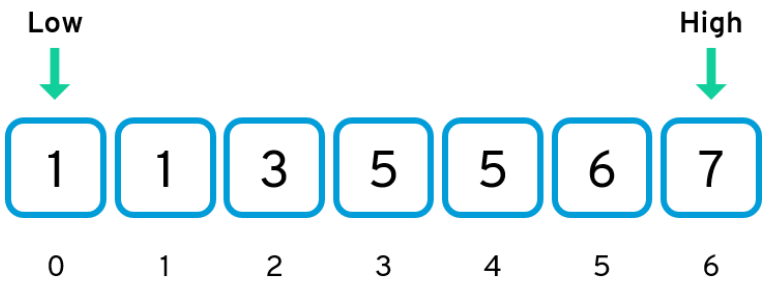
Ide utama dalam *binary search* adalah dengan mempersempit ruang pencarian agar menjadi separuhnya dalam setiap iterasi. Selain itu, terdapat tiga properti utama, yakni indeks untuk low, $high$, dan mid.

1. Set nilai low menjadi indeks terkecil, dan $high$ menjadi indeks terbesar.
2. Mulai dengan membandingkan nilai tengah ($mid = (low + high)/2$.
3. Apabila nilai/indeks $mid <$ item yang dicari, maka naikkan posisi low menjadi $low = mid + 1$. Sebaliknya, apabila nilai/indeks $mid >$ item yang dicari, maka turunkan posisi $high$ menjadi $high = mid - 1$. Jika, nilai/indeks mid sama seperti item yang dicari, selesai (item ditemukan). Jika tidak, ulangi proses nomor 2.
4. Lakukan pencarian hingga $low > high$.

Ilustrasi *Binary Search*

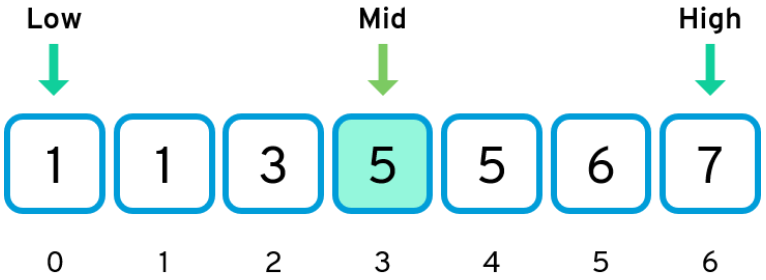
Misalkan kita ingin mencari angka 3 pada array $A = [1, 1, 3, 5, 5, 6, 7]$

- Inisialisasi nilai $low = 0$ dan $high = 6$.



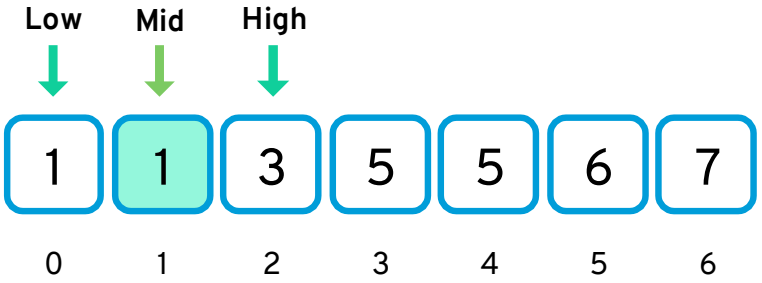
Gambar 1.2.1 Ilustrasi Binary Search

- Hitung nilai $mid = (0 + 6)/2 = 3$. Periksa apakah indeks 3 adalah nilai yang dicari.



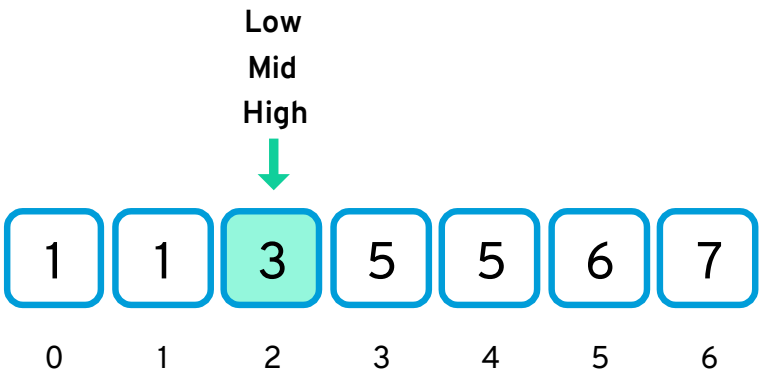
Gambar 1.2.2 Ilustrasi Binary Search

- Ternyata $A[3] > 3$, maka turunkan $high$ menjadi $high = 3 - 1 = 2$. Hitung nilai $mid = (0 + 2)/2 = 1$. Periksa apakah $A[1] = 3$.



Gambar 1.2.3 Ilustrasi Binary Search

- Ternyata, $A[1] < 3$, naikkan nilai low menjadi $low = mid + 1 = 1 + 1 = 2$. Hitung nilai $mid = (2 + 2)/2 = 2$.



Gambar 1.2.4 Ilustrasi Binary Search

- Periksa apakah $A[2] = 3$. Karena $A[2] = 3$, maka pencarian berhenti. Data telah ditemukan pada indeks ke-3.

Implementasi *Binary Search*

Kode Program 1.2.1 Implementasi Binary Search

```
1.  int binarySearch(int arr[], int n, int item)
2.  {
3.      int low = 0, high = n-1, mid;
4.      while (low <= high) {
5.          mid = (low + high)/2;
6.          if (arr[mid] < item)
7.              low = mid + 1;
8.          else if (arr[mid] > item)
9.              high = mid - 1;
10.         else return mid;
11.     }
12.     return -1; // not found
13. }
```

2. SORTING

Sorting atau pengurutan seringkali dilakukan pada pemrograman agar data lebih mudah untuk diolah. Pengurutan seringkali dilakukan pada kumpulan data, biasanya adalah array, dan dapat dilakukan dalam dua urutan, yakni *ascending* atau *descending*. Terdapat bermacam-macam algoritma yang dapat digunakan untuk *sorting*. Beberapa akan dijelaskan sebagai berikut.

2.1. Bubble Sort

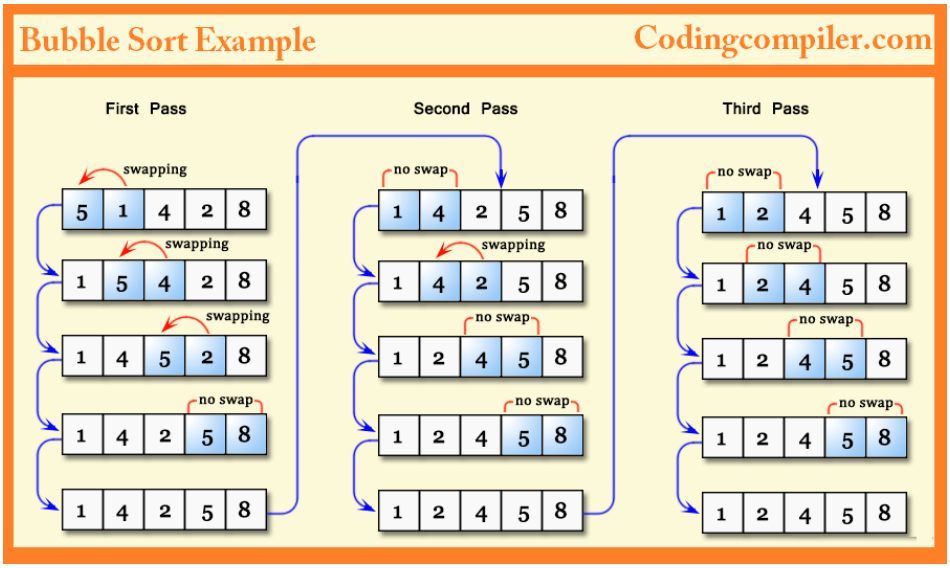
Algoritma *bubble sort* ini merupakan proses pengurutan yang secara berangsur-angsur berpindah ke posisi yang tepat, karena itulah dinamakan *Bubble* yang artinya gelembung. Algoritma ini akan mengurutkan data dari yang terbesar ke yang terkecil (ascending) atau sebaliknya (descending).

Secara sederhana, algoritma *bubble sort* adalah pengurutan dengan cara pertukaran data dengan data disebelahnya secara terus menerus sampai dalam satu iterasi tertentu tidak ada lagi perubahan.

Bubble Sort adalah metode *sorting* yang sederhana. Namun merupakan metode pengurutan yang tidak efisien karena ketika mengurutkan data yang sangat besar akan sangat lambat prosesnya.

Pada dasarnya, sifat algoritma *bubble sort* adalah sebagai berikut:

1. Jumlah iterasi sama dengan banyaknya bilangan dikurang 1.
2. Di setiap iterasi, jumlah pertukaran bilangannya sama dengan jumlah banyaknya bilangan.
3. Dalam algoritma Bubble Sort, meskipun deretan bilangan tersebut sudah terurut, proses sorting akan tetap dilakukan.
4. Tidak ada perbedaan cara yang berarti untuk teknik algoritma Bubble Sort Ascending dan Descending.



Gambar 2.1.1 Ilustrasi Bubble Sort

Kita dapat mengoptimasinya dengan memberikan variable *bool swapped* dengan implementasi sebagai berikut:

Implementasi Bubble Sort

Kode 2.1.1 Implementas *Bubble Sort*

```
1.  void swap(int *xp, int *yp)
2.  {
3.      int temp = *xp;
4.      *xp = *yp;
5.      *yp = temp;
6.  }
7.
8.  // Bubble Sort yang telah terooptimase dengan bool swapped
9.  void bubbleSort(int arr[], int n)
10. {
11.     int i, j;
12.     bool swapped;
13.     for (i = 0; i < n-1; i++)  {
14.         swapped = false;
15.         for (j = 0; j < n-i-1; j++) {
16.             if (arr[j] > arr[j+1]) {
17.                 swap(&arr[j], &arr[j+1]);
18.                 swapped = true;
19.             }
20.         }
```

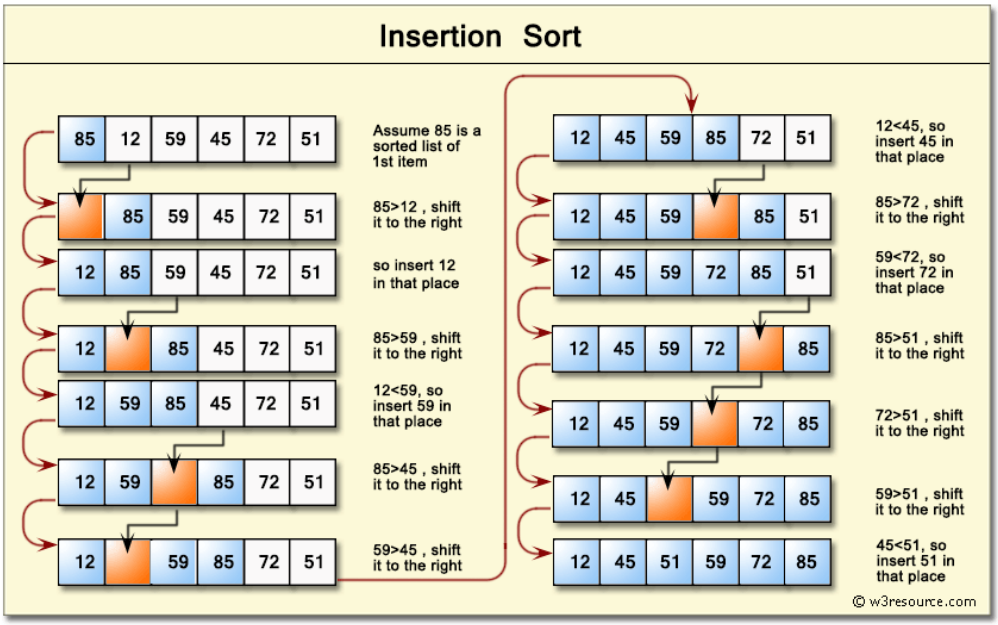
```
21.
22.         if (swapped == false)
23.             break;
24.     }
25. }
```

2.2. Insertion Sort

Insertion Sort merupakan salah satu jenis sort dimana sort ini mirip seperti saat kalian melakukan pengurutan kartu remi di tangan. Sort jenis ini dilakukan dengan membandingkan dua elemen data pertama, mengurutkannya, kemudian mengecek elemen data berikutnya satu persatu dan membandingkannya dengan elemen data yang telah diurutkan. Ide dasar dari sort ini adalah mencari tempat yang "tepat" untuk setiap elemen array.

Secara umum, langkah-langkah algoritma insertion sort adalah sebagai berikut.

1. Ambil elemen dari array ke-i.
2. Taruh di urutan sebelumnya (arr[0…i-1]) yang telah diurutkan di tempat yang sesuai.



Gambar 2.2.1 Ilustrasi Insertion Sort

Implementasi Insertion Sort

Kode 2.2.1 Implementasi Insertion Sort

```
1. //n adalah ukuran dari array
2. void insertionSort(int arr[], int n)
3. {
4.     int i, key, j;
5.     for (i = 1; i < n; i++) {
6.         key = arr[i];
7.         j = i - 1;
8.
9.         while (j >= 0 && arr[j] > key) {
10.            arr[j + 1] = arr[j];
11.            j = j - 1;
12.        }
13.        arr[j + 1] = key;
14.    }
15. }
```

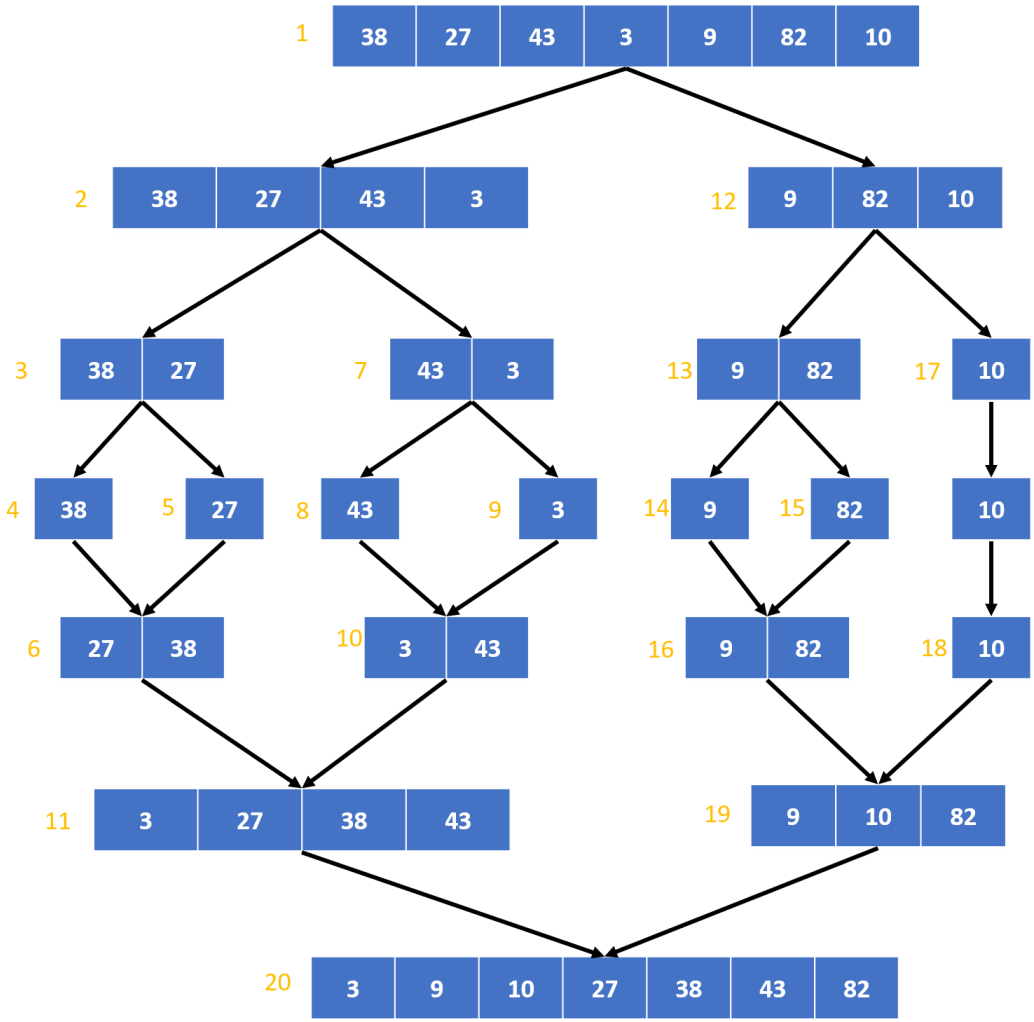
2.3. Merge Sort

Merge Sort merupakan algoritma Divide and Conquer. Merge sort membagi array inputnya menjadi 2 bagian, memanggil dirinya sendiri untuk masing-masing bagian array dan menggabungkan dua bagian yang telah terurut. Algoritma Merge Sort adalah sebagai berikut.

```
mergeSort(arr[], l, r)

jika r > l

   1. Cari titik tengah untuk membagi array menjadi 2 bagian:
      titik tengah m = (l+r) /2
   2. Panggil mergeSort untuk bagian pertama:
      mergeSort(arr, l, m)
   3. Panggila mergeSort untuk bagian kedua:
      mergeSort(arr, m+1, r)
   4. Gabungkan 2 bagian yang telah terurut pada langkah 2
      dan 3.
      merge(arr, l, m, r)
```



Dari diagram dapat dilihat bahwa array input secara rekursif dibagi hingga berukuran 1, baru kemudian proses merge dilakukan hingga diperoleh array seperti semula yang telah terurut.

Implementasi Merge Sort

```
1. void mergeSort(int arr[], int lo, int hi)
2. {
3.     if(hi > lo){
4.         int m = (lo + hi) / 2;
5.
6.         mergeSort(arr, lo, m);
7.         mergeSort(arr, m+1, hi);
8.
```

```
9.          merge(arr, lo, m, hi);
10.     }
11. }
12.
13. void merge(int arr[], int lo, int m, int hi){
14.     int i, ki, ka, mr;
15.
16.     // Ukuran bagian kiri dan kanan
17.     int n_kiri = m - lo + 1;
18.     int n_kanan = hi - m;
19.
20.     // Array Temporer untuk bagian kiri dan kanan
21.     int Kiri[n_kiri], Kanan[n_kanan];
22.
23.     // Copy data ke array temporer
24.     for(i=0; i<n_kiri; i++){
25.         Kiri[i] = arr[lo+i];
26.     }
27.     for(i=0; i<n_kanan; i++){
28.         Kanan[i] = arr[m+1+i];
29.     }
30.
31.     // PROSES MERGE
32.     ki = 0; // index awal untuk bagian kiri;
33.     ka = 0; // index awal untuk bagian kanan;
34.     mr = lo; // index awal untuk array yang sudah terurut;
35.
36.     while(ki < n_kiri && ka < n_kanan){
37.         if(Kiri[ki] <= Kanan[ka]){
38.             arr[mr] = Kiri[ki];
39.             ki++;
40.         }
41.         else{
42.             arr[mr] = Kanan[ka];
43.             ka++;
44.         }
45.         mr++;
46.     }
47.
48.     // Jika bagian kiri masih sisa
49.     while(ki < n_kiri){
50.         arr[mr] = Kiri[ki];
51.         ki++;
52.         mr++;
53.     }
54.
55.     // Jika bagian kanan masih sisa
56.     while(ka < n_kanan){
```

```
57.         arr[mr] = Kanan[ka];
58.         ka++;
59.         mr++;
60.     }
61. }
```

3. BASIC NUMBER THEORY

3.1. Bilangan Prima

3.1.1. Definisi Bilangan Prima

Bilangan Prima adalah bilangan bulat lebih dari 1 yang hanya memiliki dua faktor, yakni 1 dan dirinya sendiri. Contoh beberapa bilangan prima adalah 2, 3, 5, 7, 11 dst. Sedangkan lawan dari bilangan prima adalah bilangan komposit. Bilangan komposit adalah bilangan yang memiliki lebih dari dua faktor.

3.1.2. Primality Testing

Primality Testing adalah aktivitas yang dilakukan untuk menguji apakah suatu bilangan N merupakan bilangan prima atau tidak. Solusi paling mudah untuk melakukan pengujian keprimaan adalah dengan melakukan pengecekkan apakah ada bilangan selain 1 dan N yang membagi N. Hal ini dapat dilakukan dengan melakukan perulangan dari 2 sampai $N - 1$. Metode ini disebut dengan ***trial division***.

Implementasi Trial Division

Kode 3.1.1 Implementasi Trial Division

```
1. #include <stdbool.h>
2.
3. bool isPrime(int N)
4. {
5.     int i;
6.     for (i = 2; i < N; ++i) {
7.         if (N % i == 0) return false;
8.     }
9.     return true;
10. }
```

Solusi di atas tidak akan efisien apabila bilangan yang hendak diuji sangat besar. Lalu apakah ada solusi yang lebih efisien? Mari kita lakukan observasi terhadap N.

Bilangan N dapat kita tulis sebagai perkalian dua bilangan $N = a \times b$. Jika N habis dibagi a, maka tentu N habis dibagi (N/a) atau b.

Kita ambil contoh $N = 18$. N dapat dituliskan sebagai perkalian dua bilangan $a \times b$.

a	b
1	18
2	9
3	6

Dapat diperhatikan, nilai dari a selalu $\leq \sqrt{18} \cong 4$ dan nilai b selalu $\geq \sqrt{18}$. Jika 18 habis dibagi 2, maka 18 habis (18/2) atau 9 dst. Kita dapat mengabaikan 1 dan 18, dan hanya perlu mengecek dari 2 hingga $\sqrt{18}$ saja.

Maka dari itu, kita hanya perlu melakukan pengecekkan dari 2 sampai $\sqrt{N}$ saja.

Implementasi Optimized Trial Division

Kode 3.1.2 Implementasi SQRT Trial Division

```
1.  #include <stdbool.h>
2.  #include <math.h>
3.
4.  bool isPrimeSQRT(int N)
5.  {
6.      int i;
7.      for (i = 2; i <= (int) sqrt(N); ++i) {
8.          if (N % i == 0) return false;
9.      }
10.     return true;
11. }
```

3.1.3. Prime Generation

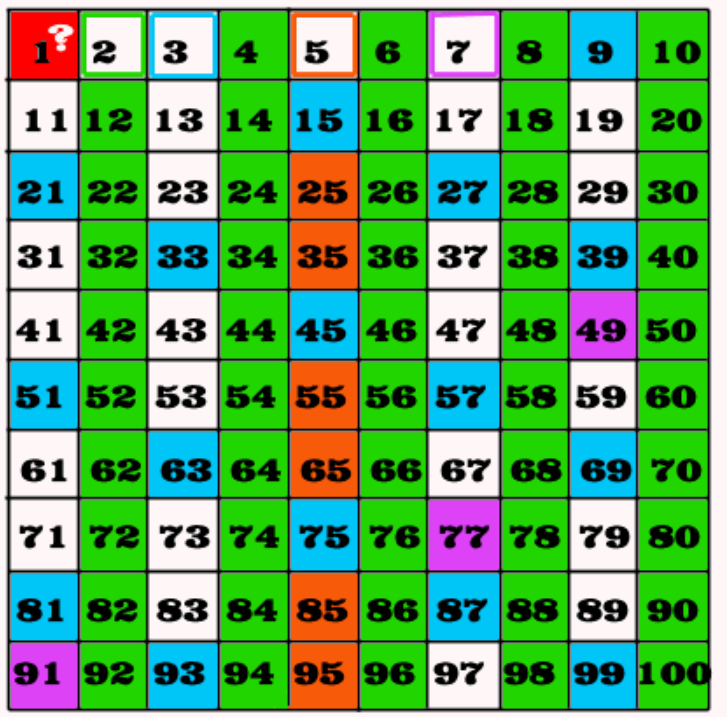
Prime Generation adalah aktivitas untuk melakukan pembangkitan (*generate*) bilangan prima, sehingga tidak perlu melakukan komputasi berulang kali untuk setiap bilangan hingga N.

Sieve of Eratosthenes

Metode yang paling umum digunakan untuk melakukan generasi bilangan prima adalah ***Sieve of Eratosthenes* (saringan Eratosthenes).** Ide utama dari metode ini adalah mengeliminasi calon-calon bilangan prima, yakni bilangan komposit.

Langkah-langkah untuk melakukan generasi bilangan prima dari 2 hingga N adalah sebagai berikut.

1. Awalnya, tandai seluruh bilangan 2 hingga N sebagai prima.
2. Lakukan iterasi dari 2 hingga N :
    a. Jika bilangan ini prima (belum dieliminasi), maka :
    b. Eliminasi kelipatan bilangan ini dengan menandai seluruh kelipatannya sebagai bukan prima.



Gambar 3.1.1 Ilustrasi Sieve of Eratosthenes hingga 100

Untuk mengimplementasikan ***Sieve of Eratosthenes***, digunakan array of boolean dengan ukuran $N + 1$ dan menginisialisasi semua nilainya menjadi *true*.

Implementasi *Sieve of Eratosthenes*

Kode 3.1.3 Implementasi Sieve of Eratosthenes

```
1. #include <stdbool.h>
2.
3. void sieveOfEratosthenes(int N, bool isPrime[])
4. {
5.     int i, j;
```

```
6.      // Awalnya semua nilai pada array isPrime adalah true
7.      for (i = 2; i <= N; i++) {
8.          if (isPrime[i] == true) {
9.              for (j = i*i; j <= N; j += i)
10.                 isPrime[j] = false;
11.         }
12.     }
13. }
```

3.2. FPB dan KPK

Pasti kita tidak asing dengan kedua istilah di atas (FPB dan KPK). FPB atau Faktor Persekutuan Terbesar (atau dalam bahasa inggris disebut GCD) dari bilangan a dan b adalah bilangan terbesar yang masing-masing membagi a dan b. Contoh $FPB(8,4) = 4$.

- Faktor dari $8 = \{1,2,\mathbf{4},8\}$
- Faktor dari $4 = \{1,2,\mathbf{4}\}$

Dapat diperhatikan bahwa bilangan terbesar yang sama-sama membagi 8 dan 4 adalah 4. Bagaimana cara untuk menghitung FPB ?

FPB dapat dihitung menggunakan algoritma Euclid sebagai berikut.

$$FPB(a,b) = \begin{cases} a, & \text{if } b = 0 \\ FPB(b, a \bmod b), & \text{otherwise} \end{cases}$$

KPK atau Kelipatan Persekutuan Terkecil (atau dalam bahasa inggris disebut LCM) dari bilangan a dan b merupakan bilangan bilangan positif terkecil yang sama-sama habis dibagi oleh a dan b. KPK dapat dihitung menggunakan FPB :

$$KPK(a,b) = \frac{a \times b}{FPB(a,b)}$$

Implementasi FPB dan KPK

Kode Program 3.2.1 Implementasi FPB dan KPK

```
1.  int fpb(int a, int b)
2.  {
3.      if (b == 0) return a;
4.      return fpb(b, a % b);
5.  }
6.
```

```
7.  int kpk(int a, int b)
8.  {
9.      return (a * b)/fpb(a,b);
10. }
```

3.3. Modular Arithmetic

Modulo merupakan operasi dasar dalam matematika yang digunakan untuk menghitung sisa hasil bagi suatu bilangan terhadap suatu bilangan lain. Contoh :

- $5 \; mod \; 2 = 1$
- $11 \; mod \; 4 = 7$

3.3.1. Operasi Dasar

Sifat-sifat dasar dalam operasi aritmetika modulo adalah sebagai berikut.

- $(a + b) \bmod m = ((a \bmod m) + (b \bmod m)) \bmod m$
- $(a - b) \bmod m = ((a \bmod m) - (b \bmod m)) \bmod m$
- $(a \times b) \bmod m = ((a \bmod m) \times (b \bmod m)) \bmod m$
- $a^b \bmod m = (a \bmod m)^b \bmod m$
- $(-a) \bmod m = (-(a \bmod m) + m) \bmod m$

Untuk pembagian, sifat yang digunakan berbeda. Biasa ditulis dalam notasi berikut.

- $\left(\frac{a}{b}\right) \bmod m = (a \times b^{-1}) \bmod m$

dengan b^{-1} adalah **modular multiplicative inverse** dari b. Modular multiplicative inverse tidak akan dibahas pada modul ini.

3.3.2. Modular Exponentiation

Modular Exponentiation seperti namanya, adalah melakukan operasi perpangkatan dalam modulus tertentu, atau menghitung $a^b \bmod m$. Cara paling mudah untuk menghitung $a^b \bmod m$ adalah dengan menghitung a^b terlebih dahulu kemudian melakukan modulo terhadap m. Contoh :

- $4^{13} \bmod 497 = 67.108.864 \bmod 497 = 445$

Cara manual seperti di atas tidak akan pernah efisien untuk angka yang sangat besar, ditambah lagi bila angkanya sangat besar, kemungkinan tidak ada tipe data yang dapat menampung nilai eksponennya.

Terdapat cara yang lebih efisien untuk menghitung $a^b \bmod m$, yakni dengan menggunakan konsep *binary exponentiation*. Berikut adalah implementasi modular exponentitation secara iteratif.

Implementasi Modular Exponentiation

Kode 3.3.1 Implementasi Mod. Exp. menggunakan konsep Binary Exponentiation

```
1.  int modularExponentiation(int a, int b, int m)
2.  {
3.      int result = 1;
4.      while (b > 0)
5.      {
6.          if (power % 2 == 1)
7.              result = (result * a) % m;
8.          a = (a*a) % m;
9.          b= b/2;
10.     }
11.     return result;
12. }
```

Note: Binary exponentiation tidak akan dijelaskan pada modul ini.