Chapter 3
Sorting

Sorting atau pengurutan, adalah masalah mendasar dalam desain algoritma. Banyak algoritma yang efisien menggunakan sorting sebagai subrutin, karena sering kali lebih mudah memproses data jika elemen-elemen dalam keadaan terurut.

Sebagai contoh, masalah *"apakah sebuah array mengandung dua elemen yang sama?"* dapat diselesaikan dengan mudah menggunakan sorting. Jika array mengandung dua elemen yang sama, elemen-elemen tersebut akan berada bersebelahan setelah disorting, sehingga mudah ditemukan. Selain itu, masalah *"elemen mana yang paling sering muncul dalam array?"* juga dapat diselesaikan dengan cara yang serupa.

Ada banyak algoritma sorting, dan algoritma-algoritma ini juga merupakan contoh yang baik tentang bagaimana menerapkan berbagai teknik desain algoritma. Algoritma sorting umum yang efisien bekerja dalam waktu $O(n \log n)$, dan banyak algoritma yang menggunakan sorting sebagai subrutin juga memiliki kompleksitas waktu yang sama.

Teori Sorting

Masalah dasar dalam sorting adalah sebagai berikut:

> Diberikan sebuah array yang berisi n elemen, tugas Anda adalah mengurutkan elemen-elemen tersebut dalam urutan naik.

Sebagai contoh, array

1	3	8	2	9	2	5	6

akan menjadi seperti berikut setelah disorting:

1	2	2	3	5	6	8	9

Algoritma $\mathbf{O}(\boldsymbol{n^2})$

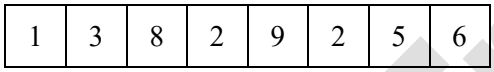
Algoritma sederhana untuk mengurutkan sebuah array bekerja dalam waktu $O(n^2)$. Algoritma semacam itu pendek dan biasanya terdiri dari dua loop bersarang (*nested loops*). Salah satu algoritma sorting terkenal dengan kompleksitas $O(n^2)$ adalah **bubble sort**, di mana elemen-elemen dalam array akan "menggelembung" sesuai dengan nilainya.

Bubble sort terdiri dari n putaran (*round*). Pada setiap putaran, algoritma akan menelusuri elemen-elemen dalam array. Jika ditemukan dua elemen berurutan yang tidak dalam urutan yang benar, algoritma akan menukar (*swap*) keduanya. Algoritma ini dapat diimplementasikan sebagai berikut:

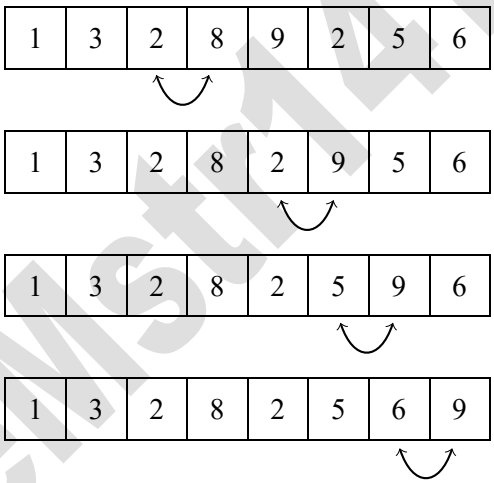
```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n-1; j++) {
        if (array[j] > array[j+1]) {
            swap(array[j], array[j+1]);
        }
    }
}
```

Setelah putaran pertama, elemen terbesar akan berada di posisi yang benar. Secara umum, setelah k putaran, k elemen terbesar akan berada di posisi yang benar. Dengan demikian, setelah n putaran, seluruh array akan terurut.

Sebagai contoh, dalam array



putaran pertama dari bubble sort menukar elemen sebagai berikut:



Inversions

Bubble sort adalah contoh algoritma sorting yang selalu menukar (*swap*) elemen-elemen yang berdekatan dalam array. Ternyata, kompleksitas waktu algoritma semacam itu selalu setidaknya $O(n^2)$, karena dalam kasus terburuk, diperlukan $O(n^2)$ pertukaran untuk mengurutkan array.

Konsep yang berguna dalam menganalisis algoritma sorting adalah **inversion** (*inversi*), yaitu pasangan elemen dalam array ($\texttt{array}[a]$, $\texttt{array}[b]$) yang memenuhi kondisi:

- $a < b$
- $\texttt{array}[a] > \texttt{array}[b]$

Dengan kata lain, elemen-elemen dalam pasangan tersebut berada dalam urutan yang salah. Contohnya, dalam array berikut:

1	2	2	6	3	5	9	8

Array berikut memiliki tiga inversi: (6,3), (6,5), dan (9,8). Jumlah inversi menunjukkan seberapa banyak pekerjaan yang dibutuhkan untuk mengurutkan array.

- **Array sepenuhnya terurut** jika tidak ada inversi.
- **Jika elemen array dalam urutan terbalik**, jumlah inversi akan mencapai nilai maksimum.

Sebagai contoh, dalam array dengan elemen dalam urutan terbalik, setiap pasangan elemen akan membentuk inversi, menghasilkan jumlah inversi maksimum yang mungkin.

$$1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2} = O(n^2)$$

Menukar sepasang elemen berurutan yang berada dalam urutan yang salah akan menghapus tepat satu inversi dari array. Oleh karena itu, jika suatu algoritma sorting hanya dapat menukar elemen yang berdekatan, setiap pertukaran menghapus paling banyak satu inversi. Akibatnya, dalam kasus terburuk ketika terdapat $O(n^2)$ inversi, algoritma akan membutuhkan $O(n^2)$ pertukaran untuk menyortir array, sehingga kompleksitas waktu minimalnya adalah $O(n^2)$.

Algoritma $O(n \log n)$

Kita dapat menyortir sebuah array secara efisien dalam $O(n \log n)$ waktu dengan menggunakan algoritma yang tidak terbatas pada penukaran elemen yang berdekatan. Salah satu algoritma yang memenuhi kriteria ini adalah **merge sort**, yang berbasis pada rekursi.

Merge sort menyortir subarray $\texttt{array}[a \ldots b]$ dengan langkah-langkah berikut:

1. **Basis rekursi**: Jika $a == b$, tidak perlu melakukan apa pun, karena subarray dengan satu elemen sudah tersortir.
2. **Tentukan titik tengah**: Hitung posisi elemen tengah dengan rumus: $k = \lfloor (a+b)/2 \rfloor$.
3. **Rekursif**:
   - Urutkan subarray $\texttt{array}[a \ldots k]$.
   - Urutkan subarray $\texttt{array}[k+1 \ldots b]$.
4. **Penggabungan (Merging)**: Gabungkan dua subarray ($\texttt{array}[a \ldots k]$ dan $\texttt{array}[k+1 \ldots b]$) yang telah tersortir menjadi satu subarray $\texttt{array}[a \ldots b]$ yang juga tersortir.

Merge sort adalah algoritma yang efisien karena pada setiap langkah, ukuran subarray dibagi dua. Rekursinya terdiri dari $O(\log n)$ level, dan pemrosesan setiap level memerlukan waktu $O(n)$. Penggabungan subarray $\texttt{array}[a \ldots k]$ dan $\texttt{array}[k+1 \ldots b]$ dapat dilakukan dalam waktu linear karena keduanya sudah tersortir.

Sebagai contoh, pertimbangkan untuk mengurutkan array berikut ini:

1	3	6	2	8	2	5	9

Array akan dibagi menjadi dua sub-array sebagai berikut:

1	3	6	2

8	2	5	9

Kemudian, sub-array akan diurutkan secara rekursif sebagai berikut:

1	2	3	6

2	5	8	9

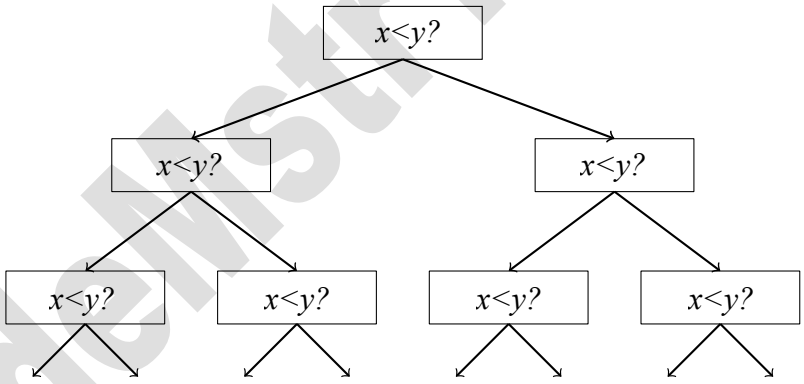
Terakhir, algoritma menggabungkan sub-array yang diurutkan dan menciptakan array terakhir yang terurut:

1	2	2	3	5	6	8	9

Batas bawah penyortiran

Apakah mungkin untuk menyortir sebuah array lebih cepat dari waktu $O(n\ log\ n)$? Ternyata ini tidak mungkin jika kita membatasi diri pada algoritma penyortiran yang didasarkan pada perbandingan elemen-elemen array.

Batas bawah untuk kompleksitas waktu dapat dibuktikan dengan mempertimbangkan penyortiran sebagai sebuah proses di mana setiap perbandingan antara dua elemen memberikan lebih banyak informasi tentang isi array. Proses ini menghasilkan tree berikut:



Di sini, "$x < y$?" berarti bahwa beberapa elemen x dan y dibandingkan. Jika $x < y$, proses berlanjut ke kiri, dan sebaliknya ke kanan. Hasil dari proses ini adalah kemungkinan cara untuk menyortir array, dengan total sebanyak $n!$ cara. Karena alasan ini, tinggi tree harus setidaknya

$$log_2(n!) = log_2(1) + log_2(2) + \cdots + log_2(n)$$

Kita mendapatkan batas bawah untuk jumlah ini dengan memilih $n/2$ elemen terakhir dan mengubah nilai setiap elemen menjadi $log_2(n/2)$. Ini menghasilkan perkiraan

$$log_2(n!) \geq (n/2) \cdot log_2(n/2)$$

Jadi, tinggi tree dan jumlah langkah minimum yang mungkin dalam sebuah algoritma sorting dalam kasus terburuk setidaknya sebesar $n\ log\ n$.

Counting Sort

Batas bawah $n \log n$ tidak berlaku untuk algoritma yang tidak membandingkan elemen array, tetapi menggunakan informasi lain. Salah satu contoh algoritma semacam itu adalah **counting sort**, yang dapat mengurutkan array dalam waktu $O(n)$ dengan asumsi bahwa setiap elemen dalam array adalah bilangan bulat dalam rentang $0 \ldots c$ dan $c = O(n)$.

Algoritma ini membuat sebuah *bookkeeping array* (array pencatatan), di mana indeksnya mewakili elemen-elemen dari array asli. Kemudian, algoritma akan melakukan iterasi melalui array asli dan menghitung berapa kali setiap elemen muncul dalam array.

Sebagai contoh, dari array

1	3	6	9	9	3	5	9

sesuai dengan bookkeeping array berikut ini:

1	2	3	4	5	6	7	8	9
1	0	2	0	1	1	0	0	3

Misalnya, nilai pada posisi 3 dalam array pencacahan adalah 2, karena elemen 3 muncul 2 kali dalam array asli.

Pembangunan bookkeeping array memerlukan waktu $O(n)$. Setelah itu, array yang telah diurutkan dapat dibuat dalam waktu $O(n)$ karena jumlah kemunculan setiap elemen dapat diperoleh dari bookkeeping array. Dengan demikian, kompleksitas waktu total counting sort adalah $O(n)$.

Counting sort adalah algoritma yang sangat efisien, tetapi hanya dapat digunakan ketika konstanta c cukup kecil, sehingga elemen dalam array dapat digunakan sebagai indeks dalam bookkeeping array.

Sorting dalam C++

Hampir tidak pernah menjadi ide yang baik untuk menggunakan algoritma sorting buatan sendiri dalam sebuah kontes, karena sudah ada implementasi yang baik yang tersedia dalam bahasa pemrograman. Sebagai contoh, pustaka standar C++ memiliki fungsi `sort` yang dapat digunakan dengan mudah untuk mengurutkan array dan struktur data lainnya.

Ada banyak keuntungan dalam menggunakan fungsi pustaka. Pertama, ini menghemat waktu karena tidak perlu mengimplementasikan fungsi sendiri. Kedua, implementasi pustaka sudah pasti benar dan efisien: kecil kemungkinan bahwa fungsi sorting buatan sendiri akan lebih baik.

Di bagian ini, kita akan melihat bagaimana menggunakan fungsi `sort` dalam C++. Kode berikut mengurutkan sebuah vektor dalam urutan menaik:

```
vector<int> v = {4, 2, 5, 3, 5, 8, 3};
sort(v.begin(), v.end());
```

Setelah sorting, isi vektor akan menjadi [2,3,3,4,5,5,8]. Urutan sorting default adalah menaik, tetapi urutan terbalik dapat dilakukan sebagai berikut:

```
sort(v.rbegin(),v.rend());
```

Sebuah array biasa dapat diurutkan sebagai berikut:

```
int n = 7; // ukuran array
int a[] = {4,2,5,3,5,8,3};
sort(a, a+n);
```

Kode berikut mengurutkan string s:

```
string s = "monkey";
sort(s.begin(), s.end());
```

Mengurutkan sebuah string berarti karakter dalam string tersebut diurutkan. Sebagai contoh, string "monkey" menjadi "ekmnoy".

Operator perbandingan

Fungsi `sort` memerlukan bahwa sebuah **operator perbandingan** didefinisikan untuk tipe data elemen yang akan diurutkan. Saat sorting, operator ini akan digunakan setiap kali perlu menentukan urutan antara dua elemen.

Sebagian besar tipe data dalam C++ memiliki operator perbandingan bawaan, sehingga elemen dari tipe tersebut dapat diurutkan secara otomatis. Misalnya, angka diurutkan berdasarkan nilainya, dan string diurutkan dalam urutan alfabet.

Pasangan (`pair`) diurutkan terutama berdasarkan elemen pertama (`first`). Namun, jika elemen pertama dari dua pasangan sama, maka sorting dilakukan berdasarkan elemen kedua (`second`):

```
vector<pair<int,int>> v;
v.push_back({1,5});
v.push_back({2,3});
v.push_back({1,2});
sort(v.begin(), v.end());
```

Setelah ini, urutan pasangan adalah (1,2), (1,5), dan (2,3).

Dengan cara yang sama, tuple `tuple` diurutkan terutama berdasarkan elemen pertama, kemudian berdasarkan elemen kedua, dan seterusnya:

```
vector<tuple<int,int,int>> v;
v.push_back({2,1,4});
v.push_back({1,5,3});
v.push_back({2,1,3});
sort(v.begin(), v.end());
```

Setelah ini, urutan tuple adalah (1,5,3), (2,1,3), dan (2,1,4).

User-defined structs

User-defined structs atau struct yang didefinisikan oleh pengguna tidak secara otomatis memiliki operator perbandingan. Operator tersebut harus didefinisikan di dalam struct sebagai fungsi `operator<`, yang parameternya adalah elemen lain dari tipe yang sama. Operator ini harus mengembalikan `true` jika elemen lebih kecil dari parameter, dan `false` jika tidak.

Sebagai contoh, berikut adalah struct `P` yang berisi koordinat x dan y dari suatu titik. Operator perbandingan didefinisikan sehingga titik-titik diurutkan terutama berdasarkan koordinat x, dan jika x sama, diurutkan berdasarkan koordinat y.

```
struct P {
    int x, y;
    bool operator<(const P &p) {
        if (x != p.x) return x < p.x;
        else return y < p.y;
    }
};
```

Fungsi Perbandingan

Kita juga bisa memberikan **fungsi perbandingan** eksternal ke fungsi `sort` sebagai **callback function**. Sebagai contoh, berikut adalah fungsi perbandingan `comp` yang mengurutkan string terutama berdasarkan panjangnya, dan jika panjangnya sama, diurutkan berdasarkan urutan alfabet:

```
bool comp(string a, string b) {
    if (a.size() != b.size()) return a.size() < b.size();
    return a < b;
}
```

Sekarang, sebuah vector yang berisi string dapat diurutkan dengan cara berikut:

```
sort(v.begin(), v.end(), comp);
```

Binary search

Metode umum untuk mencari sebuah elemen dalam array adalah dengan menggunakan loop **for** yang melewati semua elemen dalam array. Sebagai contoh, kode berikut mencari elemen x dalam array:

```
for (int i = 0; i < n; i++) {
    if (array[i] == x) {
        // x ditemukan di indeks i
    }
}
```

Kompleksitas waktu dari pendekatan ini adalah $O(n)$, karena dalam kasus terburuk, kita harus memeriksa semua elemen dalam array. Jika urutan elemen **acak**, ini adalah metode terbaik yang bisa digunakan karena tidak ada informasi tambahan yang dapat membantu menemukan elemen x lebih cepat.

Namun, jika array sudah **terurut**, situasinya berbeda. Dalam hal ini, kita bisa melakukan pencarian lebih cepat, karena urutan elemen dalam array dapat memandu proses pencarian. Algoritma **binary**

search berikut memungkinkan kita mencari elemen dalam array yang telah diurutkan dalam waktu $O(log\ n)$:

Metode 1

Cara umum untuk mengimplementasikan binary search mirip dengan mencari kata dalam kamus. Pencarian ini mempertahankan **wilayah aktif** dalam array, yang awalnya mencakup semua elemen dalam array. Kemudian, dilakukan beberapa langkah, di mana setiap langkah akan **membagi dua** ukuran wilayah tersebut.

Pada setiap langkah, pencarian memeriksa **elemen tengah** dari wilayah aktif:

- Jika elemen tengah adalah elemen yang dicari, pencarian berhenti.
- Jika elemen tengah lebih besar dari target, pencarian berlanjut ke bagian kiri.
- Jika elemen tengah lebih kecil dari target, pencarian berlanjut ke bagian kanan.

Ide di atas dapat diimplementasikan sebagai berikut:

```
int a = 0, b = n-1;
while (a <= b) {
    int k = (a + b) / 2;
    if (array[k] == x) {
        // x ditemukan di indeks k
    }
    if (array[k] > x) b = k - 1;
    else a = k + 1;
}
```

Dalam implementasi ini:

- **Wilayah aktif** adalah $a \ldots b$, yang awalnya dimulai dari $0 \ldots n-1$.
- Setiap langkah **membagi dua** ukuran wilayah pencarian.
- Oleh karena itu, kompleksitas waktu algoritma ini adalah $O(log\ n)$.

Metode 2

Cara alternatif untuk mengimplementasikan binary search didasarkan pada metode yang lebih efisien dalam mengiterasi elemen-elemen dalam array. Idenya adalah melakukan lompatan dan memperlambat kecepatan saat semakin dekat dengan elemen target.

Pencarian dilakukan dari kiri ke kanan dalam array, dengan panjang lompatan awal sebesar $n/2$. Pada setiap langkah, panjang lompatan dibagi dua: pertama $n/4$, lalu $n/8$, $n/16$, dan seterusnya, hingga akhirnya panjangnya menjadi 1. Setelah melakukan lompatan-lompatan ini, elemen target akan ditemukan atau dapat dipastikan bahwa elemen tersebut tidak ada dalam array.

Kode berikut mengimplementasikan ide tersebut:

```
int k = 0;
for (int b = n/2; b >= 1; b /= 2) {
    while (k+b < n && array[k+b] <= x) k += b;
}
if (array[k] == x) {
    // x ditemukan pada indeks k
}
```

Selama pencarian, variabel b menyimpan panjang lompatan saat ini. Kompleksitas waktu algoritma ini adalah $O(log\ n)$, karena kode dalam loop `while` dijalankan paling banyak dua kali untuk setiap panjang lompatan.

Fungsi C++

Pustaka standar C++ menyediakan fungsi-fungsi berikut yang berbasis binary search dan bekerja dalam waktu logaritmik:

- `lower_bound` mengembalikan pointer ke elemen pertama dalam array yang nilainya setidaknya x.
- `upper_bound` mengembalikan pointer ke elemen pertama dalam array yang nilainya lebih besar dari x.
- `equal_range` mengembalikan kedua pointer di atas.

Fungsi-fungsi ini mengasumsikan bahwa array sudah diurutkan. Jika tidak ada elemen yang memenuhi kriteria, pointer akan menunjuk ke elemen setelah elemen terakhir dalam array.

Sebagai contoh, kode berikut menentukan apakah sebuah array mengandung elemen dengan nilai x:

```
auto k = lower_bound(array, array+n, x) - array;
if (k < n && array[k] == x) {
    // x ditemukan pada indeks k
}
```

Kode berikut menghitung jumlah elemen yang bernilai x:

```
auto a = lower_bound(array, array+n, x);
auto b = upper_bound(array, array+n, x);
cout << b - a << "\n";
```

Menggunakan `equal_range`, kode menjadi lebih ringkas:

```
auto r = equal_range(array, array+n, x);
cout << r.second - r.first << "\n";
```

Mencari Solusi Terkecil

Salah satu penggunaan penting dari binary search adalah untuk menemukan posisi di mana nilai suatu *fungsi* berubah. Misalkan kita ingin mencari nilai terkecil k yang merupakan solusi valid untuk suatu masalah.

Kita diberikan sebuah fungsi `ok`(x) yang mengembalikan `true` jika x adalah solusi valid dan `false` jika tidak. Selain itu, kita tahu bahwa `ok`(x) bernilai `false` untuk $x < k$ dan `true` untuk $x \geq k$. Situasinya dapat digambarkan sebagai berikut:

x	0	1	…	$k-1$	k	$k+1$	…
`ok`(x)	`false`	`false`	…	`false`	`true`	`true`	…

Sekarang, nilai k dapat ditemukan menggunakan binary search:

```
int x = -1;
for (int b = z; b >= 1; b /= 2) {
    while (!ok(x + b)) x += b;
}
int k = x + 1;
```

Pencarian menemukan nilai x terbesar di mana $\texttt{ok}(x)$ bernilai `false`. Dengan demikian, nilai berikutnya $k = x + 1$ adalah nilai terkecil yang mungkin di mana $\texttt{ok}(k)$ bernilai `true`. Panjang lompatan awal z harus cukup besar, misalnya, suatu nilai yang sebelumnya kita ketahui bahwa $\texttt{ok}(z)$ bernilai `true`.

Algoritma memanggil fungsi `ok` sebanyak $O(\log z)$ kali, sehingga kompleksitas waktu total bergantung pada fungsi `ok`. Sebagai contoh, jika fungsi tersebut bekerja dalam waktu $O(n)$, maka kompleksitas waktu totalnya adalah $O(n \log z)$.

Mencari Nilai Maksimum

Binary search juga dapat digunakan untuk menemukan nilai maksimum dari suatu fungsi yang awalnya meningkat lalu menurun. Tugas kita adalah menemukan posisi k sehingga:

- $f(x) < f(x+1)$ ketika $x < k$, dan
- $f(x) > f(x+1)$ ketika $x \geq k$.

Idenya adalah menggunakan binary search untuk mencari nilai x terbesar di mana $f(x) < f(x+1)$. Ini berarti bahwa $k = x + 1$, karena $f(x+1) > f(x+2)$. Kode berikut mengimplementasikan pencarian ini:

```
int x = -1;
for (int b = z; b >= 1; b /= 2) {
    while (f(x+b) < f(x+b+1)) x += b;
}
int k = x+1;
```

Perhatikan bahwa, tidak seperti pada binary search biasa, di sini tidak diperbolehkan adanya nilai fungsi yang berurutan tetapi sama besar. Jika terjadi, maka tidak mungkin menentukan bagaimana pencarian harus dilanjutkan.