فصل بیستم

جستجو و مرتب‌سازی

اهداف

- جستجو برای یافتن مقداری در یک بردار با استفاده از روش جستجوی باینری.
- مرتب‌سازی بردار با استفاده از الگوریتم بازگشتی بازگشتی merge.
- تعیین کارایی الگوریتم‌های جستجو و مرتب‌سازی.

رئوس مطالب

20-1 مقدمه

جستجوی داده‌ها مستلزم تعیین وجود یک مقدار (که از آن به عنوان کلید جستجو یاد می‌شود) در میان مقادیر دیگر بوده و اگر چنین باشد، موقعیت آن مقدار بدست می‌آید. از الگوریتم‌های محبوب در زمینه جستجو می‌توان به الگوریتم ساده *جستجوی خطی* (*linear search*) و الگوریتم سریع‌تر اما پیچیده‌تر *جستجوی باینری* (*binary search*) اشاره کرد.

در مرتب‌سازی (sorting) مبادرت به قرار دادن داده‌ها به ترتیب صعودی یا نزولی، براساس یک یا چند کلید مرتب‌سازی (sort keys) می‌شود. برای مثال، اسامی موجود در یک دفتر تلفن را می‌توان براساس الفبا، حساب‌های بانکی را براساس شماره حساب بانکی، لیست حقوق کارمندان را براساس شماره تامین اجتماعی مرتب کرد. قبلاً در مورد مرتب‌سازی درجی (interstion) و مرتب‌سازی انتخابی (selection) در فصل‌های هفتم و هشتم توضیحاتی داده شده است. در این فصل به توضیح الگوریتم مرتب‌سازی کارآمدتری بنام *مرتب‌سازی ادغامی* (*merge sort*) خواهیم پرداخت.

جدول، شکل 20-1 حاوی الگوریتم‌های مرتب‌سازی و جستجوی مطرح شده در این کتاب است. همچنین در این فصل به معرفی نماد O بزرگ (Big O) می‌پردازیم، که برای برآورد کردن بدترین زمان اجرای یک الگوریتم است، یعنی حداکثر کاری که باید یک الگوریتم انجام دهد تا مسئله‌ای حل گردد.

20-2 الگوریتم‌های جستجو

جستجوی یک شماره تلفن، دسترسی به یک وب سایت و پیدا کردن معنی یک کلمه در واژه‌نامه همگی مستلزم جستجو در میان حجم زیادی از داده‌ها است. الگوریتم‌های جستجو همگی برای برآورده کردن چنین اهدافی بکار گرفته می‌شوند، یافتن عنصری که با کلید جستجو مطابقت دارد، اگر چنین عنصری وجود داشته باشد، پس وجود دارد. با این وجود، تمام این الگوریتم‌ها در برخی از موارد با یکدیگر تفاوت دارند. اصلی‌ترین تفاوت در میزان سعی است که برای کامل کردن جستجو صرف می‌کنند. یکی از

روش‌های توصیف این سعی و تلاش استفاده از نماد Big O می‌باشد. در الگوریتم‌های جستجو و مرتب‌سازی، این مقدار وابسته به تعداد عناصر داده است.

در فصل هفتم، در مورد الگوریتم جستجوی خطی بحث کرده‌ایم، که یک الگوریتم ساده بوده و پیاده‌سازی آن آسان می‌باشد. اکنون در ارتباط با میزان کارایی این الگوریتم که با نماد Big O اندازه‌گیری می‌شود، صحبت می‌کنیم. سپس، به معرفی یک الگوریتم جستجوی می‌پردازیم که نسبتاً از کارایی مناسبی برخوردار است، اما پیچیده بوده و پیاده‌سازی آن مشکل است.

20-2-1 کارایی جستجوی خطی

فرض کنید الگوریتم ساده‌ای داریم که تعیین می‌کند آیا عنصر اول در یک بردار معادل با عنصر دوم در بردار است یا خیر. اگر بردار 10 عنصر داشته باشد، این الگوریتم مستلزم یک مقایسه است. اگر بردار دارای 1000 عنصر باشد، هنوز هم الگوریتم مستلزم یک مقایسه است. در واقع، الگوریتم بطور کامل مستقل از تعداد عناصر در بردار عمل می‌کند.

فصل	الگوریتم
الگوریتم‌های جستجو	
هفتم	جستجوی خطی
بیستم	جستجوی باینری
بیست و یکم	درخت جستجوی باینری
الگوریتم مرتب‌سازی	
هفتم	مرتب‌سازی درجی
هشتم	مرتب‌سازی انتخابی
بیستم	مرتب‌سازی ادغامی
بیست و یکم	مرتب‌سازی درخت باینری

شکل 1-20 | الگوریتم‌های جستجو و مرتب‌سازی در این کتاب.

گفته می‌شود این الگوریتم دارای زمان اجرای ثابت (constant runtime) است که با نماد O(1) نشان داده می‌شود. الگوریتمی که دارای O(1) است، ضرورتاً مستلزم یک مقایسه نمی‌باشد. O(1) به این معنی است که تعداد مقایسه‌ها ثابت است، مقدار آن با افزایش سایز بردار رشد نمی‌کند. الگوریتمی که تست می‌کند آیا اولین عنصر از بردار برابر با هر سه عنصر بعدی است یا خیر همیشه مستلزم انجام سه مقایسه است، اما در نماد Big O با O(1) عرضه می‌شود. غالباً تلفظ O(1) بصورت "on the order of 1" یا ساده‌تر از آن "order 1" است.

الگوریتمی که تست می‌کند آیا اولین عنصر از بردار برابر با هر تعداد از عناصر دیگر در بردار است یا خیر، مستلزم انجام n-1 مقایسه است، که در این رابطه n تعداد عناصر موجود در بردار می‌باشد. اگر بردار دارای

10 عنصر باشد، این الگوریتم مستلزم انجام 9 مقایسه است. اگر بردار دارای 1000 عنصر باشد، این الگوریتم نیازمند 999 مقایسه خواهد بود. همانطوری که n بزرگتر می‌شود، بخش n از عبارت هم «مسلط» شده و تفریق از یک بی‌اهمیت می‌شود. به همین دلیل به الگوریتمی که مستلزم انجام n-1 مقایسه است (همانند موردی که بیان کردیم) گفته شود دارای O(n) است. به الگوریتمی با O(n) گفته می‌شود که دارای *زمان اجرای خطی* (*linear runtime*) است. تلفظ O(n) بصورت "on order of n" یا ساده‌تر "order n" است.

حل فرض کنید الگوریتمی داریم که تست می‌کند آیا عنصر از بردار در جای دیگری از بردار تکثیر شده است یا خیر. بایستی عنصر اول با هر عنصری در بردار مقایسه شود. عنصر دوم بایستی با هر عنصری بجز عنصر اول مقایسه شود (در بار اول مقایسه شده است). عنصر سوم بایستی با هر عنصری بجز دو عنصر اول مقایسه شود. در پایان، این الگوریتم با انجام $n^2/2 - n/2$ یا 1+2+...+(n-2)+(n+1) مقایسه خاتمه می‌یابد. همانطوری که n افزایش می‌یابد، عبارت n^2 مسلط شده و عبارت n بی‌اهمیت می‌شود. مجدداً نماد Big O عبارت n^2 را متمایز کرده، $n^2/2$ را باقی می‌گذارد. همانطوری که بزودی ملاحظه خواهید کرد، فاکتورهای ثابت در نماد Big O در نظر گرفته نمی‌شود.

نماد Big O علاقمند به نحوه رشد زمان اجرای الگوریتم در ارتباط با تعداد ایتم‌های پردازش شده است. فرض کنید الگوریتمی مستلزم n^2 مقایسه است. با چهار عنصر، الگوریتم مستلزم انجام 16 مقایسه، با هشت عنصر مستلزم 64 مقایسه خواهد بود. با این الگوریتم، با دو برابر شدن عناصر، تعداد مقایسه‌ها چهار برابر می‌شود. به الگوریتمی توجه کنید که مستلزم $n^2/2$ مقایسه است. با چهار عنصر، الگوریتم نیازمند هشت مقایسه، با هشت عنصر نیازمند 32 مقایسه خواهد بود. مجدداً با دو برابر شدن تعداد عناصر، تعداد مقایسه‌ها چهار برابر می‌شود. هر دو این الگوریتم‌ها براساس مربع n افزایش می‌یابند، از اینرو Big O ثابت‌ها را در نظر نمی‌گیرد و هر دو الگوریتم با $O(n^2)$ نشان داده می‌شود که از آن بعنوان *زمان اجرای درجه دوم* (*quadratic runtime*) یاد می‌شود. تلفظ آن بصورت "on order of n-squard" یا ساده‌تر "order n-squard" است.

زمانیکه n کوچک باشد، الگوریتم‌های $O(n^2)$ تاثیر قابل ملاحظه‌ای در کارایی نخواهند داشت. اما زمانیکه n افزایش یابد، متوجه کاهش کارایی خواهید شد. یک الگوریتم $O(n^2)$ که بر روی برداری با یک میلیون عنصر کار می‌کند، می‌تواند نیازمند یک تریلیون عملیات باشد (که هر یک می‌تواند مستلزم اجرای چندین دستورالعمل ماشین باشد). انجام چنین کاری می‌تواند به چند ساعت نیاز داشته باشد. برداری با یک میلیون عنصر، نیازمند یک عملیاتبه تعداد، عدد 1 با 18 صفر بتوان 2 است، عددی بسیار بزرگ‌ی که الگوریتم بتواند با آن کار کند. نوشتن الگوریتم‌های $O(n^2)$ آسان است، همانطوری که در فصل‌های قبلی با آنها

مواجه شده‌اید. در این فصل شاهد الگوریتم‌های با واحدهای Big O مناسب‌تر خواهید بود. غالباً چنین الگوریتم‌های نیازمند کمی هوشیاری و دقت بیشتر در پیاده‌سازی هستند، اما کارایی که بدست می‌دهند بسیار ارزشمندتر است، بویژه زمانیکه n مقدار بزرگتری دریافت می‌کند و الگوریتم در برنامه‌های بزرگتر بکار گرفته می‌شود.

الگوریتم جستجوی خطی در زمان O(n) اجرا می‌شود. بدترین حالت در این الگوریتم زمانی است که باید تمام عناصر بررسی شوند تا مشخص گردد آیا کلید جستجو در بردار وجود دارد یا خیر. اگر سایز بردار دو برابر گردد، تعداد مقایسه‌های انجام گرفته نیز دو برابر می‌شود. توجه کنید که اگر عنصری که مطابق با کلید جستجو است نزدیک به ابتدای بردار قرار گرفته باشد، کارایی جستجوی خطی مناسب خواهد بود. اما بدنبال الگوریتمی هستیم که از میانگین کارایی مناسبی در تمام حالات جستجو برخوردار باشد، چه عنصر در ابتدا، میانه یا انتهای بردار قرار گرفته باشد.

پیاده‌سازی جستجوی خطی کار آسانی است، اما در مقایسه با سایر الگوریتم‌های جستجو، آهسته‌تر عمل می‌کند. اگر برنامه‌ای نیاز به انجام جستجو در میان بردارهای بزرگ داشته باشد، بهتر خواهد بود تا الگوریتم بهتر با کارایی بالاتر همانند جستجو باینری را بکار گرفت که در بخش بعدی به توضیح آن خواهیم پرداخت.

20-2-2 جستجوی باینری

الگوریتم *جستجوی باینری* (*binary search*) به نسبت الگوریتم جستجوی خطی از کارایی بالاتری برخوردار است، اما نیازمند آن است که ابتدا بردار مرتب گردد. اینکار فقط زمانی ارزش دارد که بردار فقط یکبار مرتب شده، و به دفعات زیاد جستجو می‌گردد یا زمانیکه برنامه جستجو کننده سخت بدنبال کارایی باشد. در اولین تکرار، این الگوریتم عنصر وسط در بردار را تست می‌کند. اگر این عنصر مطابق با کلید جستجو باشد، الگوریتم پایان می‌پذیرد. فرض کنید بردار به ترتیب صعودی مرتب شده باشد، پس اگر کلید جستجو کمتر از عنصر وسط باشد، کلید جستجو نمی‌تواند با هیچ یک از عناصر موجود در نیمه دوم بردار مطابقت داشته باشد و الگوریتم کار را فقط با نیمه اول بردار ادامه می‌دهد (یعنی از عنصر اول تا رسیدن به عنصر وسط، خود عنصر وسط در نظر گرفته نمی‌شود). اگر کلید جستجو بزرگتر از عنصر وسط باشد، پس کلید جستجو نمی‌تواند با هیچ یک از عناصر موجود در نیمه اول بردار مطابقت داشته باشد و الگوریتم کار را فقط با نیمه دوم بردار ادامه می‌دهد (یعنی پس از عنصر وسط تا عنصر آخر). در هر تکرار مقدار وسط از بخش باقیمانده بردار تست می‌شود. اگر عنصر مطابق با کلید جستجو نباشد، الگوریتم نصف باقیمانده از عناصر را در نظر نمی‌گیرد. الگوریتم یا با یافتن عنصر مورد نظر خاتمه می‌یابد یا اینکه عنصر یافت نشده و زیر بردار به سایز صفر می‌رسد. بعنوان یک مثال به بردار 15 عنصری زیر توجه کنید

```
2 3 5 10 27 30 34 51 56 65 77 81 82 93 99
```

فرض کنید کلید جستجو 65 باشد. برنامه با استفاده از الگوریتم جستجوی باینری شروع به جستجو می‌کند. ابتدا بررسی می‌کند که آیا 51 کلید جستجو است یا خیر (چرا که 51 عنصر وسط در بردار است). کلید جستجو (65) بزرگتر از 51 می‌باشد، از اینرو 51 به همراه نیمه اول بردار به کنار گذاشته می‌شوند (تمام عناصر کوچکتر از 51). سپس الگوریتم بررسی می‌کند که آیا 81 (عنصر وسط از بردار باقیمانده) با کلید جستجو مطابقت می‌کند یا خیر. کلید جستجو (65) کوچکتر از 81 است، از اینرو، 81 به همراه عناصر بزرگتر از 81 به کنار گذاشته می‌شوند. فقط پس از انجام دو تست، الگوریتم تعداد عناصری که باید بررسی شوند را به سه کاهش داده است (65, 56 و 77). سپس الگوریتم به بررسی 65 می‌پردازد (که به راستی با کلید جستجو مطابقت داد) و شاخص آن عنصر (9) را که حاوی 65 است برگشت می‌دهد. در این مورد خاص، این الگوریتم با سه عملیات مقایسه موفق به تعیین مطابقت عنصری با کلید جستجو شده است. در حالیکه جستجوی خطی برای انجام اینکار مستلزم انجام 10 مقایسه است.

در برنامه شکل‌های 2–20 و 3–20 کلاس **BinarySearch** و توابع عضو آن تعریف شده‌اند. این کلاس شبیه کلاس **LinearSearch** در فصل هفتم است. این کلاس دارای یک سازنده، یک تابع جستجو (**binarySearch**)، تابع **disyplayElements**، دو عضو داده خصوصی و یک تابع کمکی بنام **displaySubElements** است. در خطوط 28-18 از شکل 3–20 یک سازنده تعریف شده است. پس از مقداردهی بردار با مقادیر تصادفی صحیح از 99-10 در خطوط 25-24، خط 27 تابع استاندارد کتابخانه‌ای **sort** را بر روی بردار **data** فراخوانی می‌کند. تابع **sort** دو تکرار شونده با دسترسی تصادفی دریافت و عناصر موجود در بردار را به ترتیب صعودی مرتب می‌کند. این نوع تکرار شونده، تکرار شونده‌ای است که دارای مجوز دسترسی به هر ایتم موجود در بردار را در هر زمان دارد. در این برنامه از **data.being()** و **data.end()** برای احاطه کردن کل بردار استفاده کرده‌ایم. بخاطر داشته باشد که الگوریتم جستجوی باینری فقط بر روی بردار مرتب شده کار می‌کند.

خطوط 36-31 تعریف کننده تابع **binarySearch** هستند. کلید جستجو به پارامتر **searchElement** ارسال می‌شود (خط 31). خطوط 35-33 مبادرت به محاسبه شاخص پایین **low**، شاخص بالا **high** و شاخص وسط یا میانی **middle** در بخشی از بردار می‌کنند که برنامه در حال حاضر آنرا جستجو می‌کند. در ابتدای کار تابع، **low** برابر صفر، **high** برابر سایز بردار منهای **1** و **middle** برابر میانگین این دو مقدار است. خط 36 مبادرت به مقداردهی اولیه **location** با **1-** می‌کند، مقداری که در صورت عدم دریافت کلید جستجو، برگشت داده خواهد شد. حلقه موجود در خطوط 58-38 تا زمانیکه **low** بزرگتر از **high** شده یا **location** برابر **1-** نشود ادامه می‌یابد.

```
1      // Fig 20.2: BinarySearch.h
2      // Class that contains a vector of random integers and a function
3      // that uses binary search to find an integer.
```

```
#include <vector>
using std::vector;

class BinarySearch
{
public:
   BinarySearch( int ); // constructor initializes vector
   int binarySearch( int ) const;// perform a binary search on vector
   void displayElements() const; // display vector elements
private:
   int size; // vector size
   vector< int > data; // vector of ints
   void displaySubElements(int,int) const;// display range of values
}; // end class BinarySearch
```

شکل 2–20 | تعریف کلاس BinarySearch.

```
// Fig 20.3: BinarySearch.cpp
// BinarySearch class member-function definition.
#include <iostream>
using std::cout;
using std::endl;

#include <cstdlib> // prototypes for functions srand and rand
using std::rand;
using std::srand;

#include <ctime> // prototype for function time
using std::time;

#include <algorithm> // prototype for sort function
#include "BinarySearch.h" // class BinarySearch definition

//constructor initializes vector with random ints and sorts the vector
BinarySearch::BinarySearch( int vectorSize )
{
   size = (vectorSize > 0 ? vectorSize : 10); // validate vectorSize
   srand( time( 0 ) ); // seed using current time

   // fill vector with random ints in range 10-99
   for ( int i = 0; i < size; i++ )
      data.push_back( 10 + rand() % 90 ); // 10-99

   std::sort( data.begin(), data.end() ); // sort the data
} // end BinarySearch constructor

// perform a binary search on the data
int BinarySearch::binarySearch( int searchElement ) const
{
   int low = 0; // low end of the search area
   int high = size - 1; // high end of the search area
   int middle = ( low + high + 1 ) / 2; // middle element
   int location = -1; // return value; -1 if not found

   do // loop to search for element
   {
      // print remaining elements of vector to be searched
      displaySubElements( low, high );

      // output spaces for alignment
      for ( int i = 0; i < middle; i++ )
        cout << "   ";

```

```
47            cout << " * " << endl; // indicate current middle
48
49            // if the element is found at the middle
50            if ( searchElement == data[ middle ] )
51               location = middle; // location is the current middle
52            else if (searchElement < data[ middle ]) // middle is too high
53               high = middle - 1; // eliminate the higher half
54            else // middle element is too low
55               low = middle + 1; // eliminate the lower half
56
57            middle = ( low + high + 1 ) / 2; // recalculate the middle
58         } while ( ( low <= high ) && ( location == -1 ) );
59
60         return location; // return location of search key
61      } // end function binarySearch
62
63      // display values in vector
64      void BinarySearch::displayElements() const
65      {
66         displaySubElements( 0, size - 1 );
67      } // end function displayElements
68
69      // display certain values in vector
70      void BinarySearch::displaySubElements( int low, int high ) const
71      {
72         for ( int i = 0; i < low; i++ ) // output spaces for alignment
73            cout << "   ";
74
75         for (int i = low; i <= high; i++)//output elements left in vector
76            cout << data[ i ] << " ";
77
78         cout << endl;
79      } // end function displaySubElements
```

شکل 3–20 | تعریف تابع عضو کلاس BinarySearch.

خط 50 تست می‌کند که آیا مقدار موجود در عنصر **middle** برابر با **searchElement** است یا خیر. اگر چنین باشد، خط 51 مبادرت به تخصیص **middle** به **location** می‌کند، سپس حلقه خاتمه یافته و **location** به فراخوان برگردانده می‌شود. هر تکرار حلقه مبادرت به تست یک مقدار می‌کند (خط 50) و نصف باقیمانده مقادیر در بردار را به کنار می‌گذارد (خط 53 یا 55).

حلقه خطوط 41-25 از شکل 4–20 تا زمانیکه کاربر مقدار 1- وارد کند، ادامه می‌یابد. برای هر عدد دیگری که کاربر وارد کند، برنامه یک جستجوی باینری بر روی داده انجام می‌دهد و تعیین می‌کند که آیا با عنصری در بردار مطابقت می‌کند یا خیر.

کارایی جستجوی باینری

در بدترین وضعیت، جستجوی یک بردار مرتب شده با 1023 عنصر با استفاده از الگوریتم جستجوی باینری فقط 10 مقایسه لازم دارد. با تقسیم متوالی 1023 به 2 و گرد کردن آن مقادیر 1,3,7,15,31,63,127,255,511 و صفر بدست می‌آیند. عدد (2^{10}- 1) 1023 به 2 فقط 10 بار تقسیم می‌شود تا مقدار صفر بدست آید، که مشخص می‌کند که هیچ عنصری برای تست باقی نمانده است.

تقسیم بر 2 معادل با یک مقایسه در الگوریتم جستجوی باینری است. از اینرو، برداری با - 2^{20}) 1,048,575 (1 عنصر حداکثر به 20 مقایسه برای یافتن کلید نیاز دارد و برداری با بیش از یک تریلیون حداکثر به 30 مقایسه نیاز خواهد داشت، که در مقایسه با جستجوی خطی، بهبود بسیار عظیمی دیده می‌شود. اگر این تعداد به الگوریتم جستجوی خطی داده شود، بطور میانگین نیاز به 500 میلیون مقایسه خواهد بود. حداکثر تعداد مقایسه مورد نیاز در جستجوی باینری بر روی هر بردار مرتب شده، توانی از 2 است بزرگتر از تعداد عناصر در بردار که بصورت خ مشابه صورت تقریباً با نر هاشود. رشد تمام لگاریتم‌نشان داده می $\log_2^n$ برای O(log n)توان پایه را در نظر نگرفت. نتیجه اینکار بصورت می Big Oگیرد، از اینرو در نماد می شود و شناخته می)*logarithmic runtime*(*زمان اجرای لگاریتمی*جستجوی باینری خواهد بود، که بعنوان ی‌اهم‌تیرو‌گلا 02–3است. order" n log"تر یا ساده on the order of log n"تلفظ آن بصورت

ار ی‌دربرا ک ی‌اهه‌مانرب ی‌اهش‌خب ن‌یرتمهم زا ی‌کی اهه‌داد ی‌زاسب‌ترم سازی‌مرتب

توان شوند، از اینروست که می‌ها توسط شماره حساب مرتب می‌دهد. در یک بانک تمام چک‌تشکیل می های خود را براساس نام ـت‌های تلفن لیسای در پایان هر ماه آماده کرد. شرکت‌صورتحساب جداگانه های تلفن آسانتر شود. در واقع هر سازمانی که با کنند تا یافتن شماره‌خانوادگی و همچنین نام مرتب می طابترا رد هطقن ن‌یرتمهم‌های خود است.سازی داده‌حجم زیادی از اطلاعات سر و کار دارد نیازمند مرتب و مهم نیست که کدام الگوریتم باشدمی‌دار مرتب شده که یک بر ،سازی در نتیجه پایانی است‌با مرتب برای مرتب کردن بردار بکار گرفته شده است. الگوریتم انتخاب شده فقط بر روی زمان اجرا و حافظه سازی انتخابی و درجی پرداختیم که های قبلی، به معرفی مرتب‌مورد استفاده برنامه تاثیر دارد. در فصل اما از کارایی ضعیفی برخوردار بودند. در بخش بعدی به بررسی کارایی این دو سازی آسان پیاده‌دارای پردازیم می (merge)پردازیم. در پایان به بررسی الگوریتم ادغامی می Big Oالگوریتم با استفاده از نماد // BinarySearch // Fig 20.4: Fig20_04.cppسازی آن مشکل است.که سریعتر بوده اما پیاده test program

20–3 الگوریتم‌های مرتب‌سازی

مرتب‌سازی داده‌ها یکی از مهمترین بخش‌های برنامه‌های کاربردی را تشکیل می‌دهد. در یک بانک تمام چک‌ها توسط شماره حساب مرتب می‌شوند، از اینروست که می‌توان صورتحساب جداگانه‌ای در پایان هر ماه آماده کرد. شرکت‌های تلفن لیست‌های خود را براساس نام خانوادگی و همچنین نام مرتب می‌کنند تا یافتن شماره‌های تلفن آسانتر شود. در واقع هر سازمانی که با حجم زیادی از اطلاعات سر و کار دارد نیازمند مرتب‌سازی داده‌های خود است.

مهمترین نقطه در ارتباط با مرتب‌سازی در نتیجه پایانی است، که یک بردار مرتب شده می‌باشد و مهم نیست که کدام الگوریتم برای مرتب کردن بردار بکار گرفته شده است. الگوریتم انتخاب شده فقط بر روی زمان اجرا و حافظه مورد استفاده برنامه تاثیر دارد. در فصل‌های قبلی، به معرفی مرتب‌سازی انتخابی و درجی پرداختیم که دارای پیاده‌سازی آسان اما از کارایی ضعیفی برخوردار بودند. در بخش بعدی به بررسی کارایی این دو الگوریتم با استفاده از نماد Big O می‌پردازیم. در پایان به بررسی الگوریتم ادغامی (merge) می‌پردازیم که سریعتر بوده اما پیاده‌سازی آن مشکل است.

```
// Fig 20.4: Fig20_04.cpp
// BinarySearch test program.
#include <iostream>
using std::cin;
using std::cout;
using std::endl;

#include "BinarySearch.h" // class BinarySearch definition

int main()
{
   int searchInt; // search key
   int position; // location of search key in vector

   // create vector and output it
   BinarySearch searchVector ( 15 );
   searchVector.displayElements();

   // get input from user
   cout << "\nPlease enter an integer value (-1 to quit): ";
   cin >> searchInt; // read an int from user
   cout << endl;

   // repeatedly input an integer; -1 terminates the program
   while ( searchInt != -1 )
   {
      // use binary search to try to find integer
      position = searchVector.binarySearch( searchInt );

      // return value of -1 indicates integer was not found
      if ( position == -1 )
         cout << "The integer " << searchInt << " was not found.\n";
      else
         cout << "The integer " << searchInt
            << " was found in position " << position << ".\n";

      // get input from user
      cout << "\n\nPlease enter an integer value (-1 to quit): ";
      cin >> searchInt; // read an int from user
      cout << endl;
   } // end while

   return 0;
} // end main
```

```
26 31 33 38 47 49 49 67 73 74 82 89 90 91 95

Please enter an integer value (-1 to quit): 38
26 31 33 38 47 49 49 67 73 74 82 89 90 91 95
                      *
```

```
26 31 33 38 47 49 49
          *
26 31 33 38 47 49 49 67 73 74 82 89 90 91 95

Please enter an integer value (-1 to quit): 38
26 31 33 38 47 49 49 67 73 74 82 89 90 91 95
                      *
26 31 33 38 47 49 49
          *
The integer 38 was found in position 3.

Please enter an integer value (-1 to quit): 91
26 31 33 38 47 49 49 67 73 74 82 89 90 91 95
                      *
                        73 74 82 89 90 91 95
                                  *
                                    90 91 95
                                       *
The integer 91 was found in position 13.

Please enter an integer value (-1 to quit): 25
26 31 33 38 47 49 49 67 73 74 82 89 90 91 95
                      *
26 31 33 38 47 49 49
          *
26 31 33
    *
26
 *
The integer 25 was not found.

Please enter an integer value (-1 to quit): -1
```

شکل 4–20 | برنامه تست BinarySearch.

1-3-20 کارایی مرتب‌سازی انتخابی

پیاده‌سازی الگوریتم *مرتب‌سازی انتخابی* (*selection sort*) کار آسانی است اما از کارایی پایینی برخوردار است. در اولین حرکت این الگوریتم، کوچکترین عنصر در بردار انتخاب شده و مکان آن با اولین عنصر عوض می‌شود. در دومین حرکت، دومین عنصر کوچک (کوچکترین عنصر در میان عناصر باقیمانده) انتخاب و مکان آن با دومین عنصر عوض می‌شود. الگوریتم تا آخرین حرکت که انتخاب دومین عنصر بزرگ و عوض کردن آن با دومین عنصر آخر ادامه می‌دهد، و بزرگترین عنصر در آخرین شاخص گذاشته می‌شود. پس از *ith* تکرار، کوچکترین عناصر i بردار بصورت صعودی مرتب می‌شوند.

الگوریتم مرتب‌سازی انتخابی n-1 بار تکرار می‌شود و در هر بار جابجایی کوچکترین عنصر باقی مانده در موقعیت مرتب شده خود جای می‌گیرد. پیدا کردن کوچکترین عنصر باقی مانده مستلزم n-1 مقایسه در اولین تکرار، n-2 در زمان دومین تکرار، سپس n-3,...,3,2,1 تکرار است. در نتیجه مجموع (n2 - n)/2 یا n(n - 1)/2 مقایسه بدست می‌آید. در نماد Big O، کوچکترین حالت و ثابت‌ها به کنار گذاشته می‌شوند و در نتیجه $O(n^2)$ بدست می‌آید.

20-3-2 کارایی مرتب‌سازی درجی

مرتب‌سازی درجی (*insertin sort*) از پیاده‌سازی ساده‌ای برخوردار بوده، اما از کارایی چندانی برخوردار نیست. در اولین تکرار این الگوریتم، دومین عنصر در بردار گرفته شده و اگر کوچکتر از اولین عنصر باشد، آنرا با عنصر اول عوض می‌کند. در دومین تکرار به سومین عنصر نگاه شده و آنرا در موقعیت صحیح با نگاهی به دو عنصر اول درج می‌کند، از اینرو هر سه عنصر مرتب می‌شوند، در i*th* تکرار این الگوریتم، اولین عناصر i در بردار اصلی مرتب شده خواهند بود.

مرتب‌سازی درجی، n-1 بار تکرار می‌شود، یک عنصر در موقعیت مناسب در میان عناصر مرتب شده تا بدین جا، قرار داده می‌شود. در هر تکرار، تعیین می‌شود که عنصر درج شونده نیاز به مقایسه با هر عنصر قبل از خود در بردار دارد یا خیر. در بدترین حالت، نیازمند n-1 مقایسه است. هر عبارت تکرار در زمان O(n) اجرا می‌شود. در تعیین نماد Big O، عبارات تودرتو باید تعداد مقایسه‌ها به هم ضرب شوند. برای هر تکرار حلقه خارجی، تعداد مشخصی از تکرار حلقه داخلی وجود دارد. در این الگوریتم، برای هر O(n) تکرار حلقه خارجی، O(n) تکرار برای حلقه داخلی وجود دارد که نتیجه آن O(n*n) یا $O(n^2)$ است.

20-3-3 مرتب‌سازی ادغامی (پیاده‌سازی بازگشتی)

مرتب‌سازی ادغامی از الگوریتم‌های مرتب‌سازی با کارایی بالا بوده اما به لحاظ مفهومی بسیار پیچیده‌تر از مرتب‌سازی انتخابی و درجی می‌باشد. الگوریتم مرتب‌سازی ادغامی اقدام به مرتب کردن یک بردار با تقسیم آن به دو زیر بردار با سایز برابر کرده، هر زیر بردار را مرتب و سپس دو زیر بردار را با هم ادغام می‌کند تا بردار بزرگتر بدست آید. در یک بردار با تعداد عناصر فرد، الگوریتم دو زیر بردار ایجاد می‌کند که یکی بیش از دیگری یک عنصر بیشتر دارد.

پیاده‌سازی مرتب‌سازی ادغامی در این بخش بصورت بازگشتی است. حالت پایه یا مبنا، برداری است با یک عنصر. البته، بردار یک عنصری، مرتب شده است، از اینرو مرتب‌سازی ادغامی بلافاصله به هنگام برخورد با بردار یک عنصری، برگشت داده می‌شود. گام بازگشتی مبادرت به تقسیم برداری با دو یا چند عنصر به دو زیر بردار با سایز برابر می‌کند، بطور بازگشتی هر زیر بردار مرتب شده، سپس آن دو زیر بردار به هم می‌پیوندند.

فرض کنید که در حال حاضر الگوریتم دارای بردارهای کوچکتر مرتب شده A و B است که می‌خواهند به با ادغام شوند تا یک بردار بزرگ مرتب شده بوجود آید، بردار A:

```
4 10 34 56 77
```

و بردار B:

```
5 30 51 52 93
```

کوچکترین عنصر در A عدد 4 است (در شاخص صفر). کوچکترین عنصر در B عدد 5 است (در شاخص صفر از B). برای تعیین کوچکترین عنصر در بردار بزرگ، الگوریتم مبادرت به مقایسه 4 و 5 می‌کند. مقدار موجود در A کوچکتر بوده، از اینرو 4 تبدیل به اولین عنصر در بردار ادغام شده می‌شود. الگوریتم با مقایسه 10 (دومین عنصر در A) با 5 (اولین عنصر در B) تعیین می‌کند که مقدار موجود در B کوچکتر است، از اینرو 5 تبدیل به دومین عنصر در بردار بزرگ می‌شود. الگوریتم به مقایسه 10 با 30، ادامه داده و 10 تبدیل به سومین عنصر در بردار می‌شود و اینکار تا پایان ادامه می‌یابد.

برنامه شکل 5-20 کلاس **MergeSort** را تعریف کرده و خطوط 34-31 از شکل 6-20 تابع **sort** را تعریف کرده‌اند. خط 33 تابع **sortSubVector** با آرگومان‌های صفر و size-1 فراخوانی کرده است. این آرگومان‌ها متناظر با شاخص‌های ابتدا و انتهای برداری هستند که مرتب شده‌اند و سبب می‌شوند تا **sortSubVector** بر روی کل بردار عمل کند. تابع **sortSubVector** در خطوط 61-37 تعریف شده است. خط 40 تست کننده حالت پایه است. اگر سایز بردار 1 باشد، تابع بردار را به دو قسمت تقسیم کرده، بطور بازگشتی تابع **sortSubVector** را برای مرتب کردن دو زیر بردار فراخوانی کرده، سپس آنها را با هم ادغام می‌کند. خط 55 بصورت بازگشتی تابع **sortSubVetor** را بر روی نیمه اول بردار فراخوانی کرده، و خط 56 بصورت بازگشتی تابع **sortSubVector** را بر روی نیمه دوم بردار فراخوانی می‌کند. زمانیکه این دو تابع باز گردند، هر نیمه از بردار مرتب شده خواهد بود.

```
// Fig 20.05: MergeSort.h
// Class that creates a vector filled with random integers.
// Provides a function to sort the vector with merge sort.
#include <vector>
using std::vector;

// MergeSort class definition
class MergeSort
{
public:
   MergeSort( int ); // constructor initializes vector
   void sort(); // sort vector using merge sort
   void displayElements() const; // display vector elements
private:
   int size; // vector size
   vector< int > data; // vector of ints
   void sortSubVector( int, int ); // sort subvector
   void merge( int, int, int, int ); // merge two sorted vectors
   void displaySubVector( int, int ) const; // display subvector
}; // end class SelectionSort
```

شکل 5-20 | تعریف کلاس MergSort.

```
1   // Fig 20.06: MergeSort.cpp
2   // Class MergeSort member-function definition.
3   #include <iostream>
4   using std::cout;
5   using std::endl;
6
7   #include <vector>
```

```
using std::vector;

#include <cstdlib> // prototypes for functions srand and rand
using std::rand;
using std::srand;

#include <ctime> // prototype for function time
using std::time;

#include "MergeSort.h" // class MergeSort definition

// constructor fill vector with random integers
MergeSort::MergeSort( int vectorSize )
{
   size = (vectorSize > 0 ? vectorSize : 10);// validate vectorSize
   srand(time(0));// seed random number generator using current time

   // fill vector with random ints in range 10-99
   for ( int i = 0; i < size; i++ )
      data.push_back( 10 + rand() % 90 );
} // end MergeSort constructor

// split vector, sort subvectors and merge subvectors into sorted vector
void MergeSort::sort()
{
   sortSubVector( 0, size - 1 ); // recursively sort entire vector
} // end function sort

// recursive function to sort subvectors
void MergeSort::sortSubVector( int low, int high )
{
   // test base case; size of vector equals 1
   if ( ( high - low ) >= 1 ) // if not base case
   {
      int middle1 = ( low + high ) / 2; //calculate middle of vector
      int middle2 = middle1 + 1; // calculate next element over

      // output split step
      cout << "split:   ";
      displaySubVector( low, high );
      cout << endl << "         ";
      displaySubVector( low, middle1 );
      cout << endl << "         ";
      displaySubVector( middle2, high );
      cout << endl << endl;

      // split vector in half; sort each half (recursive calls)
      sortSubVector( low, middle1 ); // first half of vector
      sortSubVector( middle2, high ); // second half of vector

      // merge two sorted vectors after split calls return
      merge( low, middle1, middle2, high );
   } // end if
} // end function sortSubVector

// merge two sorted subvectors into one sorted subvector
void MergeSort::merge(int left, int middle1, int middle2, int right)
{
   int leftIndex = left; // index into left subvector
   int rightIndex = middle2; // index into right subvector
   int combinedIndex = left; // index into temporary working vector
   vector< int > combined( size ); // working vector
```

```
      // output two subvectors before merging
      cout << "merge:   ";
      displaySubVector( left, middle1 );
      cout << endl << "          ";
      displaySubVector( middle2, right );
      cout << endl;

      // merge vectors until reaching end of either
      while ( leftIndex <= middle1 && rightIndex <= right )
      {
         // place smaller of two current elements into result
         // and move to next space in vector
         if ( data[ leftIndex ] <= data[ rightIndex ] )
            combined[ combinedIndex++ ] = data[ leftIndex++ ];
         else
            combined[ combinedIndex++ ] = data[ rightIndex++ ];
      } // end while

      if ( leftIndex == middle2 ) // if at end of left vector
      {
         while ( rightIndex <= right ) // copy in rest of right vector
            combined[ combinedIndex++ ] = data[ rightIndex++ ];
      } // end if
      else // at end of right vector
      {
         while ( leftIndex <= middle1 ) // copy in rest of left vector
            combined[ combinedIndex++ ] = data[ leftIndex++ ];
      } // end else

      // copy values back into original vector
      for ( int i = left; i <= right; i++ )
         data[ i ] = combined[ i ];

      // output merged vector
      cout << "         ";
      displaySubVector( left, right );
      cout << endl << endl;
   } // end function merge

   // display elements in vector
   void MergeSort::displayElements() const
   {
      displaySubVector( 0, size - 1 );
   } // end function displayElements

   // display certain values in vector
   void MergeSort::displaySubVector( int low, int high ) const
   {
      // output spaces for alignment
      for ( int i = 0; i < low; i++ )
         cout << "   ";

      // output elements left in vector
      for ( int i = low; i <= high; i++ )
         cout << " " << data[ i ];
   } // end function displaySubVector
```

شکل 6-20 | تعریف تابع عضو کلاس MergSort.

خط 59 تابع **merge** (خطوط 64-108) را بر روی بردار دو نیم شده فراخوانی می‌کند تا دو بردار مرتب شده با هم ترکیب و یک بردار مرتب شده بزرگتر بوجود آید.

خطوط 87-79 در تابع **merge** تا زمانیکه برنامه به انتهای زیر بردارها برسد، تکرار می‌شوند (حلقه). خط 83 تست می‌کند که کدام عنصر در ابتدای بردارها کوچکتر است. اگر عنصر موجود در بردار چپ کوچکتر باشد، خط 84 آنرا در موقعیت بردار ترکیبی قرار می‌دهد. اگر عنصر موجود در بردار راست کوچکتر باشد، خط 86 آنرا در موقعیت بردار ترکیبی قرار می‌دهد. زمانیکه حلقه **while** کامل شد (خط 87)، کل یک زیر بردار در بردار ترکیبی جای خواهد داشت، اما زیر بردار دیگر هنوز دارای داده است. خط 89 تست می‌کند که آیا بردار چپ به انتها رسیده است یا خیر. اگر چنین باشد، خطوط 92-91 بردار ترکیبی را با عناصر بردار راست پر می‌کنند. اگر بردار چپ به انتها نرسیده باشد، پس بایستی بردار راست به انتها رسیده باشد و خطوط 97-96 بردار ترکیبی را با عناصر بردار چپ پر می‌کنند. سرانجام، خطوط 101-102 بردار ترکیبی را بر روی بردار اولیه کپی می‌کنند. برنامه شکل 7–20 یک شی **MergeSorg** ایجاد کرده و از آن استفاده می‌کند.

```
// Fig 20.07: Fig20_07.cpp
// MergeSort test program.
#include <iostream>
using std::cout;
using std::endl;

#include "MergeSort.h" // class MergeSort definition

int main()
{
   // create object to perform merge sort
   MergeSort sortVector( 10 );

   cout << "Unsorted vector:" << endl;
   sortVector.displayElements(); // print unsorted vector
   cout << endl << endl;

   sortVector.sort(); // sort vector

   cout << "Sorted vector:" << endl;
   sortVector.displayElements(); // print sorted vector
   cout << endl;
   return 0;
} // end main
```

```
Unsorted vector:
30 47 22 67 79 18 60 78 26 54

split:   30 47 22 67 79 18 60 78 26 54
         30 47 22 67 79
                        18 60 78 26 54

split:   30 47 22 67 79
         30 47 22
                  67 79
```

```
split:    30 47 22
          30 47
                22

split:    30 47
          30
             47

merge:    30
             47
          30 47

merge:    30 47
                22
          22 30 47
split:             67 79
                   67
                      79

merge:             67
                      79
                   67 79

merge:    22 30 47
                   67 79
          22 30 47 67 79

split:                  18 60 78 26 54
                        18 60 78
                                 26 54

split:                  18 60 78
                        18 60
                              78

split:                  18 60
                        18
                           60

merge:                  18
                           60
                        18 60

merge:                  18 60
                              78
                        18 60 78

split                            26 54
                                 26
                                    54

merge:                           26
                                    54
                                 26 54

merge:                  18 60 78
                                 26 54
                        18 26 54 60 78
merge:   22 30 47 67 79
                        18 26 54 60 78
         18 22 26 30 47 54 60 67 78 79
```

```
Sorted vector:
18 22 26 30 47 54 60 67 78 79
```

شکل 7–20 | برنامه تست MergeSort.

کارایی مرتب‌سازی ادغامی

کارایی مرتب‌سازی ادغامی در مقایسه با مرتب‌سازی‌های انتخابی و درجی بسیار بیشتر است. به اولین فراخوانی (غیر بازگشتی) تابع **sortSubVector** توجه کنید. نتیجه این فراخوانی، دو فراخوانی بازگشتی برای تابع **sortSubVector** با دو زیر بردار است که هر یک تقریباً نصف سایز بردار اولیه، سایز دارند و یک فراخوانی برای تابع **merge** صورت می‌گیرد. فراخوانی تابع **merge** در بدترین حالت، نیازمند n-1 مقایسه برای پر کردن بردار اولیه (اصلی) است که برابر O(n) است. نتیجه دو فراخوانی برای تابع **sortSubVector** فراخوانی بازگشتی بیش از چهار بار تابع **sortSubVector** است که هر یک با یک زیر بردار تقریباً یک چهارم سایز بردار اولیه، به همراه دو فراخوانی تابع **merge** است. فراخوانی این دو تابع **merge** هر یک مستلزم n/2 - 1 مقایسه است که برای کل تعداد مقایسه O(n) را داریم. این فرآیند ادامه می‌یابد تا هر فراخوانی **sortSubVector** دو فراخوانی دیگر **sortSubVector** و یک فراخوانی **merge** انجام دهد تا اینکه الگوریتم به برداری با یک عنصر برسد. در هر سطح، O(n) مقایسه برای ادغام زیر بردارها لازم است. هر سطح مبادرت به تقسیم سایز بردارها به نصف کرده، از اینرو دو برابر کردن سایز بردار نیازمند یک سطح بیشتر است. چهار برابر کردن سایز بردار نیازمند دو سطح بیشتر است. این الگو حالت لگاریتمی داشته و نتیجه آن $\log_2^n$ سطح است. این نتایج بصورت کلی نشاندهنده $O(n \log n)$ می‌باشند. جدول شکل 8–20 بطور خلاصه برخی از نتایج Big O از الگوریتم‌های جستجو و مرتب‌سازی را نشان می‌دهد.

الگوریتم	Big O	الگوریتم	Big O
الگوریتم‌های جستجو		***الگوریتم‌های مرتب‌سازی***	
جستجوی خطی	$O(n)$	مرتب‌سازی درجی	$O(n^2)$
جستجوی باینری	$O(\log n)$	مرتب‌سازی انتخابی	$O(n^2)$
جستجوی خطی به روش بازگشتی	$O(n)$	مرتب‌سازی ادغامی	$O(n \log n)$
جستجوی باینری به روش بازگشتی	$O(\log n)$	مرتب‌سازی حبابی	$O(n^2)$
		مرتب‌سازی سریع (quick sort)	$O(n^2)$ *در بدترین حالت*
			$O(n \log n)$ *حالت میانگین*

شکل 8–20 | الگوریتم‌های جستجو مرتب‌سازی با مقادیر Big O.