Лекция 10-11

СҰРЫПТАУ ЖӘНЕ ІЗДЕУ

СҰРЫПТАУДЫҢ НЕГІЗГІ АЛГОРИТМДЕРІ

Деректерді іздеу әлемде ерекше орынға ие алгоритм есебі болып табылады. Есептің негізінде векторда сақталған элементті іздеу жатыр, яғни деректер жиынында сақталған қажетті деректерді табу талап етіледі, *мысалы*, телефон кітапшасынан адамдарды іздеу.

Информатика саласында да, шынайы өмірде де іздеу процестері ізделінді мәліметтерді табу шарттарына сәйкес әдістер бойынша ерекшеленеді. Егер деректер арнайы реттілікпен орналаспаса, онда мәліметтерді іздеудің ең тиімді тәсілі *деректер жиынын іріктеу арқылы іздеу* болып табылады. Ол ізделінді деректі тапқанша жиындағы (***front-to-back***) әрбір элементті зерттейді. Егер деректер жиыны аз болса онда мұндай шешім тиімді болып келеді.

Толықтай араластырылған карта ішінен белгілі бір картаны алу қажет деп тұжырымдайық. Тек елу екі картаның барлығын кезекпен оң қаратып, іріктеп (***front-to-back***) қажетті картаны табуға болады. Бірақ деректердің үлкен жиыны іздеу тәсілінің тиімді әдістерін қолдануды талап етеді.

Мысалы, адамдардың тізімі кездейсоқ ретпен орналасқан телефон кітапшасынан адамдардың телефон нөмірін іздеу қажет деп тұжырымдайық. Әрбір тізімді зерттеп отырсақ, бірнеше күн кетуі ықтимал. Сондықтан бұл тиімді тәсіл болып табылмайды және телефон кітапшасында адамдардың есімдерінің алфавит бойынша орналасуының себебін түсіндіреді. Телефон кітапшасында тізімнің орналасу ретін білгендіктен біз іздеудің тиімді әдістерін қолданамыз. Бұл іздеу әдісі телефон кітапшасын ашып кітап парақшасындағы тізімді ізделінді дерекпен салыстырамыз.

Егер біз "***Doe, John-ды***" іздеген болсақ, ал біз ашқан кітап парақшасы тек "***Smith***" тізімінен тұрады, олай болса, біз ізделінді есімге жақын парақшаны ашамыз, осылайша ізделінді ақпаратқа жеткенше іздеуді жалғастыра береміз. Осы екі әдіске информатикада сәйкесінше іздеу алгоритмдері құрылады. ***front-to-back*** тізбегін ***сызықты іздеу (linear search) алгоритмі*** деп атайды. Іздеудің кең таралған алгоритмдерінің бірі – ***бинарлы (екілік) іздеу алгоритмі*** (binary search).

Сызықты іздеу алгоритмі

Сызықты іздеу алгоритмі қажетті элементтерді тізбектей реттілікпен сұрыптай отырып табатын іздеудің қарапайым алгоритмі болып табылады. Сондықтан сызықты іздеу тәсілін кей жағдайда ***тізбектей іздеу әдісі*** деп атайды. C++ тілінде сызықты іздеуді бірнеше программалық кодтарды теріп, цикл көмегімен орындай аламыз.

```
 1: // Finding an element in a vector using linear search
 2: template <typename T>
 3: int linear_search(const vector<T>& v, const T& item) {
 4:
 5:     for (int i = 0; i < v.size(); i++) {
 6:         if (v[i] == item) {
 7:             return i; // item found
 8:         }
 9:     }
10:     return -1; // item not found
11: }
```

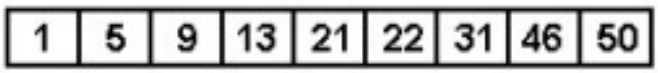
Алгоритмнің орындалуының қарапайымдылығынан бөлек, сызықты іздеу алгоритмінің ***негізі ерекшелігі*** ол деректердің сұрыпталуын талап етпейді. Егер сақталған деректер жиыны жадыда кез келген ретпен орналасса да сызықты іздеу алгоритмі өз жұмысын орындайды. Сызықты іздеу алгоритмі қажетті мәліметке жеткенше әрбір элементті іріктейді. Мұндай іздеу алгоритмінің кемшілігі, ол тек аз мәліметтер жиыны үшін тиімді орындалады

STL find функциясы сызықты іздеуді контейнерде жүзеге асырады. Бұл функция үш параметрден тұрады. Алғашқы екі параметр диапазонды анықтайтын ***iterators*** болып табылады. Үшінші параметр функция табуға тырысып жатқан шама болып табылады. Бұл функция ізделінді шама сақталған бірінші позицияны ***iterator-***ды қайтарады. Егер амал сәтсіз орындалса, онда ***find*** функциясы екінші позицияға тең ***iterator-***ды қайтарады.

Binary іздеу алгоритмі

Binary іздеу алгоритмі жылдам іздеу алгоритмі болып табылады және кез келген өлшемді деректер жиыны үшін тиімді орындалады. Сызықты іздеу алгоритмінен айырмашылығы үлкен өлшемді деректерді де жылдам іздей алады, тағы бір айырмашылығы ізделінетін деректердің сұрыпталғанын талап етеді.

Келесі вектордағы элементтерді іздеу мысалын қарастырайық.



1-сурет

9-ға тең болатын элементті табу керек. Барлық бинарлы іздеу қадамы деректер жиынының ортасында тұрған элементті іздеуден басталады. Бұл мысалда ондай элемент 21-санына тең болып келеді.

2-сурет

Біз іздеп отырған шама 9-ға тең. 9 саны 21-ден кіші болғандықтан оның орта шама орналасқан орынның сол жағында сақталуы тиіс екенін білеміз. Сондықтан бір вектордың оң жақ бөлігін жоққа шығарып, сол жақ бөлігін сұрыптауға кірісеміз. 3-сурет векторларды бөліктеу мысалы болып табылады.

3-сурет

Вектордың сол жағын орта шамасы бойынша бөлеміз. Ол 5 санына тең. 5 саны 9-дан кіші болғандықтан оның орта шама орналасқан орынның оң жағында сақталуы тиіс екенін білеміз. Сондықтан бір вектордың сол жақ бөлігін жоққа шығарып, оң жақ бөлігін сұрыптауға кірісеміз. Осылайша, қажетті элементті табамыз.

4-сурет

9 элементтен тұратын берілген векторды қажетті элементті табу үшін тек үш салыстыру жасалды. Вектордың сол жағынан бастап сызықты іздеу алгоритмі де үш салыстыру жасауы тиіс. Екілік іздеу алгоритмінің артықшылығы үлкен деректер типімен жұмыс жасағанда көрінеді.

Келесі кестеде бинарлы алгоритм әртүрлі типтегі элементтерді табуы үшін максималды салыстыру амалдарын орындайды.

1-кесте

Size of vector	Max comparisons
10	4
100	7
1,000	10
10,000	13
100,000	17
1,000,000	20
10,000,000	23
100,000,000	27
1,000,000,000	30

Тіпті миллиард элементтен тұратын вектор үшін де бинарлы іздеу алггоритмі максимум отыз салыстыру амалын жасайды.

Бинарлы іздеу алгоритмінің орындалу програмасы төменде келтірілген.

```
 1:  // Finding an element in a vector using binary search
 2: template <typename T>
 3: int binary_search(const vector<T>& v, const T& item) {
 4:
 5:     int low = 0;
 6:     int high = v.size() - 1;
 7:     int mid;
 8:
 9:     while (low <= high) {
10:
11:         // find the midpoint
12:         mid = (low + high) / 2;
13:
14:         if (v[mid] < item) {
15:             low = mid + 1; // look in upper portion
16:         }
17:         else if (v[mid] > item) {
18:             high = mid - 1; // look in lower portion
19:         }
20:         else {
21:             return mid; // found it!
22:         }
23:     }
24:
25:     return -1; // item not found
26: }
```

2-листинг

Екілік іздеу алгоритмінің кемшілігі ізделінетін деректер сұрыпталған болуы тиіс. Егер мәліметтер сұрыпталмаса онда нәтижесі дұрыс болмас еді.

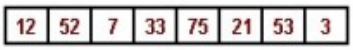
C++ тілінде ***STL binary_search*** функциясын қолдану ұсынылады. ***STL binary_search*** функциясының интерфейсі ***find*** функциясының интерфейсіне ұқсас болып келеді. Айырмашылығы *binary_search* функциясы нәтижесі дұрыс болса ***bool*** мәнін береді.

Сұрыптаудың негізгі алгоритмдері

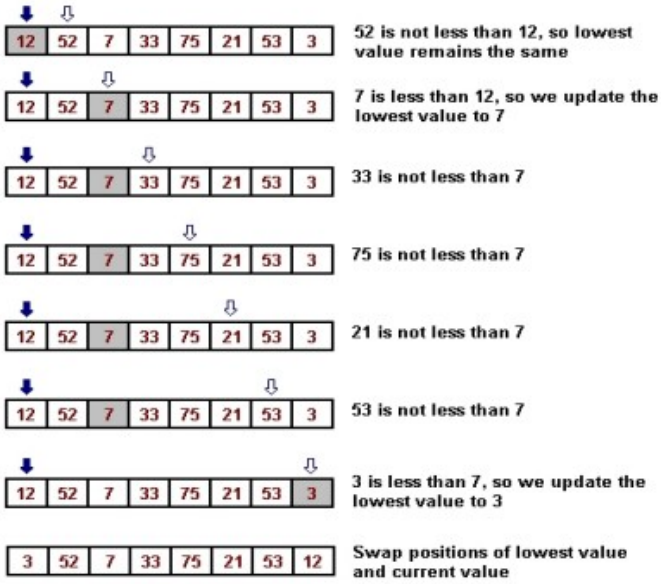
Сұрыптаудың көптеген алгоритмдері кездеседі. Әрбір алгоритмнің өзіндік сипаттамалары, орындалу тәртібі және талабы бар. Мысалы, бір есепті шығару үшін бір алгоритм екінші алгоритмге қарағанда аз салыстыру жасауы мүмкін. Немесе орындалу барысында бір алгоритм деректер элементтерінің қатарын екінші алгоритмге қарағанда аз шақыруы мүмкін.

Selection Sort

Selection sort алгоритмі итерация арқылы немесе деректерді сұрыптау арқылы орындалатын сұрыптаудың негізгі алгоритмі болып табылады. Әрбір сұрыптау барысында бір элементті өз орнына дұрыс тасымалдауға алып келеді. Нәтижесінде әрбір келесі тасымал сайын алдыңғы тасымалға қарағанда бір элементке кем зерттеу жасауға мүмкіндік алады.



5-сурет



6-сурет

Сегіз элементтен тұратын сұрыпталмаған массив қарастырайық, осы элементтерді сұрыптау үшін ***selection sort*** алгоритмі жеті тасымал орындауы тиіс.

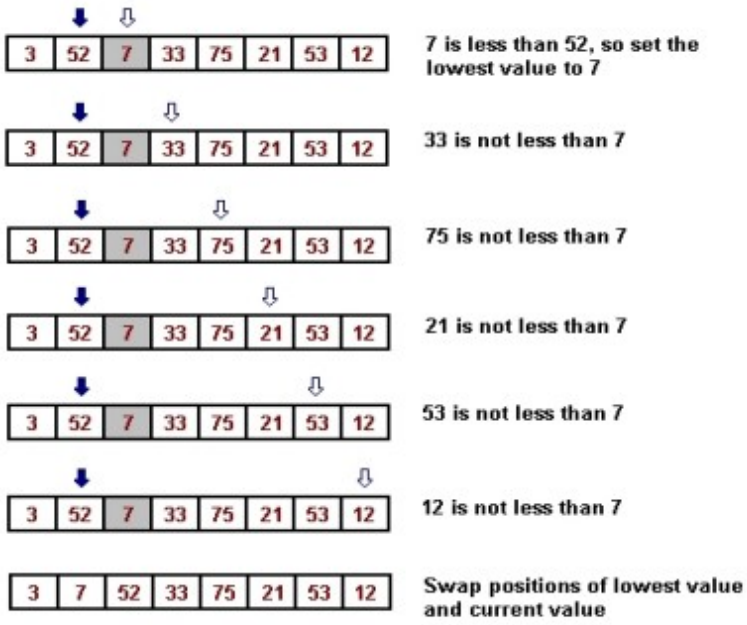
Әрбір тасымал бір элементті дұрыс позицияға орналастырады. Алғашқы екі сандар ұяшығы алғашқы екі тасымалды орындайды.

Әрбір қатардағы боялған бағыттауыш толтырылуы тиіс позицияны нұсқайды.

Әрбір тасымал кезінде алгоритм ең кіші қалдырылатын элементті іздейді.

Барлық бағыттауыш боялған кезде алгоритм тоқтайды.

Бірінші тасымалдан соң ең кіші шама массивтің бірінші позициясына орналастырылады. Келесі тасымал 7-суретте көрсетілгендей екінші ең кіші элемент массивтің екінші позициясына орналастырылады.



7-сурет

Selection sort алгоритмі алдыңғы қадамға қарағанда бір тасымалды кем жасайды. Барлық элементтер қарастырылғанша массивтерді сұрыптау жұмысын жалғастыра береді. ***Selection sort*** алгоритмінің C++ тілінде орындалу коды 3-листингте бейнеленген.

```
// Sorting using selection sort
template <typename T>
void selection_sort(vector<T>& v) {

    for(unsigned int i = 0; i < v.size() - 1; i++) {
        unsigned int best = i;
        for (unsigned int j = i + 1; j < v.size();
j++) {
            if (v[j] < v[best]) {
                best = j;
            }
        }

        if (best != i) {
            T temp = v[i];
            v[i] = v[best];
            v[best] = temp;
        }
    }
}
```

3-листинг

Сұрыптаудың жылдам алгоритмдері

Бұрын қарастырылған алгоритмдерден ерекше сұрыптаудың бірнеше алгоритмдерінің тізбегі бар. «Жылдам» сұрыптау алгоритмдері ***quicksort*** және ***Shellsort*** бірігуінен сұрыптайды. ***Quicksort*** алгоритмі кең таралған алгоритм болғандықтан, осы алгоритмді терең зерттейміз.

Quicksort

Quicksort алгортмі ***divide-and-conquer*** (бөлікте де басқар) проблемасын қолданатын сұрыптаудың жылдам алгоритмі болып табылады. ***Quicksort*** алгоритмі – массив элементтерін қарапайым түрге келтіретін рекурсия, яғни массивтерді ұсақ бөліктерге бөлу алгоритмі.

Quicksort алгоритмі ұсақ массивтерді сұрыптайды және оларды ағымдағы массивтің сұрыпталған нұсқасына біріктіреді. Рекурсивті сипаттамаларына қарай ***quicksort*** алгоритмін түсіну қиын болып келеді. Алгоритмді орындамас бұрын оның идеясын қарастырайық.

Quicksort алгоритмі келесі төрт негізгі қадам арқылы алынады:

1. Егер массив өлшемі нөлге тең болса немесе бір элементтен тұрса, онда массив сұрыпталған. Бұл негізгі жағдай болып табылады.

2. Массивтің ортасынан элемент таңдаймыз және оны массив центрі ретінде қолданамыз. Бұл центрді таңдау қадамы болып табылады.

3. Екі жаңа массив құрылады. Центр элементінің сан мәнінен кем болатын элементтерді сол жаққа бір массивке, ал центр элементінің сан мәнінен артық болатын элементтерді оң жаққа бір массивке жинақтаймыз. Бұл – бөліктеу қадамы.

4. Центр элементінен кем элементтерден тұратын массивті тағы да екі массивке бөлеміз. Бұл – бөлудің рекурсивті қадамы.