Introduction to Algorithms and Data Structures

4. Searching (2) Block search

Professor Ryuhei Uehara,
School of Information Science, JAIST, Japan.
uehara@jaist.ac.jp
http://www.jaist.ac.jp/~uehara

Search Problem

- Problem: *S* is a given set of data. For any given data *x*, determine efficiently if *S* contains *x* or not.

- Efficiency: Estimate the time complexity by $n = |S|$, the size of the set *S*
  - In this problem, "checking every data in S" is enough, and this gives us an upper bound O(*n*) in the worst case.

Roughly, "the running time is proportional to *n*."

Data structure 2
Data in the array in increasing order

- s[]=

3	9	12	25	29	33	37	65	87

- This is something like dictionary and address book…

Q: Do you use sequential search algorithm when you check dictionary?

No!

Drastic Improvement from O(n)

Algorithm 2: m-block method

Idea of m-block method

(0) Divide the array into m blocks $B_0, B_1, \ldots, B_{m-1}$

(1) Check the biggest item in each block,
and find the block B_j that can contain x

(2) Perform sequential search in B_j

Algorithm 2: m-block method

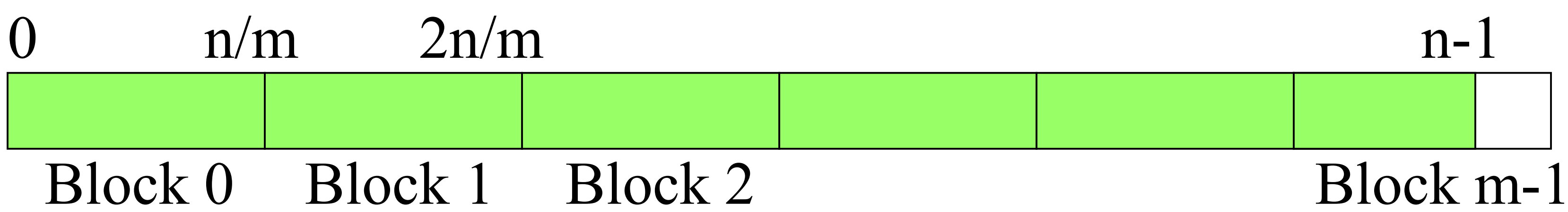
Idea of m-block method

(0) Divide the array into m blocks $B_0, B_1, \ldots, B_{m-1}$

(1) Check the biggest item in each block,
and find the block B_j that can contain x

(2) Perform sequential search in B_j

Simple implementation:
divide into the blocks of same size except the last one.



- Each block has length k, where $k = \lceil n/m \rceil$
- Block B_j has items from s[jk] to s[(j+1)k-1]: $B_j = [jk, (j+1)k-1]$

Algorithm 2: m-block method

Idea of m-block method

(0) Divide the array into m blocks $B_0, B_1, \ldots, B_{m-1}$
(1) Check the biggest item in each block, and find the block B_j that can contain x
(2) Perform sequential search in B_j

```
j=0;
while(j<=m-2)
   if x<=s[(j+1)*k-1] then exit from loop
   else j=j+1;
```

j=0,…,m-2,   m-1 is "leftover"

The maximum index of B_j

If the program exits from the loop, the variable j indicates the index of the block, and j indicates the last one otherwise.

Algorithm 2: m-block method

Idea of m-block method

(0) Divide the array into m blocks $B_0, B_1, \ldots, B_{m-1}$

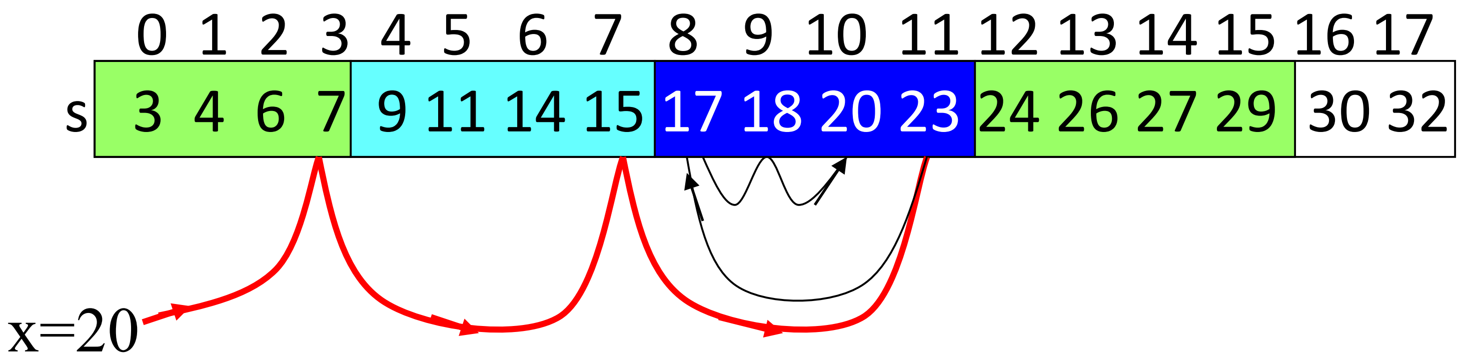
(1) Check the biggest item in each block,
and find the block B_j that can contain x

(2) Perform sequential search in B_j

```
i=j*k; t = min{ (j+1)*k-1, n-1 };
while( i < t )
  if x<=s[i] then exit from the loop;
  else i=i+1;  //next item in the block
if x == s[i] then return i and halt;
else  return -1 and halt.
```

Note that we cannot use sentinel since we have no extra space between block

Example and time complexity



- # of comparisons ≦ # of blocks + length of block = m + n/m
- What the value of m that minimize m + n/m ?
  - Let f(m) = m + n/m, and take the differential for m
  - $f'(m) = 1 - n/m^2 = 0 \rightarrow m = \sqrt{n}$
  - When $m = \sqrt{n}$, # of comparisons ≦ $\sqrt{n} + n/\sqrt{n} = 2\sqrt{n}$
- Time complexity: $O(\sqrt{n})$

5 min. ex.
Assume n=100.
Find "average" and "worst" cases for m=10, m=2, and m=50

For example, when n=1000000,
Linear search takes n/2=<u>500000</u> comparisons, but
Block search takes $\sqrt{1000000}$=<u>1000</u> comparisons!!

Example: Real code of m block method

```
public class i111_03_p27{
    public static void Main(){
         int[] data = new int[]{3,9,12,25,29,33,37,65,87};
         … the same as p7 … }

    static int find(int x, int n, int[] s) {
        int m=3;
        int k=(n-1)/m +1;

        int j=0;
        while (j<=m-2) {
            System.Console.Write(((j+1)*k-1)+" ");
            if (x<=s[(j+1)*k-1]) break;
            j++;
        }

        int i=j*k;
        int t=System.Math.Min((j+1)*k-1, n-1);
        while(i<t) {
            System.Console.Write(i+" ");
            if (x<=s[i]) break;
            i++;
        }
        if (x==s[i]) return i;
        return -1;
    }
}
```

【Pretty maniac】

Discussion of m block method

- Lengths of blocks should be the same?

(Observation)

# of comparisons ＝ # of searched blocks

＋ # of comparisons in the block

When you find former block, you can use more time in the block

➔It is better to decrease the length of blocks

- For example, we set $|B_{i+1}|=|B_i|-1$
- Make "index"+"length of a block" constant

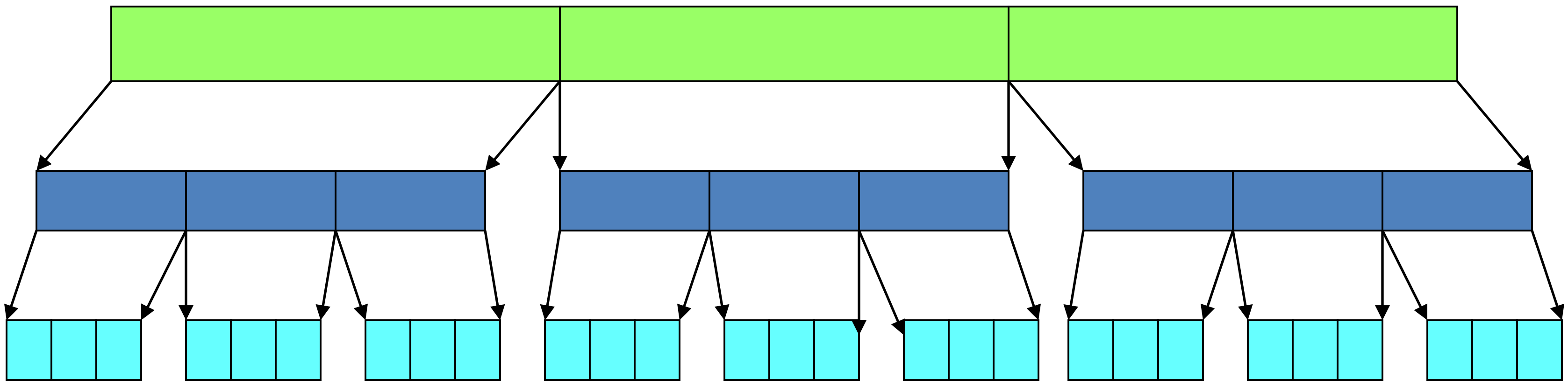
In reality, this kind of method of decreasing "unevenness" is preferred.

Can we do better than $O(\sqrt{n})$?

Algorithm 3: Double m-block method

In the m-block method, we use sequential search in each block.

➡ We can use m-block method again in the block!!



Idea of double m-block method

For example, if the number of data is 27,

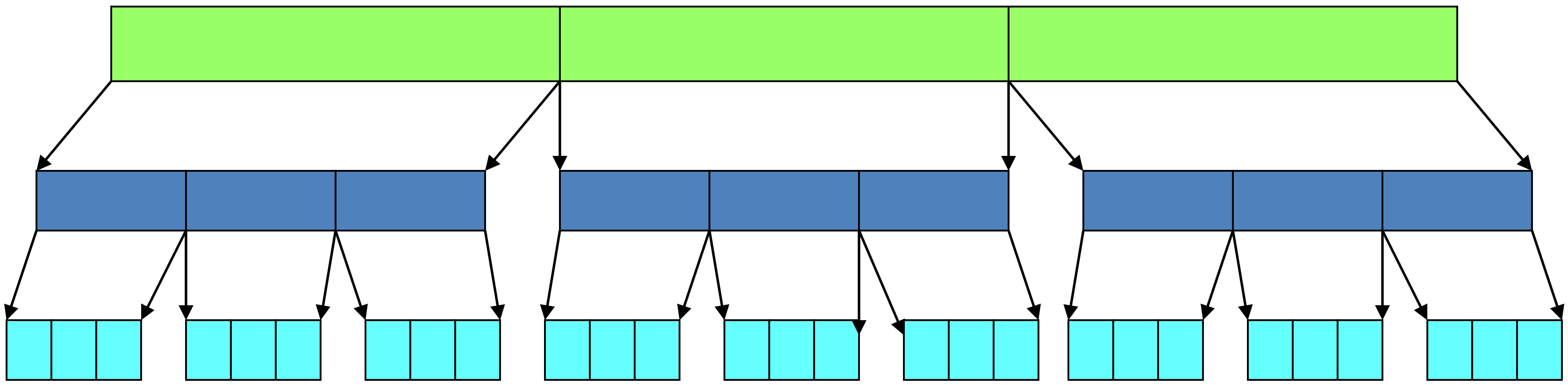
- Linear search requires 27 in the worst case
- 3-block method requires at most 3+9
- Double 3-block method needs at most 3+3+3

Algorithm 3: Double m-block method

In the m-block method, we use sequential search in each block.

➡ We can use m-block method again in the block!!

Recursive call: basic and **strong** idea

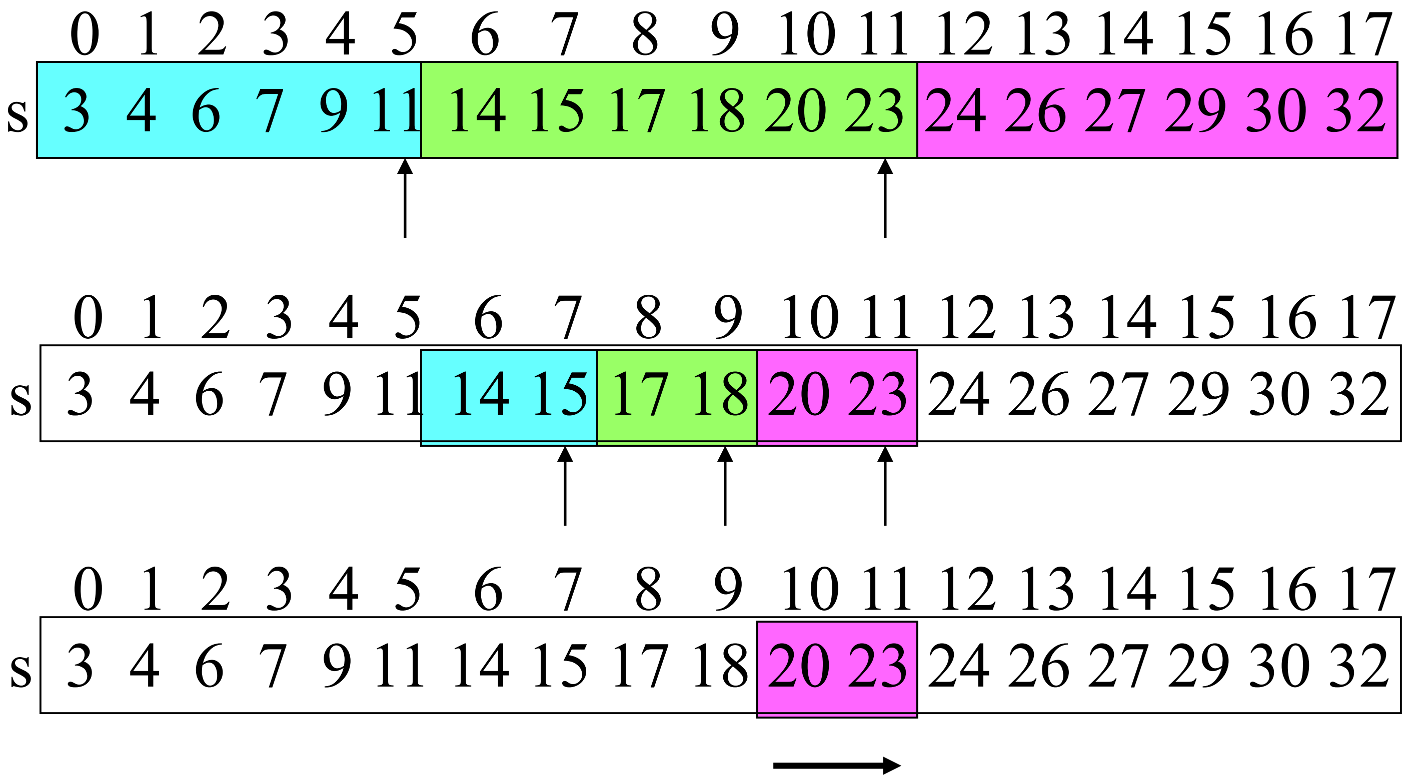


Idea of double m-block method

Why we stop only twice? We can more!!

Divide search area into m blocks, and repeat the same process for the block that contains x, and repeat again and again up to the block has length at most some constant N

Example:
find 20 (x=20) for block size 3



【I don't ask you to compute it by yourself…】

Analysis of time complexity

- Length of search space

$$n \to \left\lceil \frac{n}{m} \right\rceil \to \left\lceil \frac{\left\lceil \frac{n}{m} \right\rceil}{m} \right\rceil \to \left\lceil \frac{\left\lceil \frac{\left\lceil \frac{n}{m} \right\rceil}{m} \right\rceil}{m} \right\rceil \to \cdots$$

- Let n_i be the length after the i-th call

$$n_1 = \left\lceil \frac{n}{m} \right\rceil \leq \frac{n}{m} + 1$$

$$n_2 = \left\lceil \frac{n_1}{m} \right\rceil \leq \frac{n}{m^2} + \frac{1}{m} + 1$$

$$\cdots$$

$$n_i \leq \frac{n}{m^i} + \sum_{j=0}^{i-1} \frac{1}{m^j} \leq \frac{n}{m^i} + 2$$

【I don't ask you to compute it by yourself…】

Analysis of time complexity

- The length n_i after the i-th recursive call:

  $n_i \leqq n/m^i + 2$

- How many recursive calls made?

$$n_i \le \mathrm{Lmin} \Longleftarrow \mathrm{Lmin} \ge \frac{n}{m^i} + 2 \Longleftrightarrow i \ge \log_m \frac{n}{\mathrm{Lmin} - 2}$$

- Each recursive call make at most m-1 comparisons, so the total number of comparisons is $\le (m-1)\log_m \frac{n}{\mathrm{Lmin}-2} + \mathrm{Lmin}$

- The time complexity is O(log *n*)

【I don't ask you to compute it by yourself…】

Analysis of time complexity: The best value of m

- $$\mathrm{T}(n, m) = (m-1)\log_m \frac{n}{\mathrm{Lmin}-2} + \mathrm{Lmin}$$
$$= \frac{m-1}{\log_2 m}\log_2 \frac{n}{\mathrm{Lmin}-2} + \mathrm{Lmin}$$
- To make T(n,m) the minimum, smaller m is better because m-1 grows faster than $\log_2 m$ (which will be checked in the big-O notation).
- Therefore, m=2 is the optimal

We will have "binary search"

[Summary]

- For unorganized data, we have to use O(n) time.
- If data are sorted in increasing order,
  - We can exit from the loop when we find the position of x
  - Improved to $O(\sqrt{n})$ with m-block method with $m=\sqrt{n}$
  - Improved to O(log n) with doubly m-block method with m=2
- Honestly, in recent programming environment, you do not need to make such a search by yourself.
- Usually, we use a function indexOf(). However, it is very important that you should know that
  - "indexOf is heavy" for unorganized data
  - "indexOf is light" for SortedList