# Search and Sort

- Searching, which is the process of finding a particular element in an array
- Sorting, which is the process of rearranging the elements in an array so that they are stored in some well-defined order

# 搜索

- 在一个序列中找到某个数的位置（或确认是否存在），或找到某个特殊的数
- 基本方法：遍历

```
int max(int a[], int length):
    int r = a[0]        //    用第一个元素做种子
    for ( int i=1; i<length; i++ ) {
        if (a[i]>r) {
            r = a[i];
        }
    }
    return r;
}
```

# Test Bed for Searching

```c
int search(int key, int a[], int len);

void judge(char *prompt, int result, int std) {
    printf("%s %s: %d=%d\n", prompt, result==std?"PASS":"FAIL", result, std);
}

int main(void) {
    int a[] = {1,3,5,7,9,11,13,15,19,21};
    int len = sizeof(a)/sizeof(a[0]);
    judge("in", search(a[5], a, len), 5);
    judge("not in", search(14, a, len), -1);
    judge("first", search(a[0], a, len), 0);
    judge("last", search(a[len-1], a, len), len-1);
}
```

## Searching in an integer array

- Function `search` looks for the integer key in an array

```
int search(int key, int a[], int len)
{
    int r = -1;
    for ( int i=0; i<len; i++ ) {
        if ( key == a[i] ) {
            r = i;
            break;
        }
    }
    return r;
}
```

## Linear Search

- The algorithm used here is called the linear search algorithm.
- The search starts at the beginning of the array and goes straight down the line of elements until it finds a match or reaches the end of the array.
- In the worst case, it iterates $N$ times.

## 猜数游戏

- 如果你想一个100以内的正整数，我来猜，每猜一个数，你得告诉我这个数是偏大还是偏小，我可以在7次以内猜中

| 序号 | 城市 | 序号 | 城市 | 序号 | 城市 |
|---|---|---|---|---|---|
| 0 | Anshan | 9 | Jinan | 18 | Suzhou |
| 1 | Beijing | 10 | Kunming | 19 | Taiyuan |
| 2 | Changchun | 11 | Lanzhou | 20 | Ürümqi |
| 3 | Dalian | 12 | Mianyang | 21 | Wuhan |
| 4 | Erenhot | 13 | Nanjing | 22 | Xi'an |
| 5 | Fuzhou | 14 | Ordos | 23 | Yinchuan |
| 6 | Guangzhou | 15 | Putian | 24 | Zhengzhou |
| 7 | Hangzhou | 16 | Qamdo | | |
| 8 | Ili | 17 | Rizhao | | |

## Binary search

- 对于已经排序的数据，可以使用二分法
- 每次取中间位置上的数来测试，如果比目标大，则在低半段中搜索，否则在高半段中搜索
- 直到…
  - 找到，或
  - 被测数据不存在

$$f(s,x) = \begin{cases} can\ not\ find & ; empty\ s \\ mid & ; s[mid] == x \\ f(lower\ half\ of\ s, x) & ; s[mid] > x \\ f(higher\ half\ of\ s, x) & ; s[mid] < x \end{cases} mid = middle\ location\ of\ s$$

```c
int search_bin(int key, int a[], int begin, int end)
{
    int ret = -1;
    if ( begin<=end ) {
        int m = (begin + end)/2;
        printf("%d %d mid=%d ", begin, end, a[m]);
        if ( key < a[m] ) {
            printf("LOWER\n");
            ret = search_bin(key, a, begin, m-1);
        } else if ( key > a[m] ) {
            printf("UPPER\n");
            ret = search_bin(key, a, m+1, end);
        } else {
            printf("BINGO\n");
            ret = m;
        }
    } else {
        printf("FAILED\n");
    }
    return ret;
}
```

## Lab 1

- PTA 6-1 二分法查找

# Efficiency of the search algorithms

- To search an array of N elements requires $N$ comparisons if you use linear search and $log_2N$ comparisons if you use binary search.
- The following table shows the closest integer to $log_2N$ for various values of $N$.

| $N$ | $log_2N$ |
|---|---|
| 10 | 3 |
| 100 | 7 |
| 1000 | 10 |
| 1000000 | 20 |
| 1000000000 | 30 |

## Sort

- How can we sort a unsorted array into sorted?


03
17
19
23
04
08
25
20
21
15
22
16
01
07
09
12
02
13
05
11
06
24
10
14


19
17
23
10
25
01
20
15
22
02
21
03
13
16
04
14
12
06
09
07
05
08
24
11


01
02
03
04
05
06
07
08
09
10
11
12
13
14
15
16
17
19
20
21
22
23
24
25

```c
#define SIZE 100

void sort(int a[], int begin, int end);

int main()
{
    int a[SIZE];
    srand(0);
    for ( int i=0; i<SIZE; i++ ) {
        a[i] = rand()%SIZE;
        printf("%d\n", a[i]);
    }
    sort(a, 0, SIZE-1);
    for ( int i=0; i<SIZE; i++ ) {
        printf("%d\n", a[i]);
    }
    int r = 1;
    for ( int i=1; i<SIZE; i++ ) {
        if ( a[i] < a[i-1] ) {
            r=0;
            break;
        }
    }
    printf("%s\n", r?"PASS":"FAIL");
}
```

- 如果找到这里的最小的数，把它和第一个位置的数做交换，就落实了最小的数的位置

```c
int findmin(int a[], int len) {
    int minidx = 0;
    for ( int i=1; i<len; i++ ) {
        if ( a[i]<a[minidx] ) {
            minidx = i;
        }
    }
    return minidx;
}

int loc = findinx(lst)
int t = a[loc];
a[loc] = a[0];
a[0] = t;
```

- 如果我已经会做排序了：`sort(array, begin, end)` 能对`array`中`begin`到`end`之间的数据做排序
- 那么，找到`array`中最小的，与`begin`位置上的数据做交换
- 然后 `sort(array, begin+1, end)`
- 直到`begin==end`

```c
void sort(int a[], int begin, int end)
{
    if ( begin < end ) {
        int loc = findmin(a, begin, end);
        int t = a[begin];
        a[begin] = a[loc];
        a[loc] = t;
        sort(a, begin+1, end);
    }
}
sort(a, 0, len);
```

# 尾递归优化为循环

```
void sort(int a[], int begin, int end)
{
    while ( begin<end ) {
        int loc = findmin(a, begin, end);
        int t = a[begin];
        a[begin] = a[loc];
        a[loc] = t;
        begin += 1;
    }
}
```

## while --> for

```c
void sort(int a[], int begin, int end)
{
    for ( ; begin < end; begin++ ) {
        int loc = findmin(a, begin, end);
        int t = a[begin];
        a[begin] = a[loc];
        a[loc] = t;
    }
}
```

# 不再需要传入begin和end

```
void sort(int a[], int len)
{
    for ( int i=0; i<len; i++ ) {
        int loc = findmin(a, i, len);
        int t = a[i];
        a[i] = a[loc];
        a[loc] = t;
    }
}
```

# 把findmin放进来

```c
void sort(int a[], int len)
{
    for ( int i=0; i<len; i++ ) {
        //  int loc = findmin(a, i, len);
        int loc=i;
        for ( int j = i+1; j<len; j++ ) {
            if ( a[j] < a[loc] ) {
                loc = j;
            }
        }
        //  swap
        int t = a[i];
        a[i] = a[loc];
        a[loc] = t;
    }
}
```

## 选择排序

- 每次找出最大的值，和数列最后的值交换
- 保持最后的值不动，对剩下的值重复这个过程，直到数列只剩下一个数

*也可以是*

- 每次找出最小的值，和数列第一个值交换
- 保持第一个值不动，对剩下的值重复这个过程，直到数列只剩下一个数

## Task

- 每次寻找最大的，放到最后面

## Lab 2

- PTA 6-2 选择排序

## 算法的性能

- 算法运行的时间和所需的空间表达了算法的性能
- 性能主要由程序中循环的次数所决定
- 选择排序有两重循环，每一重的循环次数都和数据集的大小$N$正相关
- 因此它的性能可以表达为$O(N^2)$

# 冒泡排序

- 遍历数据，如果发现相邻的数据的大小关系不对，就交换这两个数据
- 这样一遍遍历下来，就能把最大的数据推到数列的末尾（冒泡）

```
void sort_bubble(int a[], int size)
{
    for ( int j=0; j<size-1; j++ ) {
        if ( a[j] > a[j+1] ) {
            int t = a[j];
            a[j] = a[j+1];
            a[j+1] = t;
        }
    }
}
```

- 确定了最后一个数之后，对剩下的数列做重复的动作
- 直到数列中只剩下一个数

```
void sort_bubble(int a[], int size)
{
    if ( size>1 ) {
        for ( int j=0; j<i; j++ ) {
            if ( a[j] > a[j+1] ) {
                int t = a[j];
                a[j] = a[j+1];
                a[j+1] = t;
            }
        }
        sort_bubble(a, size-1);
    }
}
```

# 尾递归优化

```c
void sort_bubble(int a[], int size)
{
    for ( int i=size-1; i>0; i-- ) {
        for ( int j=0; j<i; j++ ) {
            if ( a[j] > a[j+1] ) {
                int t = a[j];
                a[j] = a[j+1];
                a[j+1] = t;
            }
        }
    }
}
```

## 另一个版本

```c
void sort_bubble(int a[], int size)
{
    for ( int i=size-1; i>0; i-- ) {    //  i表示要冒到哪里
        for ( int j=0; j<i; j++ ) {
```

↓

```c
    for ( int i=0; i<size; i++ ) {      //  i表示已经冒了几个数
```

此时，接下去的代码要如何修改

# 优化

- 冒泡过程中有可能顺便把多个数据一起排好了顺序
- 如果能发现提前排好了，下一轮就可以少跑几个数据

```
void sort_bubble(int a[], int size)
{
    for ( int i=size-1; i>0; i-- ) {
        int loc = -1;
        for ( int j=0; j<i; j++ ) {
            if ( a[j] > a[j+1] ) {
                int t = a[j];
                a[j] = a[j+1];
                a[j+1] = t;
                loc = j;
            }
        }
        i = loc+1;
    }
}
```

## Lab 3

- PTA 6-3 冒泡排序

## Analysis of algorithms

- Evaluating the relative efficiency of algorithms will be a major topic that is usually called analysis of algorithms
- An algorithm that runs more quickly with one set of input values may turn out to run more slowly for others
- Some algorithms work well for a small amount of input data but deteriorate in performance when the amount of data becomes large

## Computational Complexity

- When evaluating algorithmic efficiency, using the letter N to represent the size of the problem
- The central question in analysis of algorithms is to determine how the running time changes as a function of N
- The relationship between N and the running time of an algorithm as N becomes large is called the computational complexity of that algorithm

## Big-O Notation of Selecting sort

- Eliminate any term in the formula that becomes insignificant as N becomes large.
  $O(\frac{n^2+n}{2})$ is not correct
- Eliminate any constant coefficients.
  $O(\frac{n^2}{2})$ is not correct
- The expression used to indicate the complexity of selection sort is:
  $O(n^2)$

## Quadratic Time

- Algorithms that exhibit $O(N^2)$ performance are said to run in quadratic time
- The basic characteristic of quadratic complexity is that, as the size of the problem doubles, the running time increases by a factor of four

| N | Running Time |
|---|---|
| 100 | 9.67 |
| 200 | 37.33 |
| 400 | 146.67 |
| 800 | 596.67 |

## Common O

- $O(logn)$: logarithmic O
- $O(n)$: linear O
- $O(nlogn)$
- $O(n^2)$

# What we've learned today?

- 线性搜索
- 搜索最值
- 二分搜索
- 选择排序
- 冒泡排序