Searching and Sorting

Abhiram Ranade

The search problem

Input: an array x (say int x[100];)and a value y

Search problem: Is x[i] == y for some i?

- Obvious algorithm idea: Compare every element x[i] with y and report if found.
- Question for today: Can we get by with fewer comparisons if x is sorted in non-decreasing order, i.e. values are such that

x[0] <= x[1] <= x[2] <= ...

Key idea

- First compare y with x[n/2], where n = length of x.
- If x[n/2] < y then we know y cannot be in the subarray x[0..n/2]. Suffices to search x[n/2 +1…n-1].
- if x[n/2] >= y then we know that it suffices to search the subarray x[0..n/2]
- In both cases, by doing a single comparison, we have halved the number of elements we need to search next.
- x[i…j] : short for x[i],x[i+1],…x[j-1],x[j]

Examples

x:

index	0	1	2	3	4	5	6	7	8
value	34	55	56	68	75	77	79	88	93

Example 1: y = 77. Compare y with x[4]. Outcome: x[4] < y. So we only need to search elements x[5]…x[8] next.

Example 2: y = 37. Compare y with x[4]. Outcome: x[4] >= y. So we only need to search elements x[0]…x[4] next.

Binary search of an array

- Invariant: in each iteration a portion of the array x[start..end] can possibly contain y. "feasible portion"
- In each iteration, we compare y with the "middle" element x[(start+end)/2].
- Based on the comparison result we adjust start and end. Feasible portion shrinks.
- When start == end, feasible portion has length 1. So we check if x[start] == y

Binary Search

```
int binSearch(int *x, int n, int y){
  int start = 0, end = n-1;
  while(start < end){
    int mid = (start+end)/2;
    if(x[mid] < y) start = mid + 1;
    else                end  = mid;
    }
  if(x[start] == y) return start;  else return -1;
}
```

Correctness

- Loop invariant: start, end are valid indices throughout execution. If y is in x, then it must be in subarray x[start..end].
- Base case: true at the beginning.
- mid = (start+end)/2. So start <= mid < end.
- So mid and mid+1 are both valid indices.
- Hence start, end valid after iteration.
- end – start always decreases, approximately halving. So $\log_2 n$ iterations needed.

Example

x has length 1024.

Length of feasible portion = 1024.

First probe: mid = (0+1023)/2 = 511

Next feasible portion: x[0..511] or x[512..1023]

Length of new feasible portion = 512.

Remarks

- Function can be written using recursion. Exercise.
- Similar to “20 questions”, “Bisection method”.
- Could have been used in marks display, if rollno array was stored in sorted order.
- It is useful to store sequences in sorted order if we are going to search them.

Sorting

- Input: array x, in which values are stored in any order.
- Output: rearrange values in x so that they appear in non-decreasing order.
- Extremely important operation. Useful in many, many algorithms.
- Many, many, pretty algorithms.
- Non-increasing order might also be demanded.

Example

before

index	0	1	2	3	4	5	6	7	8
value	75	93	68	88	34	77	79	56	55

after

index	0	1	2	3	4	5	6	7	8
value	34	55	56	68	75	77	79	88	93

Selection sort

- Find the largest element.
- Move it to the last position.
- Find the second largest element.
- Move it to the second last position.
- Find the third largest element.
- …

The index of the largest element

```
int maxIndex(int *x, int n){
  int result = 0;
  for(int i=1; i<n; i++)
    if(x[i] > x[result]) result = i;
  return result;
}
// int x[]={10,29,37,55,43,55}
// maxIndex(x,6) evaluates to 3.
```

Selection Sort Algorithm

```
void ssort(int *x, int n){
  for(int L=n; L>1; L--){
    int MI = maxIndex(x,L);
    int maxvalue = x[MI]; //exchange x[MI],x[L-1]
    x[MI] = x[L-1];
    x[L-1] = maxvalue;
  }
}
```

How good is selection sort?

- We will count the number of comparisons. This is indicative of total time taken.
- Number of comparisons:

To find largest : n-1.   (n = length of array)

To find second largest: n-2

To find third largest: n-3 …

- Total: n-1 + n-2 + … + 1 = (n-1)(n-2)/2
- Approximately proportional to n^2.

A faster algorithm: Merge sort

- Time to sort: proportional to $n \log_2 n$, i.e. much better than selection sort.
- Basic idea:
  - Suppose you have sorted arrays p, q of length m, n respectively.
  - Put the elements of p, q into an array r of length m+n such that r is sorted.
  - Can be done fast by exploiting the fact that p, q were originally sorted. "Merge"

How to merge

- p: students standing in a queue in increasing order of height. q: Another such queue.
- How can they get into a single queue r and still remain in increasing order of height?
- Shorter of the students at the head of queue p and queue q should enter queue r.
- Queues p, q move up as needed.
- Repeat process to move next shortest…

Example

p: {10, 13, 14, 17} q: {9, 16, 20, 25} r:{}

p: {10, 13, 14, 17} q: {16, 20, 25} r:{9}

p: {13, 14, 17} q: {16, 20, 25} r:{9, 10}

p: {14, 17} q: {16, 20, 25} r:{9, 10, 13}

p: {17} q: {16, 20, 25} r:{9, 10, 13, 14}

p: {17} q: {20, 25} r:{9, 10, 13, 14, 16}

p: {} q: {20, 25} r:{9, 10, 13, 14, 16, 17}

p: {} q: {25} r:{9, 10, 13, 14, 16, 17, 20}

p: {} q: {} r:{9, 10, 13, 14, 16, 17, 20, 25}

Summary

- If both queues have an element, smaller of the two moves to r.
- If only one queue has an element, then the element at its front moves to r.
- How to represent the queues: do not insist that the smallest element is at index 0, instead keep track of the index at which the front of the queue is. Similar to taxi dispatch.
- pf, qf: index of front of p,q. rb: back of r.

How to merge

```
void merge(int p[], int m, int q[], int n, int r[]){
  for(int rb=0, pf=0, qf=0;  rb<m+n; rb++){
    if( pf < m && qf < n){                          // both queues non-empty
      if (p[pf] <= q[qf]) { r[rb] = p[pf]; pf++; }
      else{ r[rb] = q[qf]; qf++; }
    }
    else if (pf < m){ r[rb] = p[pf]; pf++; }  // only p is non-empty
    else{ r[rb] = q[qf]; qf++; }                    // only q is non-empty
  }
}
```

Mergesort idea (informal)

r = sequence to be sorted. Length = n.

p = first n/2 elements of r

q = remaining elements of r

sort(p). sort(q).

Merge p, q to produce r.

```
void mergesort(int* r, int n){
  if(n>1){
    int p[n/2], q[n-n/2];
    for(int i=0; i<n/2; i++) p[i] = r[i];
    for(int i=n/2; i<n; i++) q[i-n/2] = r[i];

    mergesort(p, n/2);
    mergesort(q, n - n/2);
    merge(p,n/2, q,n-n/2,r);
  }
}
```

Why is mergesort fast?

- Analysis given in book.
- Analysis is outside the scope of the course.

Intuition:

- Selection sort performs comparisons to find smallest – the results are not used to help find second smallest faster.
- In mergesort, we remember.