Search algorithms
and
Informal introduction
to
asymptotic complexity

Organization

- Searching in arrays
  - Linear search
  - Binary search
- Asymptotic complexity of algorithms

Linear search

- Input:
  - unsorted array A of Comparables
  - value v of type Comparable
- Output: true if v is in array A, false otherwise
- Algorithm: examine the elements of A in some order till you either
  - find v: return true, or
  - you have unsuccessfully examined all the elements of the array: return false

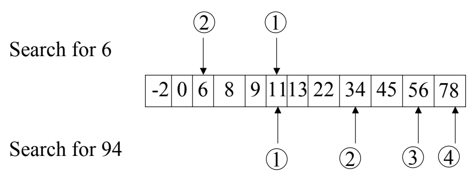
```
//linear search on possibly unsorted array
public static boolean linearSearch(Comparable[] a, Object v) {
  int i = 0;
  while (i < a.length)
    if (a[i].compareTo(v) == 0) return true;
    else i++;
  return false;
}
```

Linear search:

7	4	6	19	3	7	8	10	32	54	67	98

Binary search

- Input:
  - sorted array A[0..n-1] of Comparable
  - Value v of type Comparable
- Output: returns true if v is in the array; false otherwise
- Algorithm: similar to looking up telephone directory
  - Let m be the middle element of the array
  - If (m ==v) return true
  - If (m < v) search right half of array
  - If (m > v) search left half of array



Comparison of algorithms

- If you run binary search and linear search on a computer, you will find that binary search runs much faster than linear search.
- Stating this precisely can be quite subtle.
- One approach: asymptotic complexity of programs
  - big-O notation
- Two steps:
  - Compute running time of program
  - Running time → asymptotic running time

```
//left and right are the two end points of interval of array
 public static boolean binarySearch(Comparable[] a, int lo, int hi, Object v) {
        int  middle = (lo + hi)/2;
        int c = A[middle].compareTo(v);

        //base cases
        if (c == 0) return true;
        //check if array interval has only one element
        if (lo == hi) return false;

        //array interval has more than one element, so continue searching
        if (c  > 0) return binarySearch(a, lo, middle -1, v);  //left half
        else         return binarySearch(a, middle+1, hi, v); // right half
}
```

Invocation: assume array named data contains values

`….. binarySearch(data, 0, data.length -1, v)…..`

Running time of algorithms

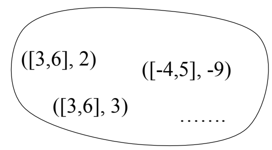
- In general, running time of a program such as linear search depends on many factors:
  1. machine on which program is executed
     - laptop vs. supercomputer
  2. size of input (array A)
     - big array vs. small array
  3. values of input
     - v is first element in array vs. v is not in array
- To talk precisely about running times of programs, we must specify all three factors above.

Defining running time of programs

1. Machine on which programs are executed.
   - Random-access Memory (RAM) model of computing
     - Measure of running time: number of operations executed
   - Other models used in CS: Turing machine, Parallel RAM model, …
   - Simplified RAM model for now:
     - Each data comparison is one operation.
     - All other operations are free.
     - Evaluate searching/sorting algorithms by estimating number of comparisons they execute
       - it can be shown that for searching and sorting algorithms, total number of operations executed on RAM model is proportional to number of data comparisons executed

Define running time (contd.)

3. Dependence of running time on input values



Possible inputs of size 2 for linear/binary search

- Consider set I_n of possible inputs of size n.
- Find number of comparisons for each possible input in this set.
- Compute
  - Average: hard to compute usually
  - Worst-case: easier to compute
- We will use worst-case complexity.

Defining running time (contd.)

2. Dependence on size of input
   - Rather than compute a single number, we will compute a function from problem size to number of comparisons.
     - (eg) $f(n) = 32n^2 – 2n + 23$ where is problem size
   - Each program has its own measure of problem size.
   - For searching/sorting, natural measure is size of array on which you are searching/sorting.

Computing running times

Linear search:

7	4	6	19	3	7	8	10	32	54	67	98

Assume array is of size n.
Worst-case number of comparisons: v is not in array.
Number of comparisons = n.
Running time of linear search: $T_L(n) = n$

Binary search: sorted array of size n

-2	0	6	8	9	11	13	22	34	45	56	78

Worst-case number of comparisons: v is not in array.

$T_B(n) = \lfloor \log_2(n) \rfloor + 1$

Running time → Asymptotic running time

Linear search: $T_L(n) = n$

Binary search: $T_B(n) = \lfloor \log_2(n) \rfloor + 1$

We are really interested only in comparing running times for large problem sizes.

- For small problem sizes, running time is small enough that we may not care which algorithm we use.

For large values of n, we can drop the "+1" term and the floor operation, and keep only the leading term, and say that $T_B(n) \rightarrow \log_2(n)$ as n gets larger.

Formally, $T_B(n) = O(\log_2(n))$ and $T_L(n) = O(n)$

Summary of informal introduction

- Asymptotic running time of a program
  1. Running time: compute worst-case number of operations required to execute program on RAM model as a function of input size.
     - for searching/sorting algorithms, we will compute only the number of comparisons
  2. Running time → asymptotic running time: keep only the leading term(s) in this function.

Rules for computing asymptotic running time

- Compute running time as a function of input size.
- Drop lower order terms.
- From the term that remains, drop floors/ceilings as well as any constant multipliers.
- Result: usually something like $O(n)$, $O(n^2)$, $O(n\log(n))$, $O(2^n)$, etc.