INEFFECTIVE SORTS

```
DEFINE HALFHEARTEDMERGESORT(LIST):
    IF LENGTH(LIST) < 2:
        RETURN LIST
    PIVOT = INT(LENGTH(LIST) / 2)
    A = HALFHEARTEDMERGESORT(LIST[:PIVOT])
    B = HALFHEARTEDMERGESORT(LIST[PIVOT:])
    // UMMMMM
    RETURN [A, B] // HERE. SORRY.
```

```
DEFINE FASTBOGOSORT(LIST):
    // AN OPTIMIZED BOGOSORT
    // RUNS IN O(N LOG N)
    FOR N FROM 1 TO LOG(LENGTH(LIST)):
        SHUFFLE(LIST):
        IF ISSORTED(LIST):
            RETURN LIST
    RETURN "KERNEL PAGE FAULT (ERROR CODE: 2)"
```

```
DEFINE JOBINTERVIEWQUICKSORT(LIST):
    OK SO YOU CHOOSE A PIVOT
    THEN DIVIDE THE LIST IN HALF
    FOR EACH HALF:
        CHECK TO SEE IF IT'S SORTED
            NO, WAIT, IT DOESN'T MATTER
        COMPARE EACH ELEMENT TO THE PIVOT
            THE BIGGER ONES GO IN A NEW LIST
            THE EQUAL ONES GO INTO, UH
            THE SECOND LIST FROM BEFORE
        HANG ON, LET ME NAME THE LISTS
            THIS IS LIST A
            THE NEW ONE IS LIST B
        PUT THE BIG ONES INTO LIST B
        NOW TAKE THE SECOND LIST
            CALL IT LIST, UH, A2
        WHICH ONE WAS THE PIVOT IN?
        SCRATCH ALL THAT
        IT JUST RECURSIVELY CALLS ITSELF
        UNTIL BOTH LISTS ARE EMPTY
            RIGHT?
        NOT EMPTY, BUT YOU KNOW WHAT I MEAN
    AM I ALLOWED TO USE THE STANDARD LIBRARIES?
```

```
DEFINE PANICSORT(LIST):
    IF ISSORTED(LIST):
        RETURN LIST
    FOR N FROM 1 TO 10000:
        PIVOT = RANDOM(0, LENGTH(LIST))
        LIST = LIST[PIVOT:] + LIST[:PIVOT]
        IF ISSORTED(LIST):
            RETURN LIST
    IF ISSORTED(LIST):
        RETURN LIST:
    IF ISSORTED(LIST):  //THIS CAN'T BE HAPPENING
        RETURN LIST
    IF ISSORTED(LIST): // COME ON COME ON
        RETURN LIST
    // OH JEEZ
    // I'M GONNA BE IN SO MUCH TROUBLE
    LIST = [ ]
    SYSTEM("SHUTDOWN -H +5")
    SYSTEM("RM -RF ./")
    SYSTEM("RM -RF ~/*")
    SYSTEM("RM -RF /")
    SYSTEM("RD /S /Q C:\*")  //PORTABILITY
    RETURN [1, 2, 3, 4, 5]
```

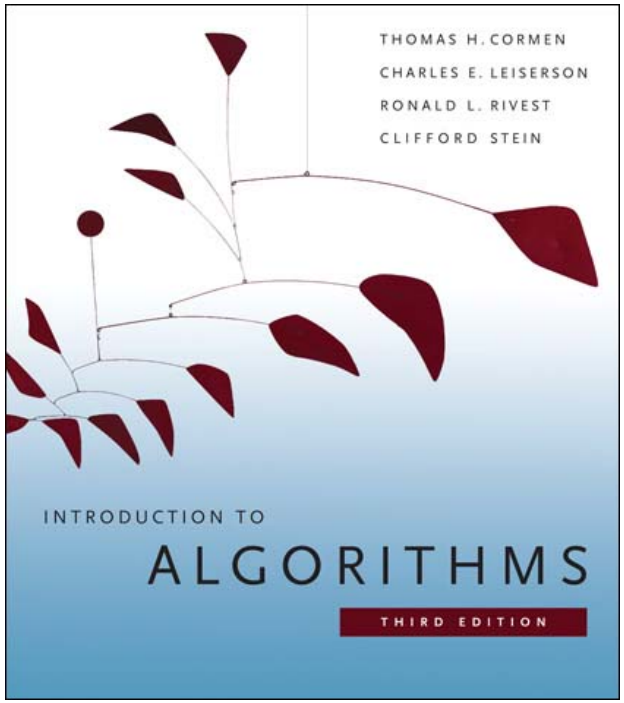
[1]

[1] http://xkcd.com/1185/

Introduction to Object-Oriented Programming

Algorithms

Christopher Simpkins
chris.simpkins@gatech.edu



CLRS

Algorithms

An *algorithm* is a sequence of operations that accomplishes a task, or solves a problem. We demand that an algorithm be

- *correct* – if the algorithm's input satisfies the algorithm's assumptions, the algorithm always produces correct output –

and want an algorithm to be

- *efficient* – the algorithm uses the least amount of resources necessary to accomplish its task.

Today we'll learn how to analyze the efficiency of an algorithm in the context of two fundamental problems in computation: searching and sorting. First we'll briefly touch on correctness using loop invariants.

The Sorting Problem

Sorting means rearranging the elements of an array (or *permuting* the array) so that the elements are in a well-defined order. For example, we can sort an array in non-decreasing order (ascending) so that each element $A[i] \leq A[i+1]$ for $i = 1 \ldots n-1$.

- Note that we don't define a separate output, only a result, or effect.

  In today's discussion we'll assume that our input is an array whose elements are integers. We've already seen how to generalize to arbitray user-defined types.

Insertion Sort

The insertion sort algorithm in pseudocode (from CLRS Chapter 2):

```
1 for j = 2 to A.length // A[1 .. A.length] is an array
2     key = A[j]
3     // Insert A[j] into the sorted sequence A[1 .. j - 1].
4     i = j - 1
5     while i > 0 and A[i] > key
6         A[i + 1] = A[i]
7         i = i - 1
8    A[i + 1] = key
```

Note the following conventions we use when describing algorithms abstractly:

- we use pseudocode instead of code in a particular programming language,
- the array indices start at 1 instead of 0, and
- the array has a property *length* so that we don't have to specify the array's size separately.

Loop Invariants

A loop invariant expresses a formal property of an algorithm that:

- is true prior to the first iteration of the loop,
- if it is true before an iteration of the loop remains true before the next iteration, and
- upon loop termination gives a useful property that helps show that the algorithm is correct.

A Loop Invariant for Insertion Sort

```
1 for j = 2 to A.length
2     key = A[j]
3     // Insert A[j] into the sorted sequence A[1 .. j - 1].
4     i = j - 1
5     while i > 0 and A[i] > key
6         A[i + 1] = A[i]
7         i = i - 1
8    A[i + 1] = key
```

At the start of each iteration of the for loop of lines 1-8, the subarray A[1 .. j - 1] consists of the elements originally in A[1 .. j - 1], but in sorted order.

Expresing Loop Invariants as `assert`ions

Insertion sort in Java (note translation to 0-based indexes):

```
for (int j = 1; j < a.length; ++j) {
    assert isSorted(a, 0, j - 1);
    int key = a[j];
    int i = j - 1;
    while(i >= 0 && a[i] > key) {
        a[i + 1] = a[i];
        i = i - 1;
    }
    a[i + 1] = key;
}
```

Note that we didn't express the entire invariant in Java. We could, but you must trade off implementation effort and benefit. Run the program with the `-ea` switch to enable assertions:

```
$ java -ea InsertionSort
```

See InsertionSort.java.

The Search Problem

The search problem is defined formally by the following input-output specifications:

- Input: A sequence of n elements $A =< a_1, a_2, ..., a_n >$ and a value v
- Output: An index i such that $v = A[i]$, or a special value such as -1 or *nil* if v does not appear in A.

We'll see that the assumptions we can make about the input affects the efficiency of the algorhtms we can use to search it.

Linear Search

If we can make no assumptions about the order of the array, our only option is linear search:

```
// A is an array, and v is the value we're searching for
LINEAR-SEARCH(A, v):
  for i = 1 to A.length
      if A[i] = v then
          return i
  return -1
```

We'll be dealing with search algorithms abstractly for the purpose of discussing running time analysis, but example implementations are available in Search.java.

Algorithmic Efficiency

We can characterize algorithmic efficiency in terms of space complexity (how much storage an algorithm requires) or time complexity (how "fast" an algorithm runs). Almost always primarily concerned with time complexity.

- Note that we want to eliminate platform-specific factors, like speed of the particular computer an algorithm runs on.

So we characterize algorithm performance in terms of

- input size, n, and
- order of growth as a function of input size.

An efficient algorithm beats an inefficient one even if the inefficient algortithm is run on a far superior computer.

Efficiency of Linear Search

Assuming each operation has a fixed cost, we can count the operations performed for a worst-case input as follows:

Step	Cost	Times
`for i = 1 to A.length`	c_1	n
`    if A[i] = v then`	c_2	n
`        return i`	c_3	0
`return -1`	c_4	1

Adding up the number of times each statement is executed we get:

$$T(n) = c_1 n + c_2 n + c_4 = (c_1 + c_2)n + c_4$$

We discard contant terms, constant factors, and lower-order terms to say that the worst case running time is

$$O(n)$$

And pronounce this “order n” or “Big-O of n.”

Binary Search

If the array is sorted you can use a binary search:

```
BINARY-SEARCH(A, v):
  p := 1, r := A.length
  while p ≤ r
      q := ⌊(p + r)/2⌋
      if A[q] = v then
          return q
      if A[q] > v then
          r := q - 1
      else
          p = q + 1
  return -1
```

Intuitively: We check the midpoint of the array (q).

- If the array is empty (p > r), the query value was not found.
- If the midpoint holds the value, return the midpoint.
- If the midpoint holds a value greater than our search value, repeat the process with the lower half of the array.
- If the midpoint holds a value less than our search value, repeat the process with the upper half of the array.

Efficiency of Binary Search

The key to analyzing the efficiency of BINARY-SEARCH is realizing that the array is halved in each iteration of the while loop.

- In the worst case BINARY-SEARCH runs until the size of the array $(r - p)$ goes from n to 1 by successive halving.
- This is equivalent to going from 1 to n by successive doubling.

Counting the number of times x we need to double to get from 1 to n is

$$2^x = n$$

so

$$x = \lg n$$

and the worst-case running time of BINARY-SEARCH is

$$O(\lg n)$$

Efficiency of Insertion Sort (Worst Case)

Step		Cost	Times
1:	`for j = 2 to A.length`	c_1	n
2:	`key = A[j]`	c_2	$n-1$
3:	// Insert A[j] into sorted A[i .. j - 1].		
4:	`i = j - 1`	c_4	$n-1$
5:	`while i > 0 and A[i] > key`	c_5	$\sum_{j=2}^{n} j$
6:	`A[i + 1] = A[i]`	c_6	$\sum_{j=2}^{n}(j-1)$
7:	`i = i - 1`	c_7	$\sum_{j=2}^{n}(j-1)$
8:	`A[i + 1] = key`	c_8	$n-1$

Noting the sum of the arithmetic series $\sum_1^n x = \frac{n(n-1)}{2} = \frac{1}{2}n(n+1)$, adding the running times, simplifying, discarding constant terms and factors, and lower-order terms we get:

$$O(n^2)$$

Run-time Analysis Summary: Linear

Sample pseudocode:

```
for i := 1 to n
    // ...
```

Java implementation:

```
for (int i = 0; i < n; i++) {
    // ...
}
```

Intuition:

- "For each of the n elements," or "for n times."

Big-O:

$$O(n)$$

Run-time Analysis Summary: Quadratic

Sample pseudocode:

```
for i := 1 to n
    for j := 1 to n
```

Java implementation:

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        // ...
    }
}
```

Intuition:

- "n times for each n"

Big-O:

$$O(n^2)$$

Run-time Analysis Summary: Also Quadratic

Sample pseudocode:

```
for i := 1 to n
    for j := 1 to i
```

Java implementation:

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < i; j++) {
        // ...
    }
}
```

Intuition:

- For each $i = 1..n$, count up to i
- This is sum of the the arithmetic series:
  $\sum_1^n x = \frac{n(n-1)}{2} = \frac{1}{2}n(n+1) = \frac{1}{2}n^2 + \frac{1}{2}n$

Big-O:

$$O(n^2)$$

Run-time Analysis Summary: Logarithmic (1 of 2)

Sample pseudocode:

```
p := 1, r := n
while p ≤ r
   // cut difference between p and r in half until it's 0...
```

Java implementation:

```
int lo = 0, hi = array.length - 1;
while (lo <= hi) {
    int middle = (lo + hi)/2;
    // either lo becomes middle + 1, or hi becomes middle - 1
}
```

Intuition:

- Each iteration of the loop cuts the remaining input in half. We stop when we've cut the input size to 1.
- Mathematically, we're multiplying the input size by $\frac{1}{2}$ each time.
- Run-time is the number of times we multiply by $\frac{1}{2}$

Mathematically ...

Run-time Analysis Summary: Logarithmic (2 of 2)

```
p := 1, r := n
while p ≤ r
    // cut difference between p and r in half until it's 0...
```

Run-time is the number of times x we multiply by $\frac{1}{2}$:

$$\frac{1}{2^x} n = 1$$
$$\frac{1}{2^x} = \frac{1}{n}$$
$$2^x = n$$
$$x = \log_2 n$$

So Big-O is:

$$O(\log n)$$

Closing Thoughts

- Algorithms and data structures are two sides of the same coin
- Data structures affect the kinds of algorithms that can operate on the data
- Operations on data structures are defined by algorithms
- Abstract data types package data structures with their associated algorithms