

An Actual Algorithm

PProblem:

< SORT: Given a finite n -tuple of integers, sort that tuple into increasing order

PAlgorithm:

- < 1. Scan the tuple left-to-right for an adjacent pair of elements that are out of order
- < 2. If you find such a pair, swap them and go back to step 1
- < 3. Your tuple is now sorted

PAnalysis:

- < It is not clear that this algorithm ever terminates
- < But we can prove (separately) that it does
 - in at most $n(n-1)/2$ swaps

Pseudo code for SORT

//Let a_1, a_2, \dots, a_n be the n -tuple to sort

```
found = true
while(found == true){
  found = false
  for i = 1 to n - 1
    if  $a_i > a_{i+1}$  then {
      swap  $a_i$  and  $a_{i+1}$ 
      found = true
      break (out of for loop)
    }
  }
}
//The  $n$ -tuple is now sorted
```

Another Algorithm

PProblem:

< MAX: Given a finite sequence a_1, a_2, \dots, a_n of real numbers, find the maximum element of that sequence

PAlgorithm:

- < 1. Let $max = a_1$
- < 2. Let $i = 2$
- < 3. If $a_i > max$, then let $max = a_i$
- < 4. Increment i
- < 5. If $i \neq n$, then go to step 3
- < 6. max is the maximum element of the sequence

Pseudo code for MAX

//Given a sequence a_1, a_2, \dots, a_n of real numbers

```
max =  $a_1$ ;
for i = 2 to n
  if  $a_i > max$  then  $max = a_i$ ;
//max is now the largest element
```

Another Algorithm

Problem:

< **SEARCH:** Given a sorted list of elements, determine if some given element s is in the list

Algorithm: (Binary Search)

- < 1. Let L be the original list
- < 2. If L has 1 or 2 elements, just check them directly. Otherwise, check the middle element of the list L
- < 3. If this element is s , then we are done
- < 4. Otherwise, compare this element with s . If s is larger, then let L be the portion of the list to the right of the middle element, otherwise let L be the portion of the list to the left of the middle element
- < 5. Go to step 2

Pseudocode for Binary Search

//Let L be a sorted list of n elements, search for s

```
start = 1; end = n;
while (start < end){
    middle = (start + end) / 2;
    if  $s > a_{middle}$  then start = middle + 1;
    else end = middle - 1;
}
if ( $s == a_{start}$ ) location = start;
else location = 0;

//location contains the subscript, or 0 if  $s$  is not present
```

Complexity of Algorithms

P “Complexity” refers to the amount of time required to carry out an algorithm

< (There are other types of complexity, for example, referring to the amount of space (memory) needed, but we won't consider those in this course)

P To compute the “time” of an algorithm, we assume that certain basic steps take unit time, and then compute the total running time by considering how many of our unit-time steps need to be done.

P As you'll see, we will often be very rough in our computations.

Complexity of Sort

1. Scan the tuple left-to-right for an adjacent pair of elements that are out of order	<pre>found = true while (found == true){ found = false for $i = 1$ to $n - 1$ if $a_i > a_{i+1}$ then { swap a_i and a_{i+1} found = true break (out of for loop) } }</pre>
2. If you find such a pair, swap them and go back to step 1	
3. Your tuple is now sorted	

This algorithm terminates after at most $n(n-1)/2$ swaps

Which description of the algorithm is easier to analyze?

Complexity of Sort

```

found = true
while(found == true){
    found = false ← unit time
    for i = 1 to n - 1
        if ai > ai+1 then {
            swap ai and ai+1
            found = true
            break (out of loop)
        }
    }
}

```

This loop loops at most $n - 1$ times

Takes a fixed amount of time (does not change over the course of the algorithm) so we consider that a "unit time" step

Complexity of Sort

- P Each run through the body of the **while** loop takes time at most n
- P The **while** loop runs at most $n(n - 1)/2$ times
- P For a total running time of # $n \times n(n - 1)/2$
- P Which we call $O(n^3)$

Complexity of Binary Search

```

start = 1; end = n;
while (start < end) {
    middle = (start + end) / 2;
    if s > amiddle then
        start = middle + 1;
    else end = middle - 1;
}
if (s == astart) location = start;
else location = 0;

```

Each pass through the body of the while loop takes constant (unit) time

Constant time

The question is: How many times will this while loop be executed?

Complexity of Binary Search

- P Calculate the number of times the while loop will be executed
- P Each pass through the while loop decreases the size of the range by at least half
- P So the question is:
 - < Given an integer n , how many times does it need to be cut in half before it reaches, or goes below, 1?
- P That is, which of the following will first be < 1 ?
 - < $n/2, n/4, n/8, n/16, \dots, n/2^k, \dots$

Complexity of Binary Search

PThat is, which of the following will first be < 1 ?
 $< n/2, n/4, n/8, n/16, \dots, n/2^k, \dots$

PWe solve the equation: $n/2^k < 1$, and get $k > \log_2 n$

PSo if we set $k = \lceil \log_2 n \rceil$, then we know that after that many iterations of the while loop, we will have found our item, or concluded that it was not in the list

Logarithmic Time Complexity is Fast

POur analysis shows that binary search can be done in time proportional to the \log of the number of items in the list

PThis is considered *very fast* when compared to linear or polynomial algorithms

PThe table to the right compares the number of operations that need to be performed for algorithms of various time complexities

n	log n	n	n ²	2 ⁿ
1	0	1	1	2
2	1	2	4	4
5	3	5	25	32
10	4	10	100	1024
20	5	20	400	1048576
50	6	50	2500	1.1E+15
100	7	100	10000	1.3E+30
200	8	200	40000	1.6E+60
500	9	500	250000	err
1000	10	1000	1E+06	err
2000	11	2000	4E+06	err
5000	13	5000	3E+07	err