

ISSUES IN ANALYZING ALGORITHMS

Many Ways to Skin a Cat

```
SQ1(x)
  return x * x
```

```
SQ2(x)
  ans = 0
  {Add x to ans x times}
  for i = 1 to x do
    do ans = ans + x
  return ans
```

```
SQ3(x)
  ans = 0
  {Add 1 to ans x^2 times}
  for i = 1 to x
    do for j = 1 to x
      do ans = ans + 1
  return ans
```


Choosing a Barometer

What should we count to measure time?

- Number of arithmetic operations (+, *, <, etc.)?
- Number of assignments (=) performed?
- Number of times a line is executed?

Input	SQ1	SQ2	SQ3
1			
2			
3			
n			

Details:

- Some operations may be more expensive than others!
- Do we count "hidden" increments, tests, and assignments in `for` loops?
- Must pick representative line(s), usually bodies of inner loops.

Model of Computation

Running times depend on model of computation!

Sequential vs. Parallel (see Takis's Parallel Algorithms Course)

Typically assume that numerical operations take constant time.

- Addition would take linear time if model only supported increment operations.
- In practice, operations take time proportional to number of bits ($\lg n$).

Measuring Input Size

Standard assumptions:

- Size of numerical input is the input itself
- Size of array input is length of array

Not always obvious:

- Size of tree may be number of nodes or height.
- Size of number n may be n or number of bits ($\lg n$).

Can have more than one size:

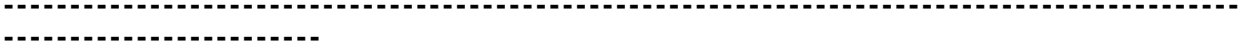
- Searching for string of length m in text of length n .
- Processing a graph with V vertices and E edges.

Running Time for a Particular Input Size

Running time may differ greatly for different inputs of the same size. How to characterize?

- **Worst-case analysis:** consider maximum time for every input size.
- **Best-case analysis:** consider minimum time for every input size (not a good idea!).
- **Average-case analysis:** ²expected running time based on probability

distribution of inputs (can be difficult!).



Example: Insertion Sort

Example of good ol' **divide-and-conquer**:

- **divide** a problem into subproblems
- **conquer** the subproblems by solving them recursively
- **combine** the solutions of the subproblems to form the solution of the whole problem

```
InsertionSort(A, k)
  Sort A[1..k] via insertion sort method
  Initially call Merge-Sort(A,1,length[A])
  if n > 0 A[1..0] = empty array is trivially sorted; do nothing
    then InsertionSort(A, k - 1)
      Insert(A, k)
```

```
Insert(A, i)
  Assume A[1..i-1] is sorted. Make A[1..i] sorted
  if i > 1 A[1..1] is trivially sorted; do nothing
    then if LessThan(A, i - 1, i) Is A[i-1] < A[i]?
      then Swap(A, i - 1, i) Swap contents of A[i-1] and A[i]
        Insert(A, i - 1)
```


Analysis of Insert

Worst-Case

i	#Calls to Insert	#LessThans	#Swaps
1			
2			
3			
<i>n</i>			

Best-Case

i	#Calls to Insert	#LessThans	#Swaps
1			
2			
3			
<i>n</i>			

Average-Case

i	#Calls to Insert	#LessThans	#Swaps
1			
2			
3			

n			
-----	--	--	--

Analysis of InsertionSort

Worst-Case

k	#Calls to Insertion-Sort	#Calls to Insert	#LessThans	#Swaps
1				
2				
3				
n				

Best-Case

k	#Calls to Insertion-Sort	#Calls to Insert	#LessThans	#Swaps
1				
2				
3				
n				

AverageCase

k	#Calls to Insertion-Sort	#Calls to Insert	#LessThans	#Swaps
1				
2				
3				
n				

Another Version of Insertion Sort

CLR version of insertion sort:

- no procedure call overhead.
- iterative algorithm.
- **invariant**: $A[1..j]$ is in sorted order after every iteration of **for** loop.
- see CLR for detailed analysis.

Insertion-Sort(A)

```

for j  2 to length[A]
  do key  A[j]
      Insert A[j] into the sorted sequence A[1..j-1].
      i  j-1
      while i > 0 and A[i] > key
        do A[i + 1] = A[i]

```

i i - 1
A[i + 1] key



Merge Sort

Idea: Divide array into two equal-sized subproblems, recursively sort, then merge results.

```
Merge-Sort(A, p, r)
  Sort A[p..r] by insertion sort method.
  Initially call Merge-Sort(A,1,length[A]).
  if p < r
    then q = (p + r) / 2
         Merge-Sort(A, p, q)
         Merge-Sort(A, q + 1, r)
         Merge(A, p, q, r)
```

Merge left as an exercise.

The Next Three Lectures

Asymptotic Notation: coarse-grained comparisons of algorithms.

Recurrences: coarse-grained analysis of algorithms.

Probability: tools for average-case analysis.

Pop Quiz

What's is the main topic of this course?

1. Quotes of the Vice President.
2. Patterns of growth in water flora.
3. Recipes and resources.
4. Habits of accompanying male friends.