

Issues In Algorithm Analysis

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
    ans ← ans + x
  return ans
```

```
SQ3(x)
  ans ← 0
  ▷ Add 1 to ans x2 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 of code is executed?

E.g., suppose we count arithmetic operations:

x	SQ1	SQ2	SQ3
1			
2			
3			
⋮	⋮	⋮	⋮
n			

Details:

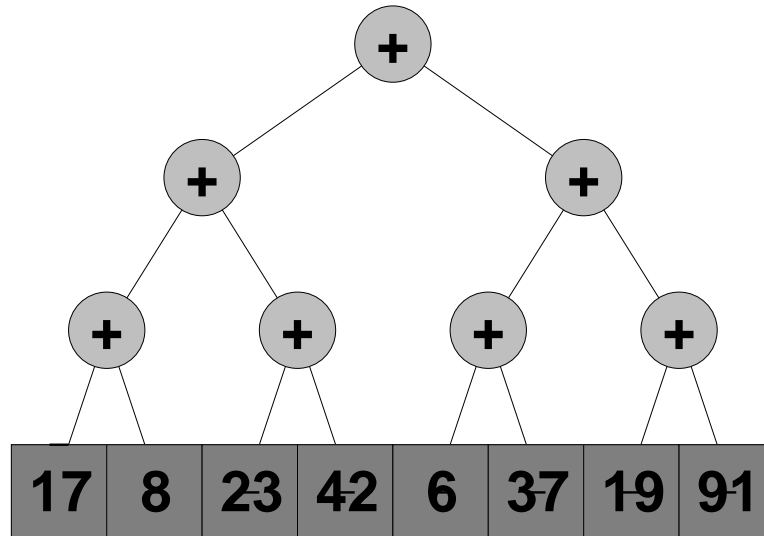
1. Some operations may be more expensive than others!
2. Do we count "hidden" increments, tests, and assignments in **for** loops?

3. Must pick representative line(s), usually bodies of inner loops.

Model of Computation

Running times depend on model of computation!

- 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 ($\lg n$).
- We will generally assume sequential rather than parallel model. (See CS331 for Parallel algorithms.)



Measuring Input Size

Standard assumptions:

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

Not always obvious:

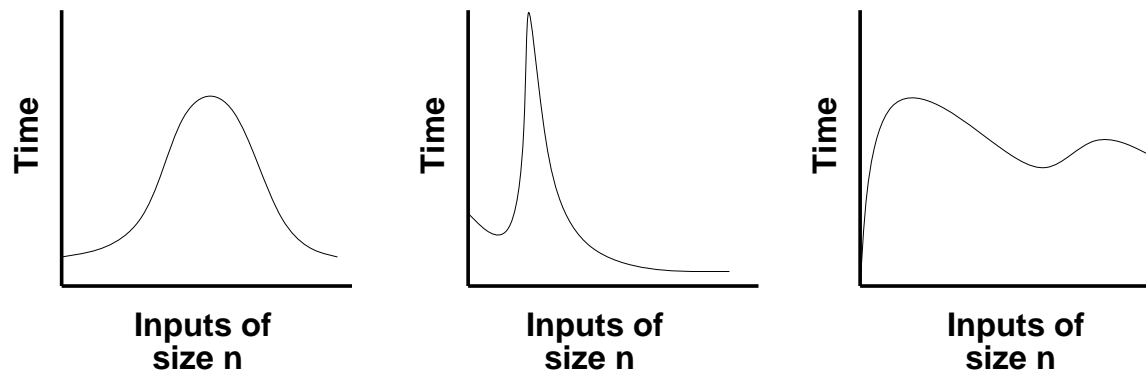
- Size of tree may be number of nodes or height.
- Size of number n may be n or number of bits ($\lg n$).

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

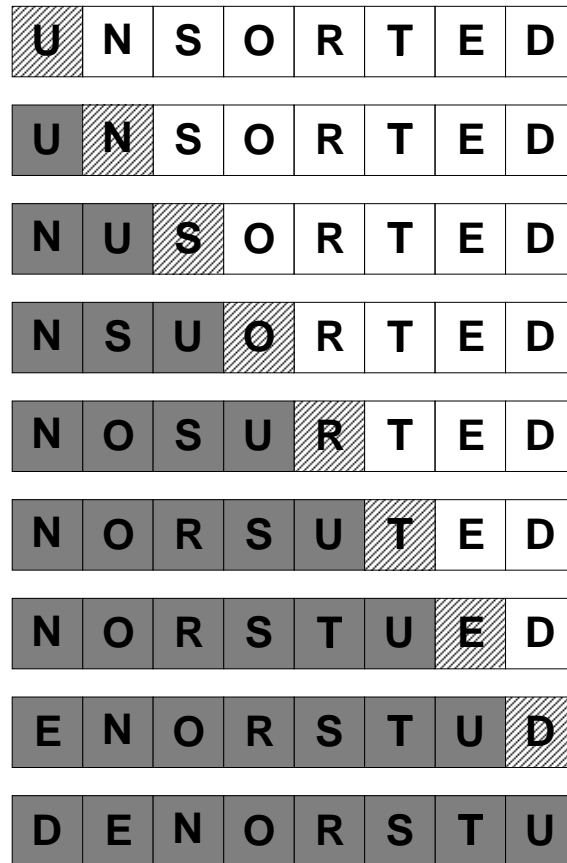
Running time may differ greatly for different inputs of the same size.



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:** expected running time based on probability distribution of inputs (can be difficult!).

Example: Insertion Sort



Example of good old **divide-conquer-and-glue**:

- **divide** a problem into subproblems
- **conquer** the subproblems by solving them recursively
- **glue** 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 `InsertionSort(A, length[A])`*

if $n > 0$ **then** ▷ *$A[1..0]$ = empty array is trivially sorted; do nothing*

`InsertionSort(A, k - 1)`

`Insert(A, k)`

`Insert(A, i)`

▷ *Assume $A[1..i-1]$ is sorted. Make $A[1..i]$ sorted*

if $i > 1$ **then** ▷ *$A[1..1]$ is trivially sorted; do nothing*

if $A[i-1] < A[i]$ **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>i</i>	# calls to Insert	# <	# Swaps
1			
2			
3			
⋮	⋮	⋮	⋮
<i>n</i>			

Best-case =

<i>i</i>	# calls to Insert	# <	# Swaps
1			
2			
3			
⋮	⋮	⋮	⋮
<i>n</i>			

Average-case =

<i>i</i>	# calls to Insert	# <	# Swaps
1			
2			
3			
⋮	⋮	⋮	⋮
<i>n</i>			

Analysis of InsertionSort

Worst-case =

k	# Insert	# InsertionSort	# <	# Swaps
1				
2				
3				
\vdots	\vdots	\vdots	\vdots	\vdots
n				

Best-case =

k	# Insert	# InsertionSort	# <	# Swaps
1				
2				
3				
\vdots	\vdots	\vdots	\vdots	\vdots
n				

Average-case =

k	# Insert	# InsertionSort	# <	# Swaps
1				
2				
3				
\vdots	\vdots	\vdots	\vdots	\vdots
n				

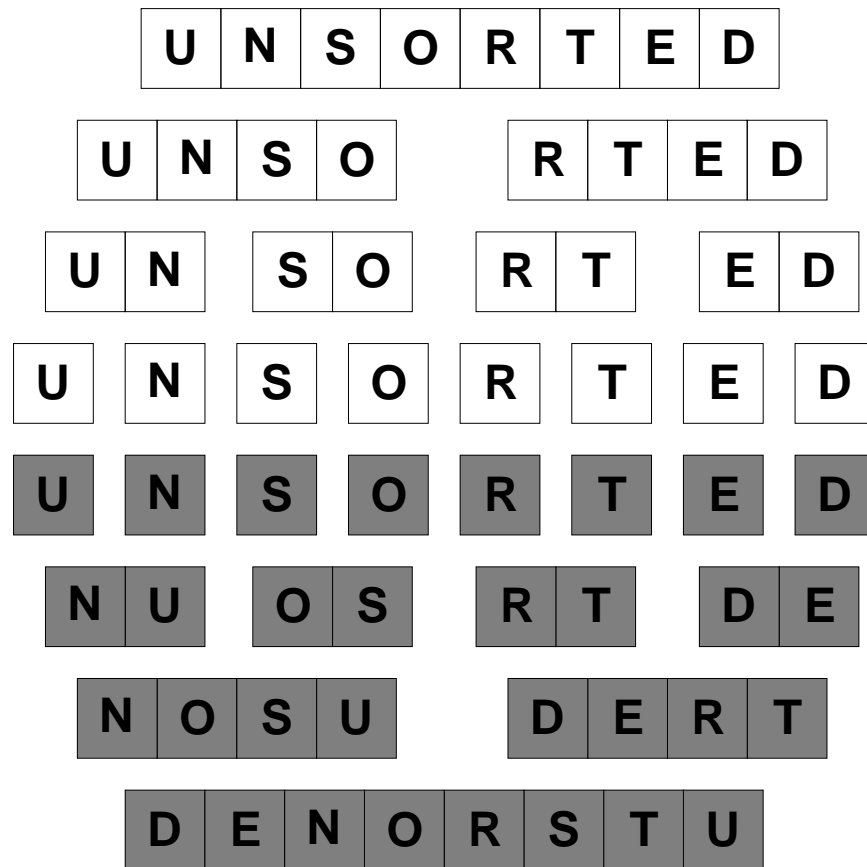
Another Version of Insertion Sort

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.

```
InsertionSort(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.

MergeSort(A, p, r)

▷ Sort $A[p..r]$ by insertion sort method.

▷ Initially call $\text{MergeSort}(A, 1, \text{length}[A])$.

if $p < r$ **then**

$q \leftarrow \lfloor (p + r) / 2 \rfloor$

MergeSort(A, p, q)

MergeSort(A, q + 1, r)

Merge(A, p, q, r)

Merge left as an exercise. Can be done in linear ($\Theta(n)$) time.

Model running time as the solution to

$$T(n) = 2T(n/2) + \Theta(n), n \geq 1$$

$$T(n) = 0, n < 1$$

The Next Four Lectures

1. **Asymptotic Notation:** coarse-grained comparisons of algorithms.
2. **Recurrences:** coarse-grained analysis of algorithms.
3. **Counting and Probability:** Tools for measuring average cases (also for algorithms that use randomness).
4. **Probabilistic Analysis:** Using probability in analysis.

Pop Quiz

What is the main topic of this course?

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