CSCI 241

Lecture 3:
Insertion and Selection Sort
Intro to Runtime Analysis
Recursion
Announcements

• First programming assignment out Sunday.
  • We’ll cover all the sorting algorithms you need by next Wednesday.

• Lab 2 also out Sunday
  • Done in the same repository as A1 - writing test code

• Norms
Quiz 0

• Quiz 0 is today. Covers only review material.
  • Will be scored but grading is based only on completion.
  • Taken on gradescope.com between 10am to 10pm today.
  • 15 minute time limit
• This is your trial run: make sure you can login and take the quiz, etc. Later quizzes will count towards your grade.
Goals

• Be able to execute insertion sort and selection sort on paper.

• Be able to implement insertion sort and selection sort.

• Know how to count primitive operations to determine the runtime of an algorithm.

• Understand how recursive methods are executed.
Insertion Sort
Insert $A[i]$ into the sorted sublist $A[0..i-1]$.

Selection Sort
Find the smallest element in $A[i..n]$ and place it at $A[i]$.

https://visualgo.net/bn/sorting
**Insertion Sort**

Insert A[i] into the sorted sublist A[0..i-1].

**Invariant:**

\[ A_{\text{sorted}} \]

**Selection Sort**

Find the smallest element in A[i..n] and place it at A[i].

**Invariant:**

\[ A_{\text{sorted, } \leq A[i..n]} \]

https://visualgo.net/bn/sorting
insertionSort(A):
  i = 0;
  while i < A.length:
    // push A[i] to its sorted position by repeatedly
    //   swapping with the element to its left
    // increment i
    i

Invariant: A \text{ sorted} \leq A[i..n]

selectionSort(A):
  i = 0;
  while i < A.length:
    // find min of A[i..A.length]
    // swap it with A[i]
    // increment i
    i

Invariant: A \text{ sorted, } \leq A[i..n]
// Sorts A using insertion sort
insertionSort(A):
    i = 0;
    while i < A.length:
        j = i;
        while j > 0 and A[j] > A[j-1]:
            swap(A[j], A[j-1])
            j--
        i++

Invariant: A sorted
Insertion Sort: Exercise

// Sorts A using insertion sort
insertionSort(A):
    i = 0;
    while i < A.length:
        j = i;
        while j > 0 and A[j] < A[j-1]:
            swap(A[j], A[j-1])
            j--
        i++

Sort the following array using **insertion sort**:

\[1 4 8 2 6\]

How many times did you swap two elements?

A. 3  
B. 4  
C. 6  
D. 8
// Sorts A using insertion sort
insertionSort(A):
    i = 0;
    while i < A.length:
        j = i;
        while j > 0 and A[j] < A[j-1]:
            swap(A[j], A[j-1])
            j--
        i++
selectionSort(A):
    i = 0;
    while i < A.length:
        // find min of A[i..A.length]
        // swap it with A[i]
        // increment i

Sort the following array using selection sort:
[1 4 8 2 6]

How many times did you swap two distinct elements?

A. 2
B. 3
C. 4
D. 5

Invariant: A sorted, <= A[i..n]
selectionSort(A):

```python
i = 0;
while i < A.length:
    // find min of A[i..A.length]
    // swap it with A[i]
    // increment i
```

1 2 4 8 6

1 2 8 4 6 (1)

1 2 4 8 6 (2)

1 2 4 6 8 (3)
Practice Problems

1. Write code for Selection Sort

2. Consider the array:

   [ 8 4 6 10 7 1 2 ]

   Write the state of the array at the conclusion of the loop iteration in which $i == 4$ (don’t forget arrays are 0-indexed!).

   InsertionSort:

   SelectionSort:
Which sort should we use?
Which sort should we use?

- Which one takes less time?
Which sort should we use?

• Which one takes less time?

• Which one takes less memory?
Which sort should we use?

- Which one takes less time?

- Which one takes less memory?

- Other considerations?
How do we measure these things?
How do we measure these things?
How do we measure these things?

- Which one takes less time?
How do we measure these things?

• Which one takes less time?

• Which one takes less memory?
How do we measure these things?

- Which one takes less time?
- Which one takes less memory?
- Other considerations?
Measuring Runtime

Question: How could we measure how "fast" an algorithm runs?

```java
public int findMax(int[] a) {
    int currentMax = a[0];
    for (int i = 1; i < a.length; i++) {
        if (currentMax < a[i]) {
            currentMax = a[i];
        }
    }
    return currentMax;
}
```
How should we measure runtime?

How about metrics that are \textit{invariant} to:

\begin{itemize}
  \item Length of the array $a$?
  \item How fast your computer is?
\end{itemize}
How should we measure runtime?

How about metrics that are **invariant** to:

- Length of the array a?
- How fast your computer is?

Approach: count the number of “operations” the computer needs to execute.

- Count it *in terms of* the input size
- “operations” may be faster or slower depending on the hardware
“Primitive” Operations

Things the computer can do in a “fixed” amount of time.

“fixed” - doesn’t depend on the input size \( (n) \)

A non-exhaustive list:

- **Get** or **set** the value of a variable or array location
- **Evaluate** a simple expression
- **Return** from a method
Strategies for counting primitive operations

Easiest case:

1. Identify all primitive operations
2. Identify how many time each one happens
3. Add them all up.

alg(A, n):

1. sum = 0
2. for i = 1..n:
   - sum += A[i]
Strategies for counting primitive operations

Easiest case:

1. Identify all primitive operations
2. Identify how many time each one happens
3. Add them all up.

\[
\text{alg}(A, n):
\]

\[
\text{sum} = 0
\]

\[
\text{for } i = 1..n:\n\]

\[
\text{sum} += A[i]
\]