INTRO TO SORTING

cs2420 | Introduction to Algorithms and Data Structures | Fall 2016
administrivia...
-assignment 3 is due Wednesday at midnight

-reminders
  - you must work in pairs on the code
  - analysis documents must be completed on your own
last time...
algorithm analysis
-correctness is only half the battle

-programs are expected to terminate in a reasonable amount of time

-running time of a program is strongly correlated to the choice of algorithms used in problem solving

-how much time and space does an algorithm require?
finding a word in a dictionary

algorithm 2 (algorithm-ified):

1) go to the middle word

2) is this our word?
   - if so, done

3) if not
   - if word of interest is greater, go to middle word in second half
   - else, go to middle word in first half

4) go back to step 2
finding a word in a dictionary

algorithm 2 (algorithm-ified):

1) go to the middle word

2) is this our word?
   - if so, done

3) if not
   - if word of interest is greater, go to middle word in second half
   - else, go to middle word in first half

4) go back to step 2

what is the big-o complexity of this algorithm?

1) $O(1)$
2) $O(\log N)$
3) $O(N)$
4) $O(N \log N)$
5) $O(N^2)$
6) $O(N^3)$
how to get log growth?

- starting at $x=1$, how many iterations of $x*2$ before $x>=N$?
  - the repeated doubling principle

- starting at $x=N$, how many iterations of $x/2$ before $x<=1$?
  - the repeated halving principle
Typical run-time complexities:

- $N^2$ and $N^3$ are typically not acceptable for moderate input sizes!
finding the maximum value in an array

```java
int max = array[0];
for (int i=0; i<n; i++)
{
    if (array[i] > max )
        max = array[i];
}
```

we want to count the number of instructions
- break code into simple instructions
- things that the CPU can directly execute (roughly)

assume the following are one instruction each:
- assigning a value to a variable
- looking up the value of a particular array element
- comparing two values
- incrementing a value
- basic arithmetic ops such as addition and multiplication

assume branching is instantaneous

# instructions = f(n) = 6n + 4 (in the worst case)
we now have a (rough) idea of how fast the algorithm is

in reality, though, the number of actual instructions varies with:

- compiler
- programming language
- available CPU instruction set
- ...

computer scientists ignore these pesky details when reasoning about algorithm complexity
f(n) = 6n + 4

- furthermore, we ignore the terms that grow slowly and just keep those that grow fast as n becomes larger
  - 4 will always be 4 regardless of n
  - but, 6n will increase as n gets bigger

f(n) = 6n

- we’ll also ignore the constant in front of a growth term
  - each of these instructions may be slightly different depending on machine specific factors

f(n) = n

- asymptotic behavior: described when we drop all factors and keep just the largest growing term
  - interested in the limit of f(n) as n tends to infinity

- this asymptotic behavior is what is known as the complexity of an algorithm
-big-O notation (\(O\)) is used to capture the run-time behavior of an algorithm
  -assuming large \(N!\) (ie. asymptotic behavior)

-for example, the running time of a quadratic algorithm is \(N^2\) is specified \(O(N^2)\)
  -read “order \(N\) squared”

-this notation allows us to establish a relative order among algorithms
  -\(O(N \log N)\) is better than \(O(N^2)\)
worst, average, best

-worst-case is a guarantee on all inputs — it will never be worse than this

-average-case is the common case, measured over all possible inputs
  -this is the most useful!

-best-case is the absolute fastest that an algorithm can terminate
  -we don’t care about this because it rarely happens
examples...
analyze the running time

for(int i=0; i<n; i+=2)
    sum++;

1) O(1)
2) O(log N)
3) O(N)
4) O(N log N)
5) O(N^2)
6) O(N^3)
analyze the running time

```cpp
for(int i=0; i<n; i+=2)
    sum++;

for(int i=0; i<n; i++)
    for(int j=0; j<n*n; j++)
        sum++
```

1) \(O(1)\)
2) \(O(\log N)\)
3) \(O(N)\)
4) \(O(N \log N)\)
5) \(O(N^2)\)
6) \(O(N^3)\)
analyze the running time

for (int i = 0; i < n; i += 2) 
    sum++; 

for (int i = 0; i < n; i++) 
    for (int j = 0; j < n * n; j++) 
        sum++; 

for (int i = 0; i < n; i *= 2) 
    sum++; 

1) O(1) 
2) O(log N) 
3) O(N) 
4) O(N log N) 
5) O(N^2) 
6) O(N^3)
finding the smallest difference algorithm?
finding the smallest difference algorithm?

diff = MAX_INTEGER;
for (int i=0; i<array.length-1; i++)
{
    num1 = array[i];
    for (int j=i+1; j<array.length; j++)
    {
        num2 = array[j];
        if (abs(num1-num2) < diff)
            diff = abs(num1-num2);
    }
}
return diff;
finding the smallest difference algorithm?

diff = MAX_INTEGER;
for(int i=0; i<array.length-1; i++)
{
    num1 = array[i];
    for(int j=i+1; j<array.length; j++)
    {
        num2 = array[j];
        if (abs(num1-num2) < diff)
            diff = abs(num1-num2);
    }
}
return diff;

what is the big-o complexity of this algorithm?

1) O(1)
2) O(log N)
3) O(N)
4) O(N log N)
5) O(N^2)
6) O(N^3)
add

```java
int[] data = new int[6];
data.add(5);
data.add(17);
data.add(9);
size = 0
```
int[] data = new int[6];
data.add(5);
data.add(17);
data.add(9);
add

```java
int[] data = new int[6];
data.add(5);
data.add(17);
data.add(9);
```

5 17 size = 2
add

int[] data = new int[6];
data.add(5);
data.add(17);
data.add(9);

size = 3
add

int[] data = new int[6];
data.add(5);
data.add(17);
data.add(9);

what is the complexity of add?

1) O(1)
2) O(log N)
3) O(N)
4) O(N log N)
5) O(N²)
6) O(N³)
grow

data → 5 17 9 12 1 33
grow

data → 5 17 9 12 1 33

tmp = new int[data.length*2];

tmp →
grow

data → 5 17 9 12 1 33

tmp = new int[data.length*2];

tmp →

copy all from data to tmp

tmp → 5 17 9 12 1 33
```java
int data[] = {5, 17, 9, 12, 1, 33};

int[] tmp = new int[data.length * 2];

copy all from data to tmp

data = tmp;
```
grow

data → 5 17 9 12 1 33

tmp = new int[data.length*2];

tmp →

copy all from data to tmp

tmp → 5 17 9 12 1 33
data = tmp;

data → 5 17 9 12 1 33

what is the complexity of growing?

1) O(1)
2) O(log N)
3) O(N)
4) O(N log N)
5) O(N^2)
6) O(N^3)
remove

\[
\begin{array}{cccccc}
5 & 17 & 9 & 12 & 1 & 33 \\
\end{array}
\]

\[\text{size} = 6\]
remove

5 17 9 12 1 33 size = 6

data.remove(9);
remove

data.remove(9);

size = 6

5 17 9 12 1 33

size = 5

5 17 12 1 33
remove

5 17 9 12 1 33 \hspace{1cm} \text{size} = 6

data.remove(9);

5 17 12 1 33 \hspace{1cm} \text{size} = 5

5 17 12 1 33 \hspace{1cm} \text{size} = 5
remove

5 17 9 12 1 33

size = 6

data.remove(9);

5 17 12 1 33

size = 5

5 17 12 1 33

size = 5

1) O(1)
2) O(log N)
3) O(N)
4) O(N log N)
5) O(N^2)
6) O(N^3)

what is the complexity of remove?
today...
- why sort?
- selection sort
- insertion sort
why sort?
- sorting is a fundamental application in computing
  - one of the most intensively studied and important operations

- most data is useless unless it is in some kind of order

- for any given problem, or specific goal isn’t necessarily sorting... but we often need to sort to efficiently solve problems
  - computer graphics
  - look-up tables
  - games
- sorting algorithms that are easy to understand (and implement) run in **quadratic time**

- more complicated algorithms cut it to $O(N \log N)$
  - implementation details are critical to attaining this bound!

- for very specific types of data we can actually do better
  - but we won’t study these algorithms in this class
without thinking too hard, how can we sort any array of items?
ie. don't try to be tricky and come up with the most complicated thing possible. what is the easiest (but algorithmic) thing you can think of? pretend you have to write the pseudo-code in under 3 minutes...

without thinking too hard, how can we sort any array of items?
selection sort
the simplest sorting algorithm
Find (i.e. select) the smallest item in the unsorted portion of the array and move to the end of the sorted portion of the array.
selection sort

1) find the minimum item in the unsorted part of the array
2) swap it with the first item in the unsorted part of the array
3) repeat steps 1 and 2 to sort the remainder of the array
selection sort

1) find the minimum item in the unsorted part of the array
2) swap it with the first item in the unsorted part of the array
3) repeat steps 1 and 2 to sort the remainder of the array

what does this look like?
```java
void selectionSort(int[] arr)
{
    for(int i=0; i < arr.length-1; i++)
    {
        min = i;
        for(int j=i+1; j < arr.length; j++)
            if (arr[j] < arr[min])
                min = j;

        temp = arr[i];
        arr[i] = arr[min];
        arr[min] = temp;
    }
}
```
void selectionSort(int[] arr) {
    for (int i = 0; i < arr.length - 1; i++) {
        min = i; // last item in sorted part of array
        for (int j = i + 1; j < arr.length; j++)
            if (arr[j] < arr[min])
                min = j;
        temp = arr[i];
        arr[i] = arr[min];
        arr[min] = temp;
    }
}
void selectionSort(int[] arr)
{
    for(int i=0; i < arr.length-1; i++)
    {
        min = i; \textcolor{red}{\textbf{last item in sorted part of array}}
        for(int j=i+1; j < arr.length; j++)
            if (arr[j] < arr[min])
                min = j;

        temp = arr[i];
        arr[i] = arr[min];
        arr[min] = temp;
    }
}
```java
void selectionSort(int[] arr) {
    for (int i = 0; i < arr.length - 1; i++) {
        min = i;  // last item in sorted part of array
        for (int j = i + 1; j < arr.length; j++)
            if (arr[j] < arr[min])
                min = j;

        temp = arr[i];
        arr[i] = arr[min];
        arr[min] = temp;  // swap items
    }
}
```
Let's talk about complexity.

```java
void selectionSort(int[] arr) {
    for (int i = 0; i < arr.length - 1; i++) {
        min = i; // last item in sorted part of array
        for (int j = i + 1; j < arr.length; j++)
            if (arr[j] < arr[min])
                min = j;

        temp = arr[i];
        arr[i] = arr[min];
        arr[min] = temp;
    }
}
```
void selectionSort(int[] arr)
{
    for(int i=0; i < arr.length-1; i++)
    {
        min = i;
        for(int j=i+1; j < arr.length; j++)
            if (arr[j] < arr[min])
                min = j;

        temp = arr[i];
        arr[i] = arr[min];
        arr[min] = temp;
    }
}

let's talk about complexity.
void selectionSort(int[] arr)
{
    for(int i=0; i < arr.length-1; i++)
    {
        min = i;
        for(int j=i+1; j < arr.length; j++)
            if (arr[j] < arr[min])
                min = j;
        temp = arr[i];
        arr[i] = arr[min];
        arr[min] = temp;
    }
}

Let's talk about complexity.
```java
void selectionSort(int[] arr) {
    for (int i = 0; i < arr.length - 1; i++) {
        min = i;
        for (int j = i + 1; j < arr.length; j++)
            if (arr[j] < arr[min])
                min = j;

        temp = arr[i];
        arr[i] = arr[min];
        arr[min] = temp;
    }
}
```

**what is the relationship of the best and worst case complexity?**
insertion sort
good for small $N$
Take the first item in the unsorted portion of the array and *insert* it into the sorted portion of the array.
insertion sort

1) the first array item in the unsorted array is the sorted portion of the array

2) take the second item and insert it in the sorted portion

3) repeat steps 1 and 2 to sort the remainder of the array
insertion sort

1) the first array item in the unsorted array is the sorted portion of the array

2) take the second item and insert it in the sorted portion

3) repeat steps 1 and 2 to sort the remainder of the array

what does this look like?
void insertionSort(int[] arr)
{
    for(int i=1; i < arr.length; i++)
    {
        index = arr[i];
        j = i;
        while(j>0 && arr[j-1]>index)
        {
            arr[j] = arr[j-1];
            j--;
        }
        arr[j] = index;
    }
}
void insertionSort(int[] arr)
{
    for(int i=1; i < arr.length; i++)
    {
        index = arr[i];  // item to be inserted
        j = i;
        while(j>0 && arr[j-1]>index)
        {
            arr[j] = arr[j-1];
            j--;
        }
        arr[j] = index;
    }
}
void insertionSort(int[] arr)
{
    for (int i = 1; i < arr.length; i++)
    {
        int index = arr[i];
        int j = i;
        while (j > 0 && arr[j - 1] > index)
        {
            arr[j] = arr[j - 1];
            j--;
        }
        arr[j] = index;
    }
}
void insertionSort(int[] arr)
{
    for(int i=1; i < arr.length; i++)
    {
        index = arr[i];  // item to be inserted
        j = i;
        while(j>0 && arr[j-1]>index)
        {
            arr[j] = arr[j-1];
            j--;
        }
        arr[j] = index;  // insert item
    }
}
void insertionSort(int[] arr)
{
    for(int i=1; i < arr.length; i++)
    {
        index = arr[i];
        j = i;
        while(j>0 && arr[j-1]>index)
        {
            arr[j] = arr[j-1];
            j--;
        }
        arr[j] = index;
    }
}

what is the best-case complexity?
1) O(1)
2) O(log N)
3) O(N)
4) O(N log N)
5) O(N²)
6) O(N³)
void insertionSort(int[] arr) {
    for(int i=1; i < arr.length; i++) {
        index = arr[i];
        j = i;
        while(j>0 && arr[j-1]>index) {
            arr[j] = arr[j-1];
            j--;
        }
        arr[j] = index;
    }
}

Determining average & worst-case requires a measure of unsortedness.
unsortedness

-inversion: a pair of array items that are out of order

| 45 | -3 | 9 | 76 | 11 | -8 | 0 |
unsortedness

-inversion: a pair of array items that are out of order

\[
45 \quad -3 \quad 9 \quad 76 \quad 11 \quad -8 \quad 0
\]

*how many inversions are there?*
unsortedness

-inversion: a pair of array items that are out of order

| 45 | -3 | 9 | 76 | 11 | -8 | 0 |

how many inversions are there?

-sorting efficiency depends on how many inversions are removed per step
insertion sort complexity

each swap to the left removes one inversion…

…we must visit each item at least once \(N\)…

…and we must undo \(I\) inversions

| 45 | -3 | 9 | 76 | 11 | -8 | 0 |

*swap removes one inversion*
insertion sort complexity

each swap to the left removes one inversion…

…we must visit each item at least once ($N$)…

…and we must undo $I$ inversions

\[
\begin{array}{ccccccc}
45 & -3 & 9 & 76 & 11 & -8 & 0 \\
\end{array}
\]

swap removes one inversion

insertion sort is $O(N+I)$
insertion sort complexity

each swap to the left removes one inversion…

…we must visit each item at least once ($N$)…

…and we must undo $I$ inversions

\[
\begin{array}{cccccccc}
45 & -3 & 9 & 76 & 11 & -8 & 0 \\
\end{array}
\]

insertion sort is $O(N+I)$

How do we figure out what $I$ is?
worst case scenario...

- what are the number of inversions in the worst case?
  - what *IS* the worst case?
worst case scenario...

- what are the number of inversions in the worst case?
  - what *IS* the worst case?
  - when every unique pair is inverted…

| 76 | 45 | 11 | 9  | 0  | -3 | -8 |

--- inverted array
worst case scenario...

- what are the number of inversions in the worst case?
  - what is the worst case?
  - when every unique pair is inverted…

- how many unique pairs are there?

| 76 | 45 | 11 | 9  | 0  | -3 | -8 |

inverted array
worst case scenario...

-what are the number of inversions in the worst case?

-what *IS* the worst case?

-when every **unique pair** is inverted…

```
76 45 11 9 0 -3 -8
```

-inverted array

-how many unique pairs are there?

-(hint: remember Gauss’ trick!)
worst case scenario...

-what are the number of inversions in the worst case?
  -what \textit{IS} the worst case?
  -when every \textit{unique pair} is inverted…

\begin{itemize}
  \item \textbf{inverted array}
    \begin{tabular}{c|c|c|c|c|c|c|c|c}
      76 & 45 & 11 & 9 & 0 & -3 & -8
    \end{tabular}
\end{itemize}

-\textbf{how many unique pairs are there?}
  -(hint: remember Gauss’ trick!)

\[(N+1) \times \frac{N}{2} = \frac{N^2 + N}{2}\]
insertion sort is $O(N+I)$

What is the worst-case complexity of insertion sort?
1) $O(1)$
2) $O(\log N)$
3) $O(N)$
4) $O(N \log N)$
5) $O(N^2)$
6) $O(N^3)$
What is the best-case complexity of insertion sort?

1) $O(1)$
2) $O(\log N)$
3) $O(N)$
4) $O(N \log N)$
5) $O(N^2)$
6) $O(N^3)$

Insertion sort is $O(N+I)$
average case scenario...
average case scenario...

- assume that there is a 50% chance that any given pair is inverted

- average number of inversions = (number of pairs) / 2
average case scenario...

- assume that there is a 50% chance that any given pair is inverted

- average number of inversions = (number of pairs) / 2

\[
\frac{\left( \frac{N^2 + N}{2} \right)}{2} = \frac{N^2 + N}{4}
\]

number of pairs
What is the average-case complexity of insertion sort?

1) $O(1)$
2) $O(\log N)$
3) $O(N)$
4) $O(N \log N)$
5) $O(N^2)$
6) $O(N^3)$

Insertion sort is $O(N + I)$.
recap...
## Selection vs Insertion

<table>
<thead>
<tr>
<th></th>
<th>Selection</th>
<th>Insertion</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>worst:</strong></td>
<td>$O(N^2)$</td>
<td>$O(N^2)$</td>
</tr>
<tr>
<td><strong>average:</strong></td>
<td>$O(N^2)$</td>
<td>$O(N^2)$</td>
</tr>
<tr>
<td><strong>best:</strong></td>
<td>$O(N^2)$</td>
<td>$O(N)$</td>
</tr>
</tbody>
</table>
selection vs insertion

<table>
<thead>
<tr>
<th></th>
<th>selection</th>
<th>insertion</th>
</tr>
</thead>
<tbody>
<tr>
<td>worst</td>
<td>$O(N^2)$</td>
<td>$O(N^2)$</td>
</tr>
<tr>
<td>average</td>
<td>$O(N^2)$</td>
<td>$O(N^2)$</td>
</tr>
<tr>
<td>best</td>
<td>$O(N^2)$</td>
<td>$O(N)$</td>
</tr>
</tbody>
</table>

Which one performs better in practice?
A) selection
B) insertion
summary

-an *inversion* is a pair of items that are out of order
  -a sorted array has 0 inversions
  -an average (and worst) array has ~N^2 inversions

-thus, we must undo N^2 inversions

-to do better than $O(N^2)$ we must remove more than 1 inversion per step
  -(insertion sort only removes 1 inversion per step!)
what we want...

- a sorting algorithm that has subquadratic complexity

- swapping adjacent items removes exactly 1 inversion

\[
\begin{array}{cccccccc}
45 & -3 & 9 & 76 & 11 & -8 & 0 & \\
\end{array}
\]

swap removes 1 inversion

- what if we consider swapping nonadjacent pairs?

\[
\begin{array}{cccccccc}
45 & -3 & 9 & 76 & 11 & -8 & 0 & \\
\end{array}
\]

swap removes 7 inversions

- removes inversions not involved with the swap
next time...
- **reading**
  - chapters 8.1 - 8.4

- **homework**
  - assignment 3 due on Wednesday