Algorithms
Classifying algorithms by the rate of growth

Lecture 10 by Marina Barsky
Classifying algorithms by the rate of growth
O(0)?
$O(1)$
$O(\log n)$
$O(\log n)$
$O(1)$
$O(\log n)$
$O(n)$
$O(1)$
$O(\log n)$
$O(n)$
$O(n \log n)$
Examples

• **O(1)**
  • Getting the length of a given array
  • Getting the i-th element from *ArrayList*

• **O(n)**
  • Min/Max value in an array
  • Search for something in an unsorted list

• **O(n^2)**
  • Finding closest pair of points in a plane
Algorithms: practical and impractical
## What does it mean in practice

Assuming n=1,000 and 1ms per operation

<table>
<thead>
<tr>
<th>Name</th>
<th>Big O</th>
<th>Time to process</th>
<th>Max n per day</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant</td>
<td>O(1)</td>
<td>1 ms</td>
<td></td>
</tr>
<tr>
<td>Logarithmic</td>
<td>O(log n)</td>
<td>9.9 ms</td>
<td></td>
</tr>
<tr>
<td>Linear</td>
<td>O(n)</td>
<td>1 s</td>
<td>86,400,000</td>
</tr>
<tr>
<td>n log n</td>
<td>O(n log n)</td>
<td>9.9 s</td>
<td>3,943,234</td>
</tr>
<tr>
<td>Quadratic</td>
<td>O(n^2)</td>
<td>16.67 min</td>
<td>9,295</td>
</tr>
<tr>
<td>Cubic</td>
<td>O(n^3)</td>
<td>11.57 days</td>
<td>442</td>
</tr>
<tr>
<td>Exponential</td>
<td>O(2^n)</td>
<td>3.395*10^{290} years</td>
<td>26</td>
</tr>
<tr>
<td>Factorial</td>
<td>O(n!)</td>
<td>???</td>
<td>11</td>
</tr>
</tbody>
</table>
CPU with a clock speed of 2 gigahertz (GHz) can carry out two thousand million ($2 \times 10^9$) cycles (operations) per second.

- Algorithm which runs in $O(2^n)$ time will process 1 KB of input in $\sim 10^{300}$ years (more than 100 millennia)

- Processing 1 GB of input will take <0.001 ms by $O(\log n)$ algorithm, < 1 sec by $O(n)$ algorithm, and >32 years by $O(n^2)$ algorithm
Complexity of sorting
void sorting1 (array A)
    i = 1
    while i < length(A)
        j = i
        while j > 0 and A[j-1] > A[j]
            swap A[j] and A[j-1]
            j = j - 1
    i = i + 1

A. O(n)
B. O(n^2)
C. O(n^3)
D. None of the above
void sorting1 (array A)
  i = 1
  while i < length(A)
    j = i

    while j > 0 and A[j-1] > A[j]
      swap A[j] and A[j-1]
      j = j - 1

  i = i + 1

Sorting 1 is a...

A. Bubble sort
B. Insertion sort
C. Selection sort
D. None of the above
void sorting2 (array A) 
    n = length(A) 
    swapped = false 
    do: 
        for i from 0 to n-1 
            if A[i-1] > A[i]: 
                swap A[i-1] and A[i] 
                swapped = true 
        n = n - 1 
    while (swapped) 

A. O(n) 
B. O(n^2) 
C. O(n^3) 
D. None of the above
void sorting2 (array A)  
    n = length(A)  
    swapped = false  
    do:  
        for i from 0 to n-1  
            if A[i-1] > A[i]:  
                swap A[i-1] and A[i]  
                swapped = true  
            n = n - 1  
    while (swapped)

A. Bubble sort  
B. Insertion sort  
C. Selection sort  
D. None of the above
Back to basic Data Structures

Complexity of operations on Arrays and Linked Lists
ArrayList and LinkedList: algorithms

• Read:
  • get (index i)
  • indexOf (Object o)

• Edit:
  • add()
  • remove()
Running time of common operations for ArrayList and LinkedList

<table>
<thead>
<tr>
<th>Operation</th>
<th>ArrayList</th>
<th>LinkedList</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get i-th element</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Search for an element (indexOf)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add new element at the end</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add element at position $i$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Remove from the end</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Remove from position $i$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Resize when full</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Running time of common operations for ArrayList and LinkedList

<table>
<thead>
<tr>
<th>Operation</th>
<th>ArrayList</th>
<th>LinkedList</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get i-th element</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Search for an element (indexOf)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Add new element at the end</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Remove from the end</td>
<td>O(1)</td>
<td>O(1) with tail pointer</td>
</tr>
<tr>
<td>Remove from position i</td>
<td>O(n)</td>
<td>Traverse in O(n) then O(1)</td>
</tr>
<tr>
<td>Resize when full</td>
<td>O(n)</td>
<td>n/a: never full</td>
</tr>
</tbody>
</table>
Knowing that worst-case performance of the `add()` method of ArrayLists is $O(n)$, what is the time complexity of the following loop?

```java
void addAll(int n) {
    ArrayList list;
    for (int i = 0; i<n; i++){
        list.add(i);
    }
}
```

A. $O(n^2)$  
B. $O(n)$  
C. $O(1)$  
D. None of the above
Resizing arrays: Amortized analysis

Sometimes, looking at the individual worst-case may be too severe.
We may want to know the total worst-case cost for a sequence of operations.

● In dynamic arrays we only resize every so often.
● Many O(1) operations are followed by an O(n) operation.
● What is the total cost of inserting n elements? $O(n^2)$?
Amortized cost: Given a sequence of $n$ operations, the amortized cost of each operation is:

$$\text{Cost (n operations)} = \frac{n}{n}$$
Dynamic arrays: amortized cost of \textit{add}

Intuition:

• Say we originally have \(k\) elements in the Array List, and the list is half-full
• Now we can add another \(k\) elements, each in time \(O(1)\) – in total \(k \times O(1) = O(k)\) steps

• Now we need to resize by copying \(2k\) elements in time \(O(2k) = O(k)\)

So in total adding \(k\) new elements takes \(O(k) + O(k) = O(k)\) which is \(O(k)/k = O(1)\) amortized cost per single \textit{add}
Aggregate method: cost of $n$ calls to \textit{add}

- Let’s start with array of size 1
- If we choose the strategy of doubling the size of the array on resizing, then during the insertion of $n$ elements we will double and copy in total $1 + 2 + 4 + 8 + ... n/2$ elements
- In total we will perform copy $\log n$ times

$1 + 1\times2 + 1\times2\times2 + 1\times2\times2\times2 + ... 1\times2^{\log n} = 1\times2^0 + 1\times2^1 + 1\times2^2 + 1\times2^3 + ... 1\times2^{\log n}$

What do we see here?
Aggregate method: cost of \( n \) calls to \textit{add}

\[ 1 \times 2^0 + 1 \times 2^1 + 1 \times 2^2 + 1 \times 2^3 + \ldots + 1 \times 2^{\log n} \]

- This is a sum of geometric series with \( a_0=1, \ d=2 \), and total of \( k=\log n \) elements

- The sum of the first \( k \) elements of the geometric series:
  \[
  \text{Sum} = a_0(d^k - 1)/(d - 1)
  \]

- For our case it is:
  \[ 2^k - 1, \ \text{and} \ k = \log n \]
  and \[ 2^{\log n} = n \]
Aggregate method: cost of $n$ calls to \textit{add}

\[1 \times 2^0 + 1 \times 2^1 + 1 \times 2^2 + 1 \times 2^3 + \ldots + 1 \times 2^{\log n}\]

- This sum is $O(2^{\log n}) = O(n)$
- Thus the cost of $n \times \text{add}()$ is $O(n)$, which is $O(1)$ per add

\textbf{Corollary:}

The amortized cost of \textit{add} in dynamic array is $O(1)$