Object-oriented programming and data-structures

CS/ENGRD 2110
SUMMER 2018
Lecture 6 Recap

- Introduced the notion of recursion and backtracking recursion
- Discussed a number of problems that could be solved using recursions
- Hinted that recursion could be expensive.
  - What does expensive mean?
This lecture

- Formalise the notion of “expensive”
- Introduce Big-O notation
- Proofs of Big-O
- Applying Big-O to datastructures
What Makes a Good Algorithm?

Suppose you have two possible algorithms that do the same thing; which is better?

Ex: is retrieving an element from `LinkedList` better than from `ArrayList`?
What Makes a Good Algorithm?

Suppose you have two possible algorithms that do the same thing; which is better?

Ex: is retrieving an element from LinkedList better than from ArrayList?

What do we mean by better?
- Faster?
- Less space?
- Easier to code?
- Easier to maintain?
- Required for homework?
What Makes a Good Algorithm?

Suppose you have two possible algorithms that do the same thing; which is better?

Ex: is retrieving an element from LinkedList better than from ArrayList?

What do we mean by better?

- Faster?
- Less space?
- Easier to code?
- Easier to maintain?
- Required for homework?

FIRST, Aim for simplicity, ease of understanding, correctness.

SECOND, Worry about efficiency only when it is needed.
Suppose you have two possible algorithms that do the same thing; which is better?

Ex: is retrieving an element from `LinkedList` better than from `ArrayList`?

What do we mean by better?
- Faster?
- Less space?
- Easier to code?
- Easier to maintain?
- Required for homework?

FIRST, Aim for simplicity, ease of understanding, correctness.

SECOND, Worry about efficiency only when it is needed.

How do we measure speed of an algorithm?
Basic Step: one “constant time” operation

Constant time operation: its time doesn’t depend on the size or length of anything. Always roughly the same. Time is bounded above by some number

Basic step:
- Input/output of a number
- Access value of primitive-type variable, array element, or object field
- Assign to variable, array element, or object field
- Do one arithmetic or logical operation
- Method call (not counting arg evaluation and execution of method body)
// Store sum of 1..n consecutive integers in sum

sum = 0;

// inv: sum = sum of 1..(k-1)
for (int k = 1; k <= n; k = k + 1) {
    sum = sum + k;
}

All basic steps take time 1.
// Store sum of 1..n consecutive integers in sum
sum = 0;
// inv: sum = sum of 1..(k-1)
for (int k = 1; k <= n; k = k+1){
    sum = sum + k;
}

All basic steps take time 1.

---

Statement: # times done
sum = 0; 1
k = 1; 1
k <= n n+1
k = k+1; n
sum = sum + k; ______

Total steps: 3n + 3
// Store sum of 1..n consecutive integers in sum
sum = 0;

// inv: sum = sum of 1..(k-1)
for (int k = 1; k <= n; k = k + 1) {
    sum = sum + k;
}

All basic steps take time 1.
There are n loop iterations. Therefore, takes time proportional to n.

Statement: # times done
sum = 0; 1
k = 1; 1
k <= n; n + 1
k = k + 1; n
sum = sum + k;

Total steps: 3n + 3

Linear algorithm in n
// Store n copies of 'c' in s
s = "";
// inv: s contains k-1 copies of 'c'
for (int k = 1; k <= n; k = k+1){
    s = s + 'c';
}

<table>
<thead>
<tr>
<th>Statement:</th>
<th># times done</th>
</tr>
</thead>
<tbody>
<tr>
<td>s = &quot;&quot;;</td>
<td>1</td>
</tr>
<tr>
<td>k = 1;</td>
<td>1</td>
</tr>
<tr>
<td>k &lt;= n</td>
<td>n+1</td>
</tr>
<tr>
<td>k = k + 1;</td>
<td>n</td>
</tr>
<tr>
<td>s = s + 'c';</td>
<td>n</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Total steps:</td>
<td>3n + 3</td>
</tr>
</tbody>
</table>

Not all operations are basic steps
// Store n copies of ‘c’ in s
s = "";
// inv: s contains k-1 copies of ‘c’
for (int k = 1; k <= n; k = k+1){
    s = s + 'c';
}

Concatenation is not a basic step. For each k, catenation creates and fills k array elements.
String Concatenation

\[ s = s + \text{"c"}; \] is NOT constant time.
It takes time proportional to \(1 + \text{length of } s\)
String Concatenation

s = s + "c"; is NOT constant time.
It takes time proportional to 1 + length of s
String Concatenation

s = s + "c"; is NOT constant time. It takes time proportional to 1 + length of s
Not all operations are basic steps

```java
// Store n copies of 'c' in s
s = "";
// inv: s contains k-1 copies of 'c'
for (int k = 1; k <= n; k = k+1){
    s = s + 'c';
}
```

Concatenation is not a basic step. For each k, concatenation creates and fills k array elements.

<table>
<thead>
<tr>
<th>Statement</th>
<th># times</th>
<th># steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>s = &quot;&quot;;</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>k = 1;</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>k &lt;= n</td>
<td>n+1</td>
<td>1</td>
</tr>
<tr>
<td>k = k+1;</td>
<td>n</td>
<td>1</td>
</tr>
<tr>
<td>s = s + 'c';</td>
<td>n</td>
<td>k</td>
</tr>
</tbody>
</table>

Total steps: \( n(n+1)/2 + 2n + 3 \)

Quadratic algorithm in n
Linear versus quadratic

// Store sum of 1..n in sum
sum = 0;
// inv: sum = sum of 1..(k-1)
for (int k = 1; k <= n; k = k+1)
    sum = sum + n

Linear algorithm

// Store n copies of ‘c’ in s
s = "";
// inv: s contains k-1 copies of ‘c’
for (int k = 1; k = n; k = k+1)
    s = s + ‘c’;

Quadratic algorithm

In comparing the runtimes of these algorithms, the exact number of basic steps is not important. What’s important is that
- One is linear in n—takes time proportional to n
- One is quadratic in n—takes time proportional to n^2
Looking at execution speed

Number of operations executed

0 1 2 3 ... size n of the input

Constant time
Looking at execution speed

Number of operations executed

- Constant time
- \( n \) ops
- \( n + 2 \) ops
- \( 2n + 2 \) ops

size n of the input

0 1 2 3 ...
Looking at execution speed

- $2n+2$, $n+2$, $n$ are all linear in $n$, proportional to $n$.
Looking at execution speed

Number of operations executed

0 1 2 3  ...

size n of the input

Constant time

2n + 2 ops

n + 2 ops

n ops

n*n ops
What do we want from a definition of “runtime complexity”? 

Number of operations executed

- $5$ ops
- $2 + n$ ops
- $n^2$ ops

size $n$ of problem
What do we want from a definition of “runtime complexity”?

1. Distinguish among cases for large n, not small n
What do we want from a definition of “runtime complexity”?

1. Distinguish among cases for large \( n \), not small \( n \)

2. Distinguish among important cases, like
   - \( n \times n \) basic operations
   - \( n \) basic operations
   - \( \log n \) basic operations
   - 5 basic operations
What do we want from a definition of “runtime complexity”?

1. Distinguish among cases for large $n$, not small $n$
2. Distinguish among important cases, like
   - $n^2$ basic operations
   - $n$ basic operations
   - $\log n$ basic operations
   - 5 basic operations
3. Don’t distinguish among trivially different cases.
   - 5 or 50 operations
   - $n$, $n+2$, or $4n$ operations

<table>
<thead>
<tr>
<th>size $n$ of problem</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>…</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of operations executed</td>
<td>5 ops</td>
<td>2+n ops</td>
<td>$n^2$ ops</td>
<td>5 ops</td>
<td>5 ops</td>
</tr>
</tbody>
</table>

Number of operations executed vs. size $n$ of problem.
"Big O" Notation

Formal definition: \( f(n) \) is \( O(g(n)) \) if there exist constants \( c > 0 \) and \( N \geq 0 \) such that for all \( n \geq N \), \( f(n) \leq c \cdot g(n) \)
"Big O" Notation

Formal definition: \( f(n) \) is \( O(g(n)) \) if there exist constants \( c > 0 \) and \( N \geq 0 \) such that for all \( n \geq N \), \( f(n) \leq c \cdot g(n) \)

Get out far enough (for \( n \geq N \))
\( f(n) \) is at most \( c \cdot g(n) \)
"Big O" Notation

Formal definition: $f(n)$ is $O(g(n))$ if there exist constants $c > 0$ and $N \geq 0$ such that for all $n \geq N$, $f(n) \leq c \cdot g(n)$

Get out far enough (for $n \geq N$)

$f(n)$ is at most $c \cdot g(n)$

Intuitively, $f(n)$ is $O(g(n))$ means that $f(n)$ grows like $g(n)$ or slower
"Big O" Notation

Formal definition: $f(n)$ is $O(g(n))$ if there exist constants $c > 0$ and $N \geq 0$ such that for all $n \geq N$, $f(n) \leq c \cdot g(n)$

Intuitively, $f(n)$ is $O(g(n))$ means that $f(n)$ grows like $g(n)$ or slower.
Prove that \( (2n^2 + n) \) is \( O(n^2) \)

**Formal definition:** \( f(n) \) is \( O(g(n)) \) if there exist constants \( c > 0 \) and \( N \geq 0 \) such that for all \( n \geq N \), \( f(n) \leq c \cdot g(n) \)

**Example:** Prove that \( (2n^2 + n) \) is \( O(n^2) \)

**Methodology:**

- Start with \( f(n) \) and slowly transform into \( c \cdot g(n) \):
  - Use = and \( \leq \) and < steps
  - At appropriate point, can choose \( N \) to help calculation
  - At appropriate point, can choose \( c \) to help calculation
Prove that \((2n^2 + n)\) is \(O(n^2)\)

Formal definition: \(f(n)\) is \(O(g(n))\) if there exist constants \(c > 0\) and \(N \geq 0\) such that for all \(n \geq N\), \(f(n) \leq c \cdot g(n)\)

Example: Prove that \((2n^2 + n)\) is \(O(n^2)\)

\[
f(n) = 2n^2 + n
\]

Transform \(f(n)\) into \(c \cdot g(n)\):
- Use \(=, \leq, <\) steps
- Choose \(N\) to help calc.
- Choose \(c\) to help calc
Prove that \((2n^2 + n)\) is \(O(n^2)\)

Formal definition: \(f(n)\) is \(O(g(n))\) if there exist constants \(c > 0\) and \(N \geq 0\) such that for all \(n \geq N\), \(f(n) \leq c \cdot g(n)\)

Example: Prove that \((2n^2 + n)\) is \(O(n^2)\)

\[
f(n) = 2n^2 + n \leq 3n^2 = 3 \cdot g(n)
\]

Choose \(N = 1\)

Transform \(f(n)\) into \(c \cdot g(n)\):
- Use =, <=, < steps
- Choose \(N\) to help calc.
- Choose \(c\) to help calc
Prove that \((2n^2 + n)\) is \(O(n^2)\)

Formal definition: \(f(n)\) is \(O(g(n))\) if there exist constants \(c > 0\) and \(N \geq 0\) such that for all \(n \geq N\), \(f(n) \leq c \cdot g(n)\)

Example: Prove that \((2n^2 + n)\) is \(O(n^2)\)

\[
\begin{align*}
f(n) &= \text{<definition of } f(n)> \\
&= 2n^2 + n \\
&\leq \text{<for } n \geq 1, \ n \leq n^2> \\
&= 2n^2 + n^2 \\
&= \text{<arith>} \\
&= 3n^2
\end{align*}
\]

Transform \(f(n)\) into \(c \cdot g(n)\):
- Use =, \(\leq\), < steps
- Choose \(N\) to help calc.
- Choose \(c\) to help calc

Choose 
\(N = 1\)
Prove that \((2n^2 + n)\) is \(O(n^2)\)

Formal definition: \(f(n)\) is \(O(g(n))\) if there exist constants \(c > 0\) and \(N \geq 0\) such that for all \(n \geq N\), \(f(n) \leq c \cdot g(n)\)

Example: Prove that \((2n^2 + n)\) is \(O(n^2)\)

\[
\begin{align*}
f(n) &= \text{<definition of } f(n)> \quad 2n^2 + n \\
\leq& \quad \text{<for } n \geq 1, \ n \leq n^2> \quad 2n^2 + n^2 \\
&= \quad \text{<arith>} \quad 3n^2 \\
&= \quad \text{<definition of } g(n) = n^2> \quad 3 \cdot g(n)
\end{align*}
\]

Transform \(f(n)\) into \(c \cdot g(n)\):

- Use \(=, \leq, <\) steps
- Choose \(N\) to help calc.
- Choose \(c\) to help calc

Choose \(N = 1\) and \(c = 3\)
Prove that $100\, n + \log n$ is $O(n)$

Formal definition: $f(n)$ is $O(g(n))$ if there exist constants $c > 0$ and $N \geq 0$ such that for all $n \geq N$, $f(n) \leq c \cdot g(n)$

\[
f(n) = \begin{cases} \text{<put in what } f(n) \text{ is>} \\
100\, n + \log n \\
\leq \begin{cases} \text{<We know } \log n \leq n \text{ for } n \geq 1> \\
100\, n + n \\
= \begin{cases} \text{<arith>} \\
101\, n \\
= \begin{cases} \text{<g(n) = n>} \\
101\, g(n)
\end{cases}
\end{cases}
\end{cases}
\]

Choose $N = 1$ and $c = 101$
Prove that $100n + \log n$ is $O(n)$

Formal definition: $f(n)$ is $O(g(n))$ if there exist constants $c > 0$ and $N \geq 0$ such that for all $n \geq N$, $f(n) \leq c \cdot g(n)$

\[
f(n) = 100n + \log n
\]
\[
\leq 100n + n
\]
\[
= 101n
\]

Choose $N = 1$ and $c = 101$
O(…) Examples

Let $f(n) = 3n^2 + 6n - 7$
- $f(n)$ is $O(n^2)$
- $f(n)$ is $O(n^3)$
- $f(n)$ is $O(n^4)$

Let $p(n) = 4n \log n + 34n - 89$
- $p(n)$ is $O(n \log n)$
- $p(n)$ is $O(n^2)$

Let $h(n) = 20 \cdot 2^n + 40n$
- $h(n)$ is $O(2^n)$

Let $a(n) = 34$
- $a(n)$ is $O(1)$
O(...) Examples

Let \( f(n) = 3n^2 + 6n - 7 \)
- \( f(n) \) is \( O(n^2) \)
- \( f(n) \) is \( O(n^3) \)
- \( f(n) \) is \( O(n^4) \)

\( p(n) = 4n \log n + 34n - 89 \)
- \( p(n) \) is \( O(n \log n) \)
- \( p(n) \) is \( O(n^2) \)

\( h(n) = 20 \cdot 2^n + 40n \)
- \( h(n) \) is \( O(2^n) \)

\( a(n) = 34 \)
- \( a(n) \) is \( O(1) \)

Only the leading term (the term that grows most rapidly) matters

If it’s \( O(n^2) \), it’s also \( O(n^3) \) etc! However, we always use the smallest one
Do NOT say or write $f(n) = O(g(n))$

Formal definition: $f(n)$ is $O(g(n))$ if there exist constants $c > 0$ and $N \geq 0$ such that for all $n \geq N$, $f(n) \leq c \cdot g(n)$

$f(n) = O(g(n))$ is simply WRONG. Mathematically, it is a disaster. You see it sometimes, even in textbooks. Don’t read such things.
Do NOT say or write \( f(n) = O(g(n)) \)

\[
\text{Formal definition: } f(n) \text{ is } O(g(n)) \text{ if there exist constants } c > 0 \text{ and } N \geq 0 \text{ such that for all } n \geq N, \quad f(n) \leq c \cdot g(n)
\]

\( f(n) = O(g(n)) \) is simply \textit{WRONG}. Mathematically, it is a disaster. You see it sometimes, even in textbooks. Don’t read such things.

Here’s an example to show what happens when we use = this way.

We know that \( n+2 \) is \( O(n) \) and \( n+3 \) is \( O(n) \). Suppose we use =

\[
\begin{align*}
n+2 & = O(n) \\
n+3 & = O(n)
\end{align*}
\]

But then, by transitivity of equality, we have \( n+2 = n+3 \).
We have proved something that is false. Not good.
Problem-size examples

- Suppose a computer can execute 1000 operations per second; how large a problem can we solve?

<table>
<thead>
<tr>
<th>operations</th>
<th>1 second</th>
<th>1 minute</th>
<th>1 hour</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>1000</td>
<td>60,000</td>
<td>3,600,000</td>
</tr>
<tr>
<td>n log n</td>
<td>140</td>
<td>4893</td>
<td>200,000</td>
</tr>
<tr>
<td>n²</td>
<td>31</td>
<td>244</td>
<td>1897</td>
</tr>
<tr>
<td>3n²</td>
<td>18</td>
<td>144</td>
<td>1096</td>
</tr>
<tr>
<td>n³</td>
<td>10</td>
<td>39</td>
<td>153</td>
</tr>
<tr>
<td>2ⁿ</td>
<td>9</td>
<td>15</td>
<td>21</td>
</tr>
</tbody>
</table>
Big-O notation is not just for time

- Applies to both time complexity and space complexity

- Same reasoning in both cases

- In this class, we’ll focus primarily on time complexity
Recall the two types of List in Java Collections (<List>)
  - ArrayList
  - LinkedList

- ArrayList is backed by an underlying array
- LinkedList is a **doubly linked list** and has pointers to the head/tail of the queue. Each element has a pointer to previous/next element
Array Lists

- ArrayList is backed by an underlying array

- Arrays allow direct access to each element
  - What is the cost of accessing the ith element of the array?
    - $O(1)$
Array Lists

- ArrayList is backed by an underlying array
- Arrays allow direct access to each element
  - What is the cost of accessing the ith element of the array?
- What is the cost of inserting an element
  - May need to allocate a new array and copy all the previous elements into new array
Array Lists

- ArrayList is backed by an underlying array

- Arrays allow direct access to each element
  - What is the cost of accessing the ith element of the array?
  - What is the cost of inserting an element
    - May need to allocate a new array and copy all the previous elements into new array

Amortised
\[O(1) / O(n)\]
Array Lists

- ArrayList is backed by an underlying array

- Arrays allow direct access to each element
  - What is the cost of accessing the ith element of the array?
  - What is the cost of inserting an element
    - May need to allocate a new array and copy all the previous elements into new array
  - What is the cost of deleting the ith element
    - When delete an element, have to shift all the remaining elements to the left
Array Lists

- ArrayList is backed by an underlying array

- Arrays allow direct access to each element
  - What is the cost of accessing the ith element of the array?
  - What is the cost of inserting an element
    - May need to allocate a new array and copy all the previous elements into new array
  - What is the cost of deleting the ith element
    - When delete an element, have to shift all the remaining elements to the left

\[ O(n) \]
Linked Lists

- LinkedList is a **doubly linked list** and has pointers to the head/tail of the queue. Each element has a pointer to previous/next element.

- What is the cost of accessing the ith element of the array?

- What is the cost of inserting an element to the head?

- What is the cost of deleting the ith element?
Linked Lists

- LinkedList is a **doubly linked list** and has pointers to the head/tail of the queue. Each element has a pointer to previous/next element.

- What is the cost of accessing the ith element of the array?
  - Need to start from the head and follow pointers
  - $O(n)$

- What is the cost of inserting an element to the head

- What is the cost of deleting the ith element
Linked List is a **doubly linked list** and has pointers to the head/tail of the queue. Each element has a pointer to previous/next element.

- What is the cost of accessing the ith element of the array?
  - Need to start from the head and follow pointers: \( O(n) \)
- What is the cost of inserting an element to the head?
  - Direct access through head pointer: \( O(1) \)
- What is the cost of deleting the ith element?

[Diagram of a doubly linked list]
Linked Lists

- LinkedList is a **doubly linked list** and has pointers to the head/tail of the queue. Each element has a pointer to previous/next element

- What is the cost of accessing the ith element of the array?
  - Need to start from the head and follow pointers  
    - O(n)

- What is the cost of inserting an element to the head?
  - Direct access through head pointer  
    - O(1)

- What is the cost of deleting the ith element?
  - Need to find the ith element first  
    - O(n)
Linked Lists

- LinkedList is a **doubly linked list** and has pointers to the head/tail of the queue. Each element has a pointer to previous/next element.

- What is the cost of accessing the ith element of the array?
  - Need to start from the head and follow pointers
  - Cost: $O(n)$

- What is the cost of inserting an element to the head?
  - Direct access through head pointer
  - Cost: $O(1)$

- What is the cost of deleting the ith element?
  - Need to find the ith element first
  - Cost: $O(n)$

- What about deleting the head/tail element?
  - Cost: $O(n)$
Do the performance numbers match up?
Only tell half the story ...

- On my machine, ArrayList add is 5 times faster than LinkedList add.
- Underlying reason is memory allocation is much more efficient for arrays than linked list: arrays can allocate large blocks of memory at once while you have to allocate individual nodes for a linked list.