Asymptotic Complexity

CS 214, Fall 2019
A comparison
How long would it take to...

- Get or set the \( n \)th element?
- Add an element to the front?
- Add an element to the back?
- Determine whether \( x \) is an element?
Getting the $n$th element

```python
def list_nth(lst, n):
    def loop(i, link):
        if not link: error('list_nth: out of bounds')
        elif i == 0: return link.car
        else: return loop(i - 1, link.cdr)
    loop(n, lst.head)
```

The loop in `list_nth` repeats $n$ times. `array_nth` has no loop.
Getting the *n*th element

def list_nth(lst, n):
    def loop(i, link):
        if not link: error('list_nth: out of bounds')
        elif i == 0: return link.car
        else: return loop(i - 1, link.cdr)
    loop(n, lst.head)

def array_nth(array, n):
    if n < array.len:
        return array.elems[n]
    else:
        error('array_nth: out of bounds')
Getting the $n$th element

```python
def list_nth(lst, n):
    def loop(i, link):
        if not link: error('list_nth: out of bounds')
        elif i == 0: return link.car
        else: return loop(i - 1, link.cdr)
    loop(n, lst.head)

def array_nth(array, n):
    if n < array.len:
        return array.elems[n]
    else:
        error('array_nth: out of bounds')

The loop in list_nth repeats $n$ times. array_nth has no loop.
Adding an element to the front

def list_push_front(lst, val):
    lst.head = cons(val, lst.head)
Adding an element to the front

```python
def list_push_front(lst, val):
    lst.head = cons(val, lst.head)

def array_push_front(array, val):
    if array.len == array.elems.len():
        error('array_push_front: out of space')
    let i = array.len
    while i > 0:
        array.elems[i] = array.elems[i - 1]
        i = i - 1
    array.elems[0] = val
    array.len = array.len + 1
```
Adding an element to the front

```python
def list_push_front(lst, val):
    lst.head = cons(val, lst.head)

def array_push_front(array, val):
    if array.len == array.elems.len():
        error('array_push_front: out of space')
    let i = array.len
    while i > 0:
        array.elems[i] = array.elems[i - 1]
        i = i - 1
    array.elems[0] = val
    array.len = array.len + 1
```

`list_push_front` is loop-free, whereas `array_push_front` loops `array.len` times.
Breaking down list\_nth

def list\_nth(lst, n):
    link = lst.head
    for i in n:
        link = link.cdr
    return link.car

\( T \) stands for “time”

\[ T_{\text{list\_nth}}(n) = \]
Breaking down list-nth

def list_nth(lst, n):
    link = lst.head
    for i in n:
        link = link.cdr
    return link.car

(\(T\) stands for “time”)

\[ T_{\text{list\_nth}}(n) = T_{\text{get\_head}} + \]
Breaking down list-nth

def list_nth(lst, n):
    link = lst.head
    for i in n:
        link = link.cdr
    return link.car

($T$ stands for “time”)

\[
T_{\text{list_nth}}(n) = T_{\text{get head}} + T_{\text{loop setup}} +
\]
Breaking down list-nth

def list_nth(lst, n):
    link = lst.head
    for i in n:
        link = link.cdr
    return link.car

(T stands for “time”)

\[ T_{list\_nth}(n) = T_{get\ head} + T_{loop\ setup} + nT_{get\ cdr} + \]


Breaking down list-nth

```python
def list_nth(lst, n):
    link = lst.head
    for i in n:
        link = link.cdr
    return link.car
```

($T$ stands for “time”)

$$T_{list\_nth}(n) = T_{\text{get head}} + T_{\text{loop setup}} + n T_{\text{get cdr}} + n T_{\text{assign link}} + n T_{\text{loop inc}} +$$
Breaking down list–nth

```python
def list_nth(lst, n):
    let link = lst.head
    for i in n:
        link = link.cdr
    return link.car
```

($T$ stands for “time”)

\[
T_{\text{list–nth}}(n) = T_{\text{get head}} + T_{\text{loop setup}} + nT_{\text{get cdr}} + \\
nT_{\text{assign link}} + nT_{\text{loop inc}} + T_{\text{get car}}
\]
Breaking down listₙ-th

```python
def list_nth(lst, n):
    link = lst.head
    for i in n:
        link = link.cdr
    return link.car
```

($T$ stands for “time”)

\[
T_{\text{list nth}}(n) = T_{\text{get head}} + T_{\text{loop setup}} + nT_{\text{get cdr}} +
\]

\[
nT_{\text{assign link}} + nT_{\text{loop inc}} + T_{\text{get car}}
\]

Let \(c_1 = T_{\text{get head}} + T_{\text{loop setup}} + T_{\text{get car}}\)
Breaking down `list_nth`

```python
def list_nth(lst, n):
    link = lst.head
    for i in n:
        link = link.cdr
    return link.car
```

($T$ stands for “time”)

\[
T_{\text{list_nth}}(n) = T_{\text{get head}} + T_{\text{loop setup}} + nT_{\text{get cdr}} + \\
nT_{\text{assign link}} + nT_{\text{loop inc}} + T_{\text{get car}}
\]

Let $c_1 = T_{\text{get head}} + T_{\text{loop setup}} + T_{\text{get car}}$

Let $c_2 = T_{\text{get cdr}} + T_{\text{assign link}} + T_{\text{loop inc}}$
Breaking down list\-nth

```python
def list_nth(lst, n):
    link = lst.head
    for _ in range(n):
        link = link.cdr
    return link.car
```

($T$ stands for “time”)

\[
T_{\text{list_nth}}(n) = T_{\text{get head}} + T_{\text{loop setup}} + nT_{\text{get cdr}} + \\
   nT_{\text{assign link}} + nT_{\text{loop inc}} + T_{\text{get car}}
\]

\[
T_{\text{list_nth}}(n) = c_1 + c_2n
\]

Let $c_1 = T_{\text{get head}} + T_{\text{loop setup}} + T_{\text{get car}}$

Let $c_2 = T_{\text{get cdr}} + T_{\text{assign link}} + T_{\text{loop inc}}$
### Operation time comparison

<table>
<thead>
<tr>
<th>Operation</th>
<th>List</th>
<th>Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>nth</td>
<td>$c_1 + c_2n$</td>
<td>$d_1$</td>
</tr>
<tr>
<td>push_front</td>
<td>$e_1$</td>
<td>$f_1 + f_2n$</td>
</tr>
</tbody>
</table>

No matter what the values of $c_1$, $c_2$, and $e_1$ are, if $n$ gets large enough then $c_1 + c_2n$ will be larger than $e_1$. The same cannot be said when comparing $c_1 + c_2n$ to $f_1 + f_2n$. 


No matter what the values of $c_1$, $c_2$, and $e_1$ are, if $n$ gets large enough then $c_1 + c_2n$ will be larger than $e_1$. 

<table>
<thead>
<tr>
<th></th>
<th>list</th>
<th>array</th>
</tr>
</thead>
<tbody>
<tr>
<td>nth</td>
<td>$c_1 + c_2n$</td>
<td>$d_1$</td>
</tr>
<tr>
<td>push_front</td>
<td>$e_1$</td>
<td>$f_1 + f_2n$</td>
</tr>
</tbody>
</table>
Operation time comparison

<table>
<thead>
<tr>
<th></th>
<th>list</th>
<th>array</th>
</tr>
</thead>
<tbody>
<tr>
<td>nth</td>
<td>(c_1 + c_2n)</td>
<td>(d_1)</td>
</tr>
<tr>
<td>push_front</td>
<td>(e_1)</td>
<td>(f_1 + f_2n)</td>
</tr>
</tbody>
</table>

No matter what the values of \(c_1\), \(c_2\), and \(e_1\) are, if \(n\) gets large enough then \(c_1 + c_2n\) will be larger than \(e_1\).

The same cannot be said when comparing \(c_1 + c_2n\) to \(f_1 + f_2n\).
Complexity classes

There’s a sense in which $c_1 + c_2 n$ and $f_1 + f_2 n$ are similar.
Complexity classes

There’s a sense in which $c_1 + c_2 n$ and $f_1 + f_2 n$ are similar. We call this sense $O(n)$. 
Another example: insertion sort

# : Link[num?] -> Link[num?]
def insertion_sort(lst):
    def insert(elt, link):
        if not link or elt < link.car:
            cons(elt, link)
        else:
            cons(link.car, insert(elt, link.cdr))

    result = None
    curr = lst.head
    while curr:
        result = insert(curr.car, result)
        curr = curr.cdr
    result
Another example: insertion sort

```python
# : Link[num?] \rightarrow Link[num?] 
def insertion_sort(lst):
    def insert(elt, link):
        if not link or elt < link.car:
            cons(elt, link)
        else:
            cons(link.car, insert(elt, link.cdr))

    let result = None
    let curr = lst.head
    while curr:
        result = insert(curr.car, result)
        curr = curr.cdr
    result

Nested loops of length $n$: $\mathcal{O}(n^2)$
```
Another example: merge sort helpers (1/2)

```python
# : Link[num?] Link[num?] -> Link[num?]
def merge(lnk1, lnk2):
    if lnk1 and lnk2:
        if lnk1.car < lnk2.car:
            cons(lnk1.car, merge(lnk1.cdr, lnk2))
        else:
            cons(lnk2.car, merge(lnk1, lnk2.cdr))
    else: lnk1 or lnk2
```

merge is $O(|lnk1| + |lnk2|) = O(n)$. 


Another example: merge sort helpers (1/2)

```python
# : Link[num?] Link[num?] -> Link[num?]
def merge(lnk1, lnk2):
    if lnk1 and lnk2:
        if lnk1.car < lnk2.car:
            cons(lnk1.car, merge(lnk1.cdr, lnk2))
        else:
            cons(lnk2.car, merge(lnk1, lnk2.cdr))
    else:
        lnk1 or lnk2

merge is $O(|lnk1| + |lnk2|) = O(n)$.
```
Another example: merge sort helpers (2/2)

def odds(link):
    if link:
        cons(link.car, evens(link.cdr))
    else:
        None

def evens(link):
    if link:
        odds(link.cdr)
    else:
        None
def odds(link):
    if link:
        cons(link.car, evens(link.cdr))
    else:
        None

def evens(link):
    if link:
        odds(link.cdr)
    else:
        None

odds and evens are both $O(n)$.
Another example: merge sort

```python
# : Link[num?] → Link[num?]  
def merge_sort(link):
    if not link or not link.cdr:
        link
    else:
        merge(merge_sort(odds(link)),
              merge_sort(evens(link)))
```
Another example: merge sort

# : Link[num?] -> Link[num?]
def merge_sort(link):
    if not link or not link.cdr:
        link
    else:
        merge(merge_sort(odds(link)),
              merge_sort(evens(link)))

In each recursion we take \(O(n)\). How many times do we recur?
Another example: merge sort

```python
# : Link[num?] -> Link[num?]
def merge_sort(link):
    if not link or not link.cdr:
        return link
    else:
        return merge(merge_sort(odds(link)),
                      merge_sort(evens(link)))
```

In each recursion we take $\mathcal{O}(n)$. How many times do we recur? $\mathcal{O}(\log n)$ times.
Merge sort versus insertion sort

Merge sort takes $O(n \log n)$. Insertion sort takes $O(n^2)$. What does this mean concretely?

<table>
<thead>
<tr>
<th>$n$</th>
<th>$n^2$</th>
<th>$n \log n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>100</td>
<td>10</td>
</tr>
<tr>
<td>100</td>
<td>10,000</td>
<td>200</td>
</tr>
<tr>
<td>1,000</td>
<td>1,000,000</td>
<td>3,000</td>
</tr>
<tr>
<td>10,000</td>
<td>100,000,000</td>
<td>40,000</td>
</tr>
<tr>
<td>100,000</td>
<td>10,000,000,000</td>
<td>500,000</td>
</tr>
</tbody>
</table>
Merge sort versus insertion sort

Merge sort takes $O(n \log n)$. Insertion sort takes $O(n^2)$. What does this mean concretely?

<table>
<thead>
<tr>
<th>$n$</th>
<th>$n^2$</th>
<th>$n \log n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1e1</td>
<td>1e2</td>
<td>1e1</td>
</tr>
<tr>
<td>1e2</td>
<td>1e4</td>
<td>2e2</td>
</tr>
<tr>
<td>1e3</td>
<td>1e6</td>
<td>3e3</td>
</tr>
<tr>
<td>1e4</td>
<td>1e8</td>
<td>4e4</td>
</tr>
<tr>
<td>1e5</td>
<td>1e10</td>
<td>5e5</td>
</tr>
</tbody>
</table>
Merge sort versus insertion sort

Merge sort takes $\mathcal{O}(n \log n)$. Insertion sort takes $\mathcal{O}(n^2)$. What does this mean concretely?

$$
\begin{array}{c||c|c}
  n & n^2 & 10^{12} n \log n \\
  \hline
  1e1 & 1e2 & 1e1 \\
  1e2 & 1e4 & 2e2 \\
  1e3 & 1e6 & 3e3 \\
  1e4 & 1e8 & 4e4 \\
  1e5 & 1e10 & 5e5 \\
\end{array}
$$
Merge sort versus insertion sort

Merge sort takes $\mathcal{O}(n \log n)$. Insertion sort takes $\mathcal{O}(n^2)$. What does this mean concretely?

<table>
<thead>
<tr>
<th>$n$</th>
<th>$n^2$</th>
<th>$10^{12} n \log n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1e1</td>
<td>1e2</td>
<td>1e13</td>
</tr>
<tr>
<td>1e2</td>
<td>1e4</td>
<td>2e15</td>
</tr>
<tr>
<td>1e3</td>
<td>1e6</td>
<td>3e16</td>
</tr>
<tr>
<td>1e4</td>
<td>1e8</td>
<td>4e17</td>
</tr>
<tr>
<td>1e5</td>
<td>1e10</td>
<td>5e18</td>
</tr>
</tbody>
</table>
Merge sort versus insertion sort

Merge sort takes $O(n \log n)$. Insertion sort takes $O(n^2)$. What does this mean concretely?

<table>
<thead>
<tr>
<th>$n$</th>
<th>$n^2$</th>
<th>$10^{12}n \log n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1e1</td>
<td>1e2</td>
<td>1e13</td>
</tr>
<tr>
<td>1e2</td>
<td>1e4</td>
<td>2e15</td>
</tr>
<tr>
<td>1e3</td>
<td>1e6</td>
<td>3e16</td>
</tr>
<tr>
<td>1e4</td>
<td>1e8</td>
<td>4e17</td>
</tr>
<tr>
<td>1e5</td>
<td>1e10</td>
<td>5e18</td>
</tr>
<tr>
<td>1e13</td>
<td>1e26</td>
<td>1.3e26</td>
</tr>
</tbody>
</table>
Merge sort versus insertion sort

Merge sort takes $O(n \log n)$. Insertion sort takes $O(n^2)$. What does this mean concretely?

<table>
<thead>
<tr>
<th>$n$</th>
<th>$n^2$</th>
<th>$10^{12} n \log n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1e1</td>
<td>1e2</td>
<td>1e13</td>
</tr>
<tr>
<td>1e2</td>
<td>1e4</td>
<td>2e15</td>
</tr>
<tr>
<td>1e3</td>
<td>1e6</td>
<td>3e16</td>
</tr>
<tr>
<td>1e4</td>
<td>1e8</td>
<td>4e17</td>
</tr>
<tr>
<td>1e5</td>
<td>1e10</td>
<td>5e18</td>
</tr>
<tr>
<td>1e13</td>
<td>1e26</td>
<td>1.3e26</td>
</tr>
<tr>
<td>1e14</td>
<td>1e28</td>
<td>1.4e27</td>
</tr>
</tbody>
</table>
Formally

If $f$ is a function, then $\mathcal{O}(f)$ is the set of functions that “grow no faster than” $f$
Formally

If $f$ is a function, then $\mathcal{O}(f)$ is the set of functions that “grow no faster than” $f$.

“$g$ grows no faster than $f$” means there exist some $c$ and $m$ such that for all $n > m$, $g(n) \leq cf(n)$.
Formally

If $f$ is a function, then $\mathcal{O}(f)$ is the set of functions that “grow no faster than” $f$

“$g$ grows no faster than $f$” means there exist some $c$ and $m$ such that for all $n > m$, $g(n) \leq cf(n)$

Intuitively: on large enough input ($m$), $g$ grows no faster than $f$, up to getting a faster computer ($c$)
Another definition

\( f \lll g \) means \( f \in O(g) \) but \( g \notin O(f) \)
Big-O equalities

There are a bunch of rules we can apply to simplify complexity expressions:

- $O(f(n) + c) = O(f(n))$

if $g \ll f$
There are a bunch of rules we can apply to simplify complexity expressions:

- $O(f(n) + c) = O(f(n))$
  - In other words: $f(n) + c \in O(f(n))$
Big-O equalities

There are a bunch of rules we can apply to simplify complexity expressions:

- $O(f(n) + c) = O(f(n))$
  In other words: $f(n) + c \in O(f(n))$
- $O(cf(n)) = O(f(n))$
Big-O equalities

There are a bunch of rules we can apply to simplify complexity expressions:

- \( \mathcal{O}(f(n) + c) = \mathcal{O}(f(n)) \)
  In other words: \( f(n) + c \in \mathcal{O}(f(n)) \)

- \( \mathcal{O}(cf(n)) = \mathcal{O}(f(n)) \)

- \( \mathcal{O}(\log_k f(n)) = \mathcal{O}(\log_j f(n)) \)
Big-O equalities

There are a bunch of rules we can apply to simplify complexity expressions:

- $\mathcal{O}(f(n) + c) = \mathcal{O}(f(n))$  
  In other words: $f(n) + c \in \mathcal{O}(f(n))$

- $\mathcal{O}(cf(n)) = \mathcal{O}(f(n))$

- $\mathcal{O}(\log_k f(n)) = \mathcal{O}(\log_j f(n))$

- $\mathcal{O}(f(n) + g(n)) = \mathcal{O}(f(n))$ if $g \ll f$
if $j < k$, then

$1 \ll \log n$  constants are less than logs...

$\ll n^j$  are less than polynomials...

$\ll n^k \log n$  are less than poly-log...

$\ll j^n$  are less than exponentials...

$\ll k^n$  are less than higher-base exponentials.
Big-O inequalities

if \( j < k \), then

\[
1 \ll \log n \ll n^{j} \ll n^{k} \ll n^{\log n} \ll n^{\log\log n} \ll n^{\text{poly-log}} \ll n^{\text{exponential}} \ll n^{\text{higher-base exponential}}.
\]

constants are less than logs…
are less than polynomials…
Big-O inequalities

if $j < k$, then

1 $\ll \log n$ \hspace{1cm} \text{constants are less than logs…}
$\ll n^j$ \hspace{1cm} \text{are less than polynomials…}
$\ll n^k$ \hspace{1cm} \text{are less than higher-degree polynomials…}
if \( j < k \), then

\[
\begin{align*}
1 & \ll \log n \\
& \ll n^j \\
& \ll n^k \\
& \ll n^k \log n
\end{align*}
\]

constants are less than logs…

are less than polynomials…

are less than higher-degree polynomials…

are less than poly-log…
Big-O inequalities

if $j < k$, then

\[ 1 \ll \log n \]
\[ \ll n^j \quad \text{constants are less than logs…} \]
\[ \ll n^k \quad \text{are less than polynomials…} \]
\[ \ll n^k \log n \quad \text{are less than higher-degree polynomials…} \]
\[ \ll j^n \quad \text{are less than poly-log…} \]
\[ \ll j^n \quad \text{are less than exponentials…} \]
if $j < k$, then

\[
\begin{align*}
1 & \ll \log n & \text{constants are less than logs}\ldots \\
& \ll n^j & \text{are less than polynomials}\ldots \\
& \ll n^k & \text{are less than higher-degree polynomials}\ldots \\
& \ll n^k \log n & \text{are less than poly-log}\ldots \\
& \ll j^n & \text{are less than exponentials}\ldots \\
& \ll k^n & \text{are less than higher-base exponentials}.
\end{align*}
\]
Next time: Trees and Tree Walks