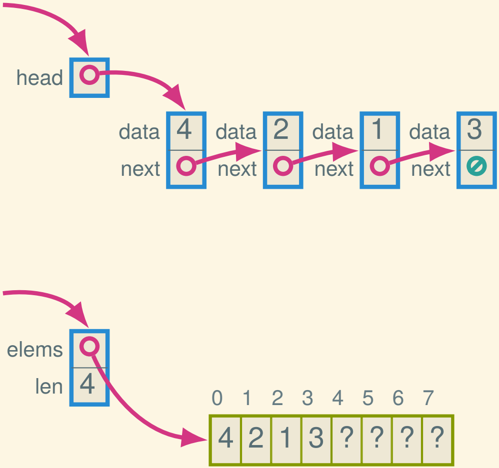
Asymptotic Complexity

EECS 214, Fall 2017

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

```
def list_nth(lst, n):
    def loop(i, link):
        if nil?(link): error('list_nth: out of bounds')
        elif i == 0:   return link.data
        else:          return loop(i - 1, link.next)
    loop(n, lst.head)
```

Getting the *n*th element

```
def list_nth(lst, n):
    def loop(i, link):
        if nil?(link): error('list_nth: out of bounds')
        elif i == 0:   return link.data
        else:          return loop(i - 1, link.next)
    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

```
def list_nth(lst, n):
    def loop(i, link):
        if nil?(link): error('list_nth: out of bounds')
        elif i == 0:   return link.data
        else:          return loop(i - 1, link.next)
    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 = node(val, lst.head)
```

Adding an element to the front

```
def list_push_front!(lst, val):
    lst.head = node(val, lst.head)

def array_push_front!(array, val):
    if array.len == len(array.elems):
        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.len = array.len + 1
    array.elems[0] = val
```

Adding an element to the front

```
def list_push_front!(lst, val):
    lst.head = node(val, lst.head)

def array_push_front!(array, val):
    if array.len == len(array.elems):
        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.len = array.len + 1
    array.elems[0] = val
```

`list_push_front!` is loop-free, whereas `array_push_front!` loops `array.len` times.

Breaking down `list-nth`

```
def list_nth(lst, n):
    let link = lst.head
    for i in n:
        link = link.next
    return link.data
```

$$T_{\texttt{list\_nth}}(n) = \quad (1)$$

(2)

(3)

Breaking down `list-nth`

```
def list_nth(lst, n):
    let link = lst.head
    for i in n:
        link = link.next
    return link.data
```

$$T_{\texttt{list\_nth}}(n) = T_{\text{get head}} + \tag{1}$$

(2)

(3)

Breaking down `list-nth`

```
def list_nth(lst, n):
    let link = lst.head
    for i in n:
        link = link.next
    return link.data
```

$$T_{\texttt{list\_nth}}(n) = T_{\text{get head}} + T_{\text{for setup}} + \tag{1}$$

(2)

(3)

Breaking down `list-nth`

```
def list_nth(lst, n):
    let link = lst.head
    for i in n:
        link = link.next
    return link.data
```

$$T_{\texttt{list\_nth}}(n) = T_{\text{get head}} + T_{\text{for setup}} + nT_{\text{assign link}} + \qquad (1)$$

(2)

(3)

Breaking down `list-nth`

```
def list_nth(lst, n):
    let link = lst.head
    for i in n:
        link = link.next
    return link.data
```

$$T_{\texttt{list\_nth}}(n) = T_{\text{get head}} + T_{\text{for setup}} + nT_{\text{assign link}} + \quad (1)$$
$$nT_{\text{get next}} + nT_{\text{for inc}} + \quad (2)$$
$$(3)$$

Breaking down `list-nth`

```
def list_nth(lst, n):
    let link = lst.head
    for i in n:
        link = link.next
    return link.data
```

$$T_{\texttt{list\_nth}}(n) = T_{\text{get head}} + T_{\text{for setup}} + nT_{\text{assign link}} + \quad (1)$$

$$nT_{\text{get next}} + nT_{\text{for inc}} + T_{\text{get data}} \quad (2)$$

$$(3)$$

Breaking down `list-nth`

```
def list_nth(lst, n):
    let link = lst.head
    for i in n:
        link = link.next
    return link.data
```

$$T_{\texttt{list\_nth}}(n) = T_{\text{get head}} + T_{\text{for setup}} + nT_{\text{assign link}} + \quad (1)$$
$$nT_{\text{get next}} + nT_{\text{for inc}} + T_{\text{get data}} \quad (2)$$
$$(3)$$

Let $c_1 = T_{\text{get head}} + T_{\text{for setup}} + T_{\text{get data}}$.

Breaking down `list-nth`

```
def list_nth(lst, n):
    let link = lst.head
    for i in n:
        link = link.next
    return link.data
```

$$T_{\texttt{list\_nth}}(n) = T_{\text{get head}} + T_{\text{for setup}} + nT_{\text{assign link}} + \tag{1}$$
$$nT_{\text{get next}} + nT_{\text{for inc}} + T_{\text{get data}} \tag{2}$$
$$\tag{3}$$

Let $c_1 = T_{\text{get head}} + T_{\text{for setup}} + T_{\text{get data}}$.

Let $c_2 = T_{\text{assign link}} + T_{\text{get next}} + T_{\text{for inc}}$.

Breaking down `list-nth`

```
def list_nth(lst, n):
    let link = lst.head
    for i in n:
        link = link.next
    return link.data
```

$$T_{\texttt{list\_nth}}(n) = T_{\text{get head}} + T_{\text{for setup}} + nT_{\text{assign link}} + \quad (1)$$

$$nT_{\text{get next}} + nT_{\text{for inc}} + T_{\text{get data}} \quad (2)$$

$$T_{\texttt{list\_nth}}(n) = c_1 + c_2 n \quad (3)$$

Let $c_1 = T_{\text{get head}} + T_{\text{for setup}} + T_{\text{get data}}$.

Let $c_2 = T_{\text{assign link}} + T_{\text{get next}} + T_{\text{for inc}}$.

Operation time comparison

	list	array
`nth`	$c_1 + c_2 n$	d_1
`push-front!`	e_1	$f_1 + f_2 n$

Operation time comparison

	list	array
`nth`	$c_1 + c_2 n$	d_1
`push-front!`	e_1	$f_1 + f_2 n$

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

Operation time comparison

	list	array
`nth`	$c_1 + c_2 n$	d_1
`push-front!`	e_1	$f_1 + f_2 n$

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

The same cannot be said when comparing $c_1 + c_2 n$ to $f_1 + f_2 n$.

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 $\mathcal{O}(n)$.

Another example: insertion sort

```
# : ListOf<Number> -> ListOf<Number>
def insertion_sort(lst):
    def insert(element, link):
        if node?(link) and link.data < element:
            node(link.data, insert(element, link.next))
        else: node(element, link)

    let acc = nil()
    let link = lst.head
    while node?(link):
        acc = insert(link.data, acc)
        link = link.next
    ll { head: acc }
```

Another example: insertion sort

```
# : ListOf<Number> -> ListOf<Number>
def insertion_sort(lst):
    def insert(element, link):
        if node?(link) and link.data < element:
            node(link.data, insert(element, link.next))
        else: node(element, link)

    let acc = nil()
    let link = lst.head
    while node?(link):
        acc = insert(link.data, acc)
        link = link.next
    ll { head: acc }
```

Nested loops of length n: $\mathcal{O}(n^2)$

Another example: merge sort helpers (1/2)

```
# : LinkOf<Number> LinkOf<Number> -> LinkOf<Number>
def merge(lnk1, lnk2):
    if node?(lnk1) and node?(lnk2):
        if lnk1.data < lnk2.data:
            node(lnk1.data, merge(lnk1.next, lnk2))
        else:
            node(lnk2.data, merge(lnk1, lnk2.next))
    elif nil?(lnk1): lnk2
    else: lnk1
```

Another example: merge sort helpers (1/2)

```
# : LinkOf<Number> LinkOf<Number> -> LinkOf<Number>
def merge(lnk1, lnk2):
    if node?(lnk1) and node?(lnk2):
        if lnk1.data < lnk2.data:
            node(lnk1.data, merge(lnk1.next, lnk2))
        else:
            node(lnk2.data, merge(lnk1, lnk2.next))
    elif nil?(lnk1): lnk2
    else: lnk1
```

`merge` is $\mathcal{O}(n)$.

Another example: merge sort helpers (2/2)

```
def odds(link):
    if node?(link): node(link.data, evens(link.next))
    else: nil()

def evens(link):
    if node?(link): odds(link.next)
    else: nil()
```

Another example: merge sort helpers (2/2)

```
def odds(link):
    if node?(link): node(link.data, evens(link.next))
    else: nil()

def evens(link):
    if node?(link): odds(link.next)
    else: nil()
```

odds and evens are both $\mathcal{O}(n)$.

Another example: merge sort

```
# : ListOf<Number> -> ListOf<Number>
def merge_sort(lst):
    def sort_links(link):
        if nil?(link) or nil?(link.next):
            link
        else:
            merge(sort_links(odds(link)),
                  sort_links(evens(link)))
    ll { head: sort_links(lst.head) }
```

Another example: merge sort

```
# : ListOf<Number> -> ListOf<Number>
def merge_sort(lst):
    def sort_links(link):
        if nil?(link) or nil?(link.next):
            link
        else:
            merge(sort_links(odds(link)),
                  sort_links(evens(link)))
    ll { head: sort_links(lst.head) }
```

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

Another example: merge sort

```
# : ListOf<Number> -> ListOf<Number>
def merge_sort(lst):
    def sort_links(link):
        if nil?(link) or nil?(link.next):
            link
        else:
            merge(sort_links(odds(link)),
                  sort_links(evens(link)))
    ll { head: sort_links(lst.head) }
```

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

n	n^2	$n \log n$
10	100	10
100	10,000	200
1,000	1,000,000	3,000
10,000	100,000,000	40,000
100,000	10,000,000,000	500,000

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?

n	n^2	$n \log n$
1E1	1E2	1E1
1E2	1E4	2E2
1E3	1E6	3E3
1E4	1E8	4E4
1E5	1E10	5E5

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?

n	n^2	$10^{12} n \log n$
1E1	1E2	1E1
1E2	1E4	2E2
1E3	1E6	3E3
1E4	1E8	4E4
1E5	1E10	5E5

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?

n	n^2	$10^{12} n \log n$
1E1	1E2	1E13
1E2	1E4	2E15
1E3	1E6	3E16
1E4	1E8	4E17
1E5	1E10	5E18

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?

n	n^2	$10^{12} n \log n$
1E1	1E2	1E13
1E2	1E4	2E15
1E3	1E6	3E16
1E4	1E8	4E17
1E5	1E10	5E18
1E13	1E26	1.3E26

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?

n	n^2	$10^{12} n \log n$
1E1	1E2	1E13
1E2	1E4	2E15
1E3	1E6	3E16
1E4	1E8	4E17
1E5	1E10	5E18
1E13	1E26	1.3E26
1E14	1E28	1.4E27

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 a change of constants (c)

Another definition

$f \lll g$ means $f \in \mathcal{O}(g)$ but $g \notin \mathcal{O}(f)$

Big-O equalities

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

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

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: $f(n) + c \in \mathcal{O}(f(n))$

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

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 $g \lll f$

Big-O inequalities

if $j < k$, then

$1 \lll \log n$ constants are less than logs… (4)

(9)

Big-O inequalities

if $j < k$, then

$$1 \lll \log n \quad \text{constants are less than logs...} \tag{4}$$

$$\lll n^j \quad \text{are less than polynomials...} \tag{5}$$

$$\tag{9}$$

Big-O inequalities

if $j < k$, then

$$1 \lll \log n \quad \text{constants are less than logs...} \quad (4)$$
$$\lll n^j \quad \text{are less than polynomials...} \quad (5)$$
$$\lll n^k \quad \text{are less than higher-degree polynomials...} \quad (6)$$

(9)

Big-O inequalities

if $j < k$, then

$$
\begin{aligned}
1 &\lll \log n && \text{constants are less than logs...} && (4)\\
&\lll n^j && \text{are less than polynomials...} && (5)\\
&\lll n^k && \text{are less than higher-degree polynomials...} && (6)\\
&\lll n^k \log n && \text{are less than poly-log...} && (7)\\
& && && (9)
\end{aligned}
$$

Big-O inequalities

if $j < k$, then

$$\begin{aligned}
1 &\lll \log n && \text{constants are less than logs...} && (4)\\
&\lll n^j && \text{are less than polynomials...} && (5)\\
&\lll n^k && \text{are less than higher-degree polynomials...} && (6)\\
&\lll n^k \log n && \text{are less than poly-log...} && (7)\\
&\lll j^n && \text{are less than exponentials...} && (8)\\
& && && (9)
\end{aligned}$$

Big-O inequalities

if $j < k$, then

$$1 \lll \log n \qquad \text{constants are less than logs...} \qquad (4)$$

$$\lll n^j \qquad \text{are less than polynomials...} \qquad (5)$$

$$\lll n^k \qquad \text{are less than higher-degree polynomials...} \qquad (6)$$

$$\lll n^k \log n \qquad \text{are less than poly-log...} \qquad (7)$$

$$\lll j^n \qquad \text{are less than exponentials...} \qquad (8)$$

$$\lll k^n \qquad \text{are less than higher-base exponentials} \qquad (9)$$