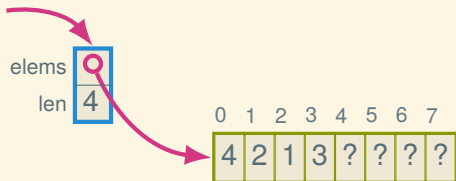
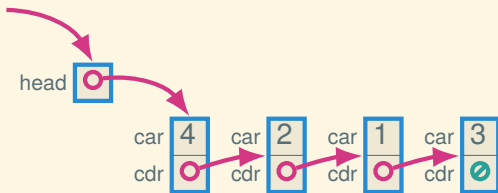


Asymptotic Complexity

EECS 214, Fall 2018

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.car  
        else: return loop(i - 1, link.cdr)  
    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.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

```
def list_nth(lst, n):
    def loop(i, link):
        if nil?(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

```
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.len = array.len + 1  
    array.elems[0] = val
```


Adding an element to the front

```
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.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.cdr  
    return link.car
```

$$T_{\text{list_nth}}(n) = \tag{1}$$

(2)

(3)

Breaking down list-nth

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

$$T_{\text{list_nth}}(n) = T_{\text{get head}} + \quad (1)$$

(2)

(3)

Breaking down list-nth

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

$$T_{\text{list_nth}}(n) = T_{\text{get head}} + T_{\text{for setup}} + \quad (1)$$

(2)

(3)

Breaking down list-nth

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

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

(2)

(3)

Breaking down list-nth

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

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

$$nT_{\text{get cdr}} + nT_{\text{for inc}} + \quad (2)$$

$$(3)$$

Breaking down list-nth

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

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

$$nT_{\text{get cdr}} + nT_{\text{for inc}} + T_{\text{get car}} \quad (2)$$

$$(3)$$

Breaking down list-nth

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

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

$$nT_{\text{get cdr}} + nT_{\text{for inc}} + T_{\text{get car}} \quad (2)$$

$$(3)$$

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

Breaking down list-nth

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

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

$$nT_{\text{get cdr}} + nT_{\text{for inc}} + T_{\text{get car}} \quad (2)$$

$$(3)$$

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

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

Breaking down list-nth

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

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

$$nT_{\text{get cdr}} + nT_{\text{for inc}} + T_{\text{get car}} \quad (2)$$

$$T_{\text{list_nth}}(n) = c_1 + c_2n \quad (3)$$

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

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

Operation time comparison

	list	array
nth	$c_1 + c_2n$	d_1
push_front	e_1	$f_1 + f_2n$

Operation time comparison

	list	array
nth	$c_1 + c_2n$	d_1
push_front	e_1	$f_1 + f_2n$

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 .

Operation time comparison

	list	array
nth	$c_1 + c_2n$	d_1
push_front	e_1	$f_1 + f_2n$

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 .

The same cannot be said when comparing $c_1 + c_2n$ to $f_1 + f_2n$.

Complexity classes

There's a sense in which $c_1 + c_2n$ and $f_1 + f_2n$ are similar.

Complexity classes

There's a sense in which $c_1 + c_2n$ and $f_1 + f_2n$ are similar.

We call this sense $\mathcal{O}(n)$.

Another example: insertion sort

```
# : LinkOf[Number] -> LinkOf[Number]
def insertion_sort(lst):
  def insert(element, link):
    if cons?(link) and link.car < element:
      cons(link.car, insert(element, link.cdr))
    else: cons(element, link)

  let result = nil()
  let link = lst.head
  while cons?(link):
    result = insert(link.car, result)
    link = link.cdr
  result
```


Another example: insertion sort

```
# : LinkOf[Number] -> LinkOf[Number]
def insertion_sort(lst):
  def insert(element, link):
    if cons?(link) and link.car < element:
      cons(link.car, insert(element, link.cdr))
    else: cons(element, link)

  let result = nil()
  let link = lst.head
  while cons?(link):
    result = insert(link.car, result)
    link = link.cdr
  result
```

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 cons?(lnk1) and cons?(lnk2):
    if lnk1.car < lnk2.car:
      cons(lnk1.car, merge(lnk1.cdr, lnk2))
    else:
      cons(lnk2.car, merge(lnk1, lnk2.cdr))
  elif nil?(lnk1): lnk2
  else: lnk1
```

Another example: merge sort helpers (1/2)

```
# : LinkOf[Number] LinkOf[Number] -> LinkOf[Number]
def merge(lnk1, lnk2):
  if cons?(lnk1) and cons?(lnk2):
    if lnk1.car < lnk2.car:
      cons(lnk1.car, merge(lnk1.cdr, lnk2))
    else:
      cons(lnk2.car, merge(lnk1, lnk2.cdr))
  elif nil?(lnk1): lnk2
  else: lnk1
```

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

Another example: merge sort helpers (2/2)

```
def odds(link):  
  if cons?(link): cons(link.car, evens(link.cdr))  
  else: nil()  
  
def evens(link):  
  if cons?(link): odds(link.cdr)  
  else: nil()
```

Another example: merge sort helpers (2/2)

```
def odds(link):  
  if cons?(link): cons(link.car, evens(link.cdr))  
  else: nil()
```

```
def evens(link):  
  if cons?(link): odds(link.cdr)  
  else: nil()
```

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

Another example: merge sort

```
# : LinkOf[Number] -> LinkOf[Number]
def merge_sort(lst):
  if nil?(link) or nil?(link.cdr):
    link
  else:
    merge(merge_sort(odds(link)),
          merge_sort(evens(link)))
```

Another example: merge sort

```
# : LinkOf[Number] -> LinkOf[Number]
def merge_sort(lst):
  if nil?(link) or nil?(link.cdr):
    link
  else:
    merge(merge_sort(odds(link)),
          merge_sort(evens(link)))
```

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

Another example: merge sort

```
# : LinkOf[Number] -> LinkOf[Number]
def merge_sort(lst):
  if nil?(link) or nil?(link.cdr):
    link
  else:
    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 $\mathcal{O}(n \log n)$. 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

$$1 \lll \log n$$

$$\lll n^j$$

constants are less than logs... (4)

are less than polynomials... (5)

(9)

Big-O inequalities

if $j < k$, then

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

$\lll n^j$ are less than polynomials... (5)

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

(9)

Big-O inequalities

if $j < k$, then

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

$\lll n^j$ are less than polynomials... (5)

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

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

(9)

Big-O inequalities

if $j < k$, then

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

$\lll n^j$ are less than polynomials... (5)

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

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

$\lll j^n$ are less than exponentials... (8)

(9)

Big-O inequalities

if $j < k$, then

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

$\lll n^j$ are less than polynomials... (5)

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

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

$\lll j^n$ are less than exponentials... (8)

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