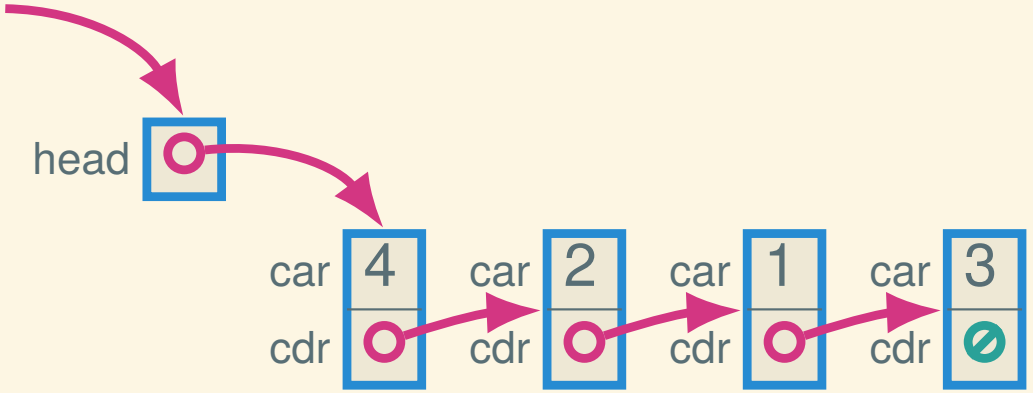
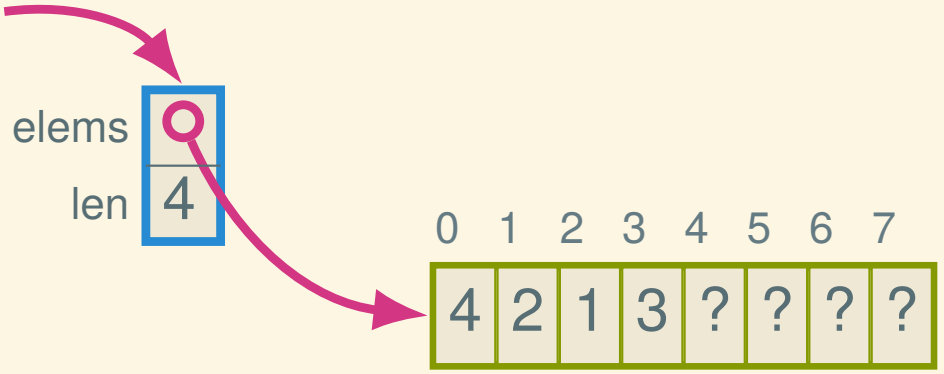
Asymptotic Complexity

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

```
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)
```

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

```
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

```
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

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

(T stands for "time")

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

Breaking down `list-nth`

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

(T stands for "time")

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

Breaking down `list_nth`

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

(T stands for "time")

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

Breaking down `list-nth`

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

(T stands for "time")

$$T_{\texttt{list\_nth}}(n) = T_{\text{get head}} + T_{\text{loop setup}} + nT_{\text{get cdr}} +$$

Breaking down `list_nth`

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

(T stands for "time")

$$T_{\texttt{list\_nth}}(n) = T_{\text{get head}} + T_{\text{loop setup}} + nT_{\text{get cdr}} + nT_{\text{assign link}} + nT_{\text{loop inc}} +$$

Breaking down `list_nth`

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

(T stands for "time")

$$T_{\texttt{list\_nth}}(n) = T_{\text{get head}} + T_{\text{loop setup}} + nT_{\text{get cdr}} + nT_{\text{assign link}} + nT_{\text{loop inc}} + T_{\text{get car}}$$

Breaking down `list_nth`

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

(T stands for "time")

$$T_{\texttt{list\_nth}}(n) = T_{\text{get head}} + T_{\text{loop setup}} + nT_{\text{get cdr}} + nT_{\text{assign link}} + nT_{\text{loop inc}} + T_{\text{get car}}$$

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

Breaking down `list_nth`

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

(T stands for "time")

$$T_{\texttt{list\_nth}}(n) = T_{\text{get head}} + T_{\text{loop setup}} + nT_{\text{get cdr}} + nT_{\text{assign link}} + nT_{\text{loop inc}} + T_{\text{get car}}$$

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`

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

(T stands for "time")

$$T_{\texttt{list\_nth}}(n) = T_{\text{get head}} + T_{\text{loop setup}} + nT_{\text{get cdr}} + nT_{\text{assign link}} + nT_{\text{loop inc}} + T_{\text{get car}}$$

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

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

	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

```
# : 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))

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

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))

    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)

```
# : 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
```

Another example: merge sort helpers (1/2)

```
# : 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 $\mathcal{O}(|\texttt{lnk1}| + |\texttt{lnk2}|) = \mathcal{O}(n)$.

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
```

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
```

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

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)))
```

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

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 $\mathcal{O}(n)$. 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 getting a faster computer (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…

Big-O inequalities

if $j < k$, then

$$1 \lll \log n \qquad \text{constants are less than logs…}$$
$$\lll n^j \qquad \text{are less than polynomials…}$$

Big-O inequalities

if $j < k$, then

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

$$\lll n^j$$ are less than polynomials…

$$\lll n^k$$ are less than higher-degree polynomials…

Big-O inequalities

if $j < k$, then

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

$$\lll n^j$$ are less than polynomials…

$$\lll n^k$$ are less than higher-degree polynomials…

$$\lll n^k \log n$$ are less than poly-log…

Big-O inequalities

if $j < k$, then

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

$$\lll n^j$$ are less than polynomials…

$$\lll n^k$$ are less than higher-degree polynomials…

$$\lll n^k \log n$$ are less than poly-log…

$$\lll j^n$$ are less than exponentials…

Big-O inequalities

if $j < k$, then

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

$$\lll n^j$$ are less than polynomials…

$$\lll n^k$$ are less than higher-degree polynomials…

$$\lll n^k \log n$$ are less than poly-log…

$$\lll j^n$$ are less than exponentials…

$$\lll k^n$$ are less than higher-base exponentials.

Next time: Trees and Tree Walks