COMP2521 19T0
Week 6, Tuesday: Order! Order! (I)

Jashank Jeremy

jashank.jeremy@unsw.edu.au

basic sorting algorithms
more sorting algorithms

Administrivia

MYEXPERIENCE now open!
`myexperience.unsw.edu.au`

PRAC EXAM #1 results look pretty good
a majority of people passed the exam!
no problem required >10 LoC;
if you just threw code at the wall,
consider a different strategy next time.

ASSIGNMENT 2 part 1 is underway!
views due **20 Jan 2019**, no extensions.
view dryruns out now — how does your code do?
hunt spec to be released during week07tue lecture

Sorting

Sorting: The Problem

Sorting

... arranging a collection of items in order,
... based on some property of an item (a 'key'),
... using an ordering relation on that property.

Sorting: The Problem

Why? What? Where?

Why are we interested?

- speeds up subsequent searches;
- arranges data in useful ways (human- or otherwise)
  ... *e.g.*, a list of students in a tutorial
- provides useful intermediate for other algorithms
  ... *e.g.*, duplicate detection/removal; DBMS operations

What contexts?

- arrays, linked lists (in-memory, internal)
- files (external, on-disk)
- ... distributed across a network (map/reduce)

We'll focus on sorting arrays (and lists)

Sorting: The Problem

(More) Formally

Pre-conditions:
array `a[N]` of `Items`
`lo`, `hi` are valid indices on `a`
(roughly, $0 \le \mathtt{lo} < \mathtt{hi} \le \mathsf{N} - 1$)

Post-conditions:
array `a'[lo..hi]` contains same values
$\mathsf{a}'[\mathtt{lo}] \le \mathsf{a}'[\mathtt{lo} + 1] \le \mathsf{a}'[\mathtt{lo} + 2] \le \cdots \le \mathsf{a}'[\mathtt{hi}]$

Sorting: The Problem

Properties: Stability, Adaptive, In-Place

Properties: stable sorts

let $x =$ `a[i]`, $y =$ `a[j]`, where KEY$(x) \equiv$ KEY(y)

let the 'precedes' relation be that index $i \leq j$.

if x 'precedes' y in a, then x 'precedes' y in a′

Properties: adaptive sorts

where the algorithm's behaviour or performance
is affected by the input data —
that best/average/worst case performance differs
... and can take advantage of *existing order*

Properties: in-place sorts

sort data within original structure,
using only a constant additional amount of space

A Concrete Framework

```c
// we deal with generic `Item's
typedef int Item;

// abstractions to hide details of items
#define key(A) (A)
#define less(A,B) (key(A) < key(B))
#define eq(A,B) (key(A) == key(B))
#define swap(A,B) { Item t; t = A; A = B; B = t; }
#define cas(A,B) { if (less (A, B)) swap (A, B); }
// cas == Compare And Swap, often hardware assisted

/// Sort a slice of an array of Items.
void sort (Item a[], int lo, int hi);

/// Check for sortedness (to validate functions).
bool sorted_p (Item a[], int lo, int hi);
```

A Concrete Framework

This framework can be adapted by...
defining a different data structure for `Item`;
defining a method for extracting sort keys;
defining a different ordering (`less`);
defining a different `swap` method for different `Item`

```c
typedef struct { char *name; char *course; } Item;
#define key(A) (A.name)
#define less(A, B) (strcmp(key(A), key(B)) < 0)
#define swap(A,B) { Item t; t = A; A = B; B = t; }
// ... works because struct assignment works in C
```

Complexity of Sorting Algorithms

In analysing sorting algorithms:

- N: the number of items ($\texttt{hi} - \texttt{lo} + 1$)
- C: the number of comparisons between items
- S: the number of times items are swapped

(We usually aim to minimise C and S.)

Cases to consider for input order:

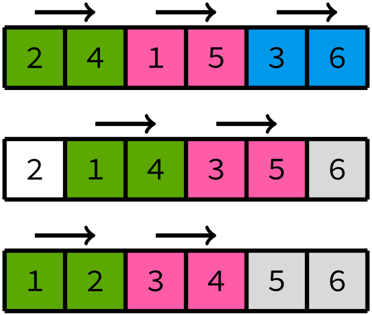
- random order: `Items` in `a[lo..hi]` have no ordering
- sorted-ascending order: $\texttt{a}[\texttt{lo}] \leq \texttt{a}[\texttt{lo} + 1] \leq \cdots \leq \texttt{a}[\texttt{hi}]$
- sorted-descending order: $\texttt{a}[\texttt{lo}] \geq \texttt{a}[\texttt{lo} + 1] \geq \cdots \geq \texttt{a}[\texttt{hi}]$

Three Simple Sorts

- Bubble Sort (oblivious and early-exit)
- Selection Sort
- Insertion Sort

Bubble Sort

Values ‘bubble up’ the array.



Bubble Sort

C Implementation — Oblivious

```c
void sort_bubble (Item items[], size_t lo, size_t hi)
{
    for (size_t i = hi; i > lo; i--)
        for (size_t j = lo + 1; j <= i; j++)
            if (less (items[j], items[j - 1]))
                swap_idx (items, j, j - 1);
}
```

Bubble Sort

C Implementation — Analysis

- Outer loop (C_0) $\quad \Rightarrow N$
  ```
  for (size_t i = n - 1; i > 0; i--)
  ```
- Inner loop (C_1) $\quad \Rightarrow N + (N-1) + (N-2) + \cdots + 2 = (N^2 + N)/2 - 1$
  ```
  for (size_t j = 1; j <= i; j++)
  ```
- Comparisons (C_2) $\quad \Rightarrow N + (N-1) + (N-2) + \cdots + 1 + 0 = (N^2 - N)/2$
- Swaps (C_3) $\quad \Rightarrow N + (N-1) + (N-2) + \cdots + 1 + 0 = (N^2 - N)/2$
  (assuming the worst case: we *always* have to swap)

$$T(n) = NC_0 + \left(\frac{N^2 + N}{2} - 1 \right) C_1 + \frac{N^2 - N}{2} C_2 + \frac{N^2 - N}{2} C_3$$

$$\Rightarrow \; O(N^2)$$

Bubble Sort

Summary

How many steps does it take
to sort a collection of N elements?

For the ith iteration, we have
$N - i$ comparisons and
best 0, worst $N - i$ swaps
(depending on sortedness.)

Bubble sort is $O(n^2)$.
Stable, in-place, non-adaptive.

Improving Bubble Sort

‘oblivious’ bubble-sort continues, even if the list is sorted
so what’s a better stopping-case than ‘we ran out of array’?

if we complete a whole pass without swaps, we’re ordered!
this is bubble sort with early exit, or adaptive bubble sort

Adaptive Bubble Sort

C Implementation — Adaptive

```
void sort_bubble_ee (Item items[], size_t lo, size_t hi)
{
    bool no_swaps = false;
    for (size_t i = hi; i > lo && !no_swaps; i--) {
        no_swaps = true;
        for (size_t j = lo + 1; j <= i; j++)
            if (less (items[j], items[j - 1])) {
                swap_idx (items, j, j - 1);
                no_swaps = false;
            }
    }
}
```

Adaptive Bubble Sort

Analysis; Summary

How many steps does it take
to sort a collection of N elements?

Each traversal does N comparisons.
Best case: exit after one iteration
(if the collection is already sorted.)
Worst case: N traversals still necessary.

$$T_{\text{worst}}(N) = N - 1 + N - 2 + \cdots + 1 \approx N^2$$
$$T_{\text{best}}(N) = N$$

Bubble-sort with early exit is *still* $O(N^2)$.
Stable, in-place, adaptive (!).

Selection Sort

Select the smallest element.
Swap it with the first position.

Select the next smallest element.
Swap it with the second position.

... continue until sorted!

Selection Sort

4	1	7	3	8	6	5	2
1	4	7	3	8	6	5	2
1	2	7	3	8	6	5	4
1	2	3	7	8	6	5	4
1	2	3	4	8	6	5	7
1	2	3	4	5	6	8	7
1	2	3	4	5	6	8	7
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8

Selection Sort

C Implementation

```
void sort_selection (Item items[], size_t lo, size_t hi)
{
    for (size_t i = lo; i < hi; i++) {
        size_t low = i;
        for (size_t j = i + 1; j <= hi; j++)
            if (less (items[j], items[low]))
                low = j;
        swap_idx (items, i, low);
    }
}
```

Selection Sort

Analysis; Summary

How many steps does it take
to sort a collection of N elements?

... picking the minimum of a sequence of N elements: N steps.
... inserting at the right place: 1.

$$T\left(N\right) = N + (N-1) + (N-2) + \cdots + 1 = \frac{1}{2}N(N+1)$$

Selection sort is $O\left(N^2\right)$.
Unstable, in-place, oblivious.s

Insertion Sort

Take the first element, insert into the first position.
This starts our ‘sorted sublist’.

Take the next element.
Insert it into the sorted sublist
in the right spot!

Repeat until sorted!

Insertion Sort

4	1	7	3	8	6	5	2
1	4	7	3	8	6	5	2
1	4	7	3	8	6	5	2
1	3	4	7	8	6	5	2
1	3	4	7	8	6	5	2
1	3	4	6	7	8	5	2
1	3	4	5	6	7	8	2
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8

Insertion Sort

C Implementation

```
void sort_insertion (Item items[], size_t lo, size_t hi)
{
    for (size_t i = lo + 1; i <= hi; i++) {
        Item item = items[i];
        size_t j = i;
        for (/* j */; j > lo; j--) {
            if (! less (item, items[j - 1])) break;
            items[j] = items[j - 1];
        }
        items[j] = item;
    }
}
```

Insertion Sort

Analysis; Summary

How many steps does it take
to sort a collection of N elements?

For every element (of N elements):
1 step to pick an element;
insert into a $N' \leq N$ sequence: up to N steps.

$$T_{\text{worst}}\left(N\right) = 1 + 2 + \cdots + N = \frac{N}{2}(N+1)$$
$$T_{\text{best}}\left(N\right) = 1 + 1 + \cdots + 1 = N$$

Insertion sort is $O\left(N^2\right)$.
Stable, in-place, adaptive.

Shell Sort

One Small Swap for a Sort ...

Bubble- and Insertion-Sort
really only consider *adjacent* elements.

If we make longer-distance exchanges,
can we be more efficient?

What if we consider elements that are some distance apart?
... sort sublists of mod-h indices,
for decreasing h until $h = 1$?

Shell Sort

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
unsorted	4	1	7	3	8	6	5	2
$h = 3$ passes	3			4			5	
		1			2			8
			6			7		
3-sorted	3	1	6	4	2	7	5	8
$h = 2$ passes	2		3		5		6	
		1		4		7		8
2-sorted	2	1	3	4	5	7	6	8
$h = 1$ pass	1	2	3	4	5	6	7	8

Shell Sort

C Implementation

```
void sort_shell (Item items[], size_t lo, size_t hi)
{
    size_t h;
    for (h = 1; h <= (n - 1) / 9; h = (3 * h) + 1);

    for (/* h */; h > 0; h /= 3) {
        // when `h' = 1, this is an insertion sort.
        for (size_t i = h; i < n; i++) {
            Item item = items[i];
            size_t j = i;
            for (/* j */; j >= h && item < items[j - h]; j -= h)
                items[j] = items[j - h];
            items[j] = item;
        }
    }
}
```

Shell Sort

... One Giant Leap for Complexity?

The exact complexity-class depends on the h-sequence.
Probably safe to assume that $O\left(\leq n^2\right)$,
because otherwise what's the point?
Lots of h-value sequences are $O\left(n^{\frac{3}{2}}\right)$.

No 'general' analysis exists.

Shell Sort is $O\left(\leq n^2\right)$.
It is unstable, adaptive, in-place.

Aside: Sorting Linked Lists

Bubble traverse list; if curr > next, swap.

Selection delete selected element, insert at head of sorted list.

Insertion delete first element, do order-preserving insertion.

Shell *(screaming)*