Elementary Sorting Algorithms

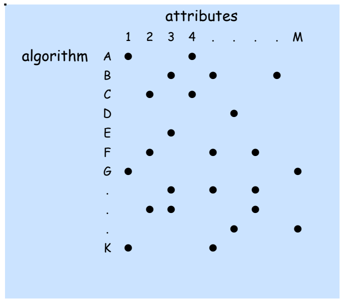
Insertion sort
Selection sort
Bubble sort
Shell sort

Why study sorting algorithms?

Q: Isn't the system sort good enough?

A: Maybe.

- Is your file randomly ordered?
- Need guaranteed performance?
- Multiple key types?
- Multiple keys?
- Keys all distinct?
- Linked list or array?
- Large or small records?



many more combinations of attributes than algorithms

A: An elementary method may be the method of choice

A: Use well-understood topic to address basic issues.

A: Interesting open research problems still arise.

Basic terms

Ex: `struct { char name[20]; int class; char grade; long id; char addr[30]; }`

file →
record →
key →

Fox	1	A	243-456-9091	101 Brown
Quilici	1	C	343-987-5642	32 McCosh
Chen	2	A	884-232-5341	11 Dickinson
Furia	3	A	766-093-9873	22 Brown
Kanaga	3	B	898-122-9643	343 Forbes
Andrews	3	A	874-088-1212	121 Whitman
Rohde	3	A	232-343-5555	115 Holder
Battle	4	C	991-878-4944	308 Blair
Aaron	4	A	664-480-0023	097 Little
Gazsi	4	B	665-303-0266	113 Walker

SORT: Rearrange records such that keys are in order

Aaron	4	A	664-480-0023	097 Little
Andrews	3	A	874-088-1212	121 Whitman
Battle	4	C	991-878-4944	308 Blair
Chen	2	A	884-232-5341	11 Dickinson
Fox	1	A	243-456-9091	101 Brown
Furia	3	A	766-093-9873	22 Brown
Gazsi	4	B	665-303-0266	113 Walker
Kanaga	3	B	898-122-9643	343 Forbes
Rohde	3	A	232-343-5555	115 Holder
Quilici	1	C	343-987-5642	32 McCosh

Stability

A STABLE sort preserves relative order of records with equal keys

Ex: sort file
on 1st key

Aaron	4	A	664-480-0023	097 Little
Andrews	3	A	874-088-1212	121 Whitman
Battle	4	C	991-878-4944	308 Blair
Chen	2	A	884-232-5341	11 Dickinson
Fox	1	A	243-456-9091	101 Brown
Furia	3	A	766-093-9873	22 Brown
Gazsi	4	B	665-303-0266	113 Walker
Kanaga	3	B	898-122-9643	343 Forbes
Rohde	3	A	232-343-5555	115 Holder
Quilici	1	C	343-987-5642	32 McCosh

then sort file
on 2nd key

@#%&@!!
records with
key value 3
not in order
on first key →

Fox	1	A	243-456-9091	101 Brown
Quilici		C	343-987-5642	32 McCosh
Chen		A	884-232-5341	11 Dickinson
Kanaga		B	898-122-9643	343 Forbes
Andrews		A	874-088-1212	121 Whitman
Furia		A	766-093-9873	22 Brown
Rohde		A	232-343-5555	115 Holder
Battle		C	991-878-4944	308 Blair
Gazsi		B	665-303-0266	113 Walker
Aaron		A	664-480-0023	097 Little

Annoying fact: many otherwise useful algorithms are not stable

Abstract compare and exchange

GOAL: Specify key such that sort code is reusable

- use records of type `Item`
- use `less(Item, Item)` to compare two records
- use `exch(Item, Item)` to exchange two records

Ex: array of integers

```
typedef int Item;
#define less(A, B) (A < B)
#define exch(A, B) { Item t = A; A = B; B = t; }
```

Ex: record with associated information

```
typedef struct { ... } Item;
#define less(A, B) (strcmp(A.name, B.name) < 0)
```

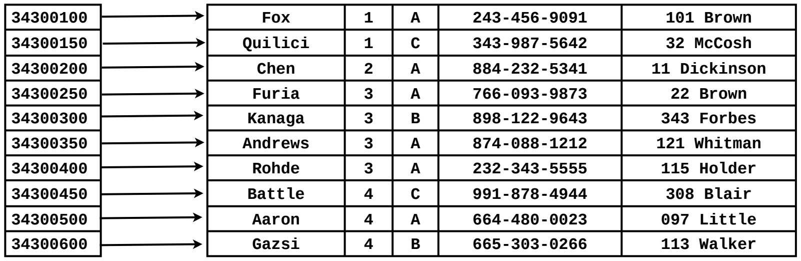
or

```
#define less(A, B) (A.id < B.id)
```

More general (and more expensive) alternative: `Item` ADT

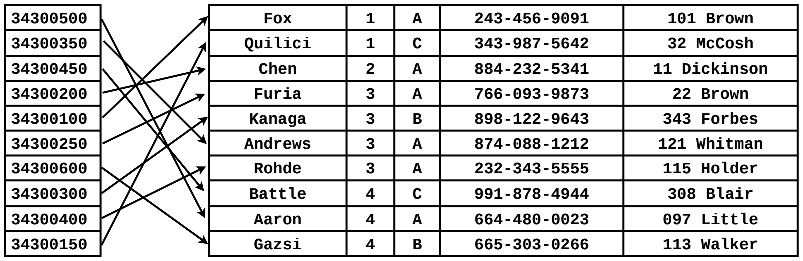
Pointer sort

Maintain pointers to records



34300100	Fox	1	A	243-456-9091	101 Brown
34300150	Quilici	1	C	343-987-5642	32 McCosh
34300200	Chen	2	A	884-232-5341	11 Dickinson
34300250	Furia	3	A	766-093-9873	22 Brown
34300300	Kanaga	3	B	898-122-9643	343 Forbes
34300350	Andrews	3	A	874-088-1212	121 Whitman
34300400	Rohde	3	A	232-343-5555	115 Holder
34300450	Battle	4	C	991-878-4944	308 Blair
34300500	Aaron	4	A	664-480-0023	097 Little
34300600	Gazsi	4	B	665-303-0266	113 Walker

Rearrange pointers, leave records untouched



34300500	Fox	1	A	243-456-9091	101 Brown
34300350	Quilici	1	C	343-987-5642	32 McCosh
34300450	Chen	2	A	884-232-5341	11 Dickinson
34300200	Furia	3	A	766-093-9873	22 Brown
34300100	Kanaga	3	B	898-122-9643	343 Forbes
34300250	Andrews	3	A	874-088-1212	121 Whitman
34300600	Rohde	3	A	232-343-5555	115 Holder
34300300	Battle	4	C	991-878-4944	308 Blair
34300400	Aaron	4	A	664-480-0023	097 Little
34300150	Gazsi	4	B	665-303-0266	113 Walker

Pointer sort implementation

GOAL: Use reusable sort code

- use records of type `Item`
- use `less(Item, Item)` to compare two records
- use `exch(Item, Item)` to exchange two records
-

Ex: record with associated information

```
typedef struct { ... } Record;
typedef Record* Item;
#define less(A, B) (A->key < B->key)
#define exch(A, B) { Item t = A; A = B; B = t; }
```

Avoids excessive moves of large records

Validates rule of thumb: cost of compare similar to cost of exchange

Insertion sort

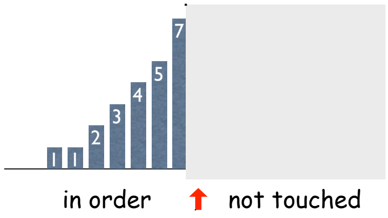
↑ scans from left to right

Invariant:

- elements to right of ↑ are not touched

Implication:

- keep elements to left of ↑ in sorted order



Insertion sort inner loop: maintaining the invariant

Save current element

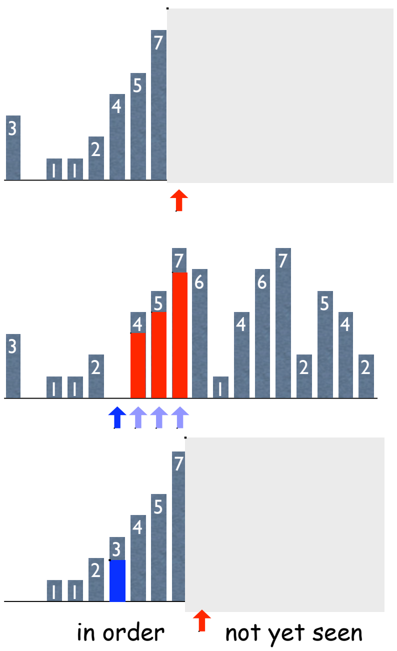
```
v = a[i];
```

Shift right all larger elements on left

```
j = i;
while (j > l && less(v, a[j-1]))
  { a[j] = a[j-1]; j--; }
```

Store v in vacant spot

```
a[j] = v;
```



Insertion sort example

A	S	O	R	T	I	N	G	E	X	A	M	P	L	E
A	S	O	R	T	I	N	G	E	X	A	M	P	L	E
A	O	S	R	T	I	N	G	E	X	A	M	P	L	E
A	O	R	S	T	I	N	G	E	X	A	M	P	L	E
A	O	R	S	T	I	N	G	E	X	A	M	P	L	E
A	I	O	R	S	T	N	G	E	X	A	M	P	L	E
A	I	N	O	R	S	T	G	E	X	A	M	P	L	E
A	G	I	N	O	R	S	T	E	X	A	M	P	L	E
A	E	G	I	N	O	R	S	T	X	A	M	P	L	E
A	E	G	I	N	O	R	S	T	X	A	M	P	L	E
A	A	E	G	I	N	O	R	S	T	X	M	P	L	E
A	A	E	G	I	M	N	O	R	S	T	X	P	L	E
A	A	E	G	I	M	N	O	P	R	S	T	X	L	E
A	A	E	G	I	L	M	N	O	P	R	S	T	X	E
A	A	E	E	G	I	L	M	N	O	P	R	S	T	X

■ = 1 comparison and 1 assignment to memory

Insertion sort code

```
                                               To sort a[l]...a[r]
  void insertion(Item a[], int l, int r)
    { int i;
      for (i = l+1; i <= r; i++)              scan left to right
        { Item v = a[i];                          save item
          int j = i;                               shift
          while (j > l && less(v, a[j-1]))         larger
            { a[j] = a[j-1]; j--; }                items
          a[j] = v;                              insert item
        }
    }
```

Selection sort

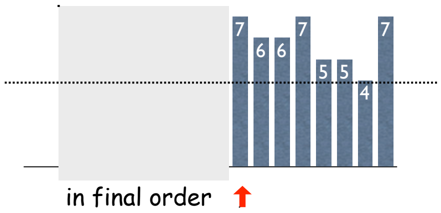
↑ scans from left to right

Invariant:

- elements to left of ↑ are not touched

Implications:

- keep elements to left of ↑ are in final order
- no element to left of ↑ is larger than any element to its right



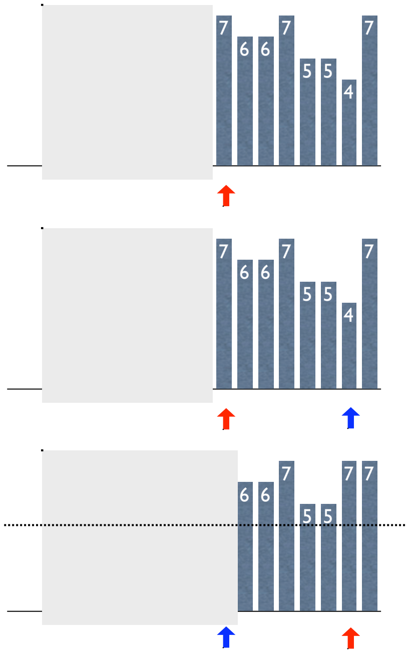
Selection sort inner loop: maintaining the invariant

Select minimum

```
int j, min = i;
for (j = i+1; j < r; j++)
  if (less(a[j], a[min])) min = j;
```

Exchange into position

```
exch(a[i], a[min]);
```



Selection sort example

A	S	O	R	T	I	N	G	E	X	A	M	P	L	E
A	A	O	R	T	I	N	G	E	X	S	M	P	L	E
A	A	E	R	T	I	N	G	O	X	S	M	P	L	E
A	A	E	E	T	I	N	G	O	X	S	M	P	L	R
A	A	E	E	G	I	N	T	O	X	S	M	P	L	R
A	A	E	E	G	I	N	T	O	X	S	M	P	L	R
A	A	E	E	G	I	L	T	O	X	S	M	P	N	R
A	A	E	E	G	I	L	M	O	X	S	T	P	N	R
A	A	E	E	G	I	L	M	N	X	S	T	P	O	R
A	A	E	E	G	I	L	M	N	O	S	T	P	X	R
A	A	E	E	G	I	L	M	N	O	P	T	S	X	R
A	A	E	E	G	I	L	M	N	O	P	R	S	X	T
A	A	E	E	G	I	L	M	N	O	P	R	S	X	T
A	A	E	E	G	I	L	M	N	O	P	R	S	T	X
A	A	E	E	G	I	L	M	N	O	P	R	S	T	X

🟥 = 1 assignment to memory

🟦 = 1 comparison

Selection sort code

```
                                             To sort a[l]...a[r]
void selection(Item a[], int l, int r)
  { int i;                                   scan left to right
    for (i = l; i < r; i++)
      { int j, min = i;                        select
        for (j = i+1; j < r; j++)              minimum
          if (less(a[j], a[min])) min = j;
        exch(a[i], a[min]);                  exchange into place
      }
  }
```

Bubble sort

↑ scans from left to right

Algorithm:

- compare-exchange item at ↑ with item on its right

Implications:

- first pass puts max element into position
- like selection sort, but with more data movement

Bubble sort example

```
A S O R T I N G E X A M P L E
A O R S I N G E T A M P L E X
A O R I N G E S A M P L E T X
A O I N G E R A M P L E S T X
A I N G E O A M P L E R S T X
A I G E N A M O L E P R S T X
A G E I A M N L E O P R S T X
A E G A I M L E N O P R S T X
A E A G I L E M N O P R S T X
A A E G I E L M N O P R S T X
A A E G E I L M N O P R S T X
A A E E G I L M N O P R S T X
A A E E G I L M N O P R S T X
A A E E G I L M N O P R S T X
A A E E G I L M N O P R S T X
```

🟪 = 1 comparison and 2 assignments to memory

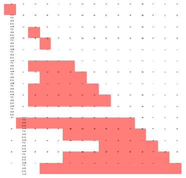
Performance for randomly ordered files

INSERTION

each element moves halfway back

(1 + 2 + ... + N)/2 ~ $N^2/4$ compares

~ $N^2/4$ exchanges

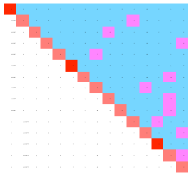


SELECTION

always search through right part

(1 + 2 + ... + N) ~ $N^2/2$ compares

~ N exchanges

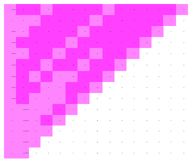


BUBBLE

mostly compare-exchanges

(1 + 2 + ... + N) ~ $N^2/2$ compares

~ $N^2/2$ exchanges



Bottom line: insertion/selection comparable; bubble twice as slow

Sorting Challenge 1

Problem: Sort a file of huge records with tiny keys

Example application: Reorganize your MP-3 files

Which sorting method to use?

A. a system sort, guaranteed to run in time ~N log N

B. selection sort

C. bubble sort

D. a custom algorithm for huge records/tiny keys

E. insertion sort

Sorting files with huge records and small keys

Insertion sort or bubble sort?

NO, too many exchanges

Selection sort?

YES, takes linear time under reasonable assumptions

Fast system sort or custom method?

Probably not: selection sort simpler and faster

Ex: 5,000 records, each 2,000,000 bytes with 100-byte keys

cost of comparisons: 5000×5000×100/2 = 1,250,000,000

cost of exchanges: 2,000,000×5000 = 10,000,000,000

(system method might be a factor of log 5000 slower)

Sorting Challenge 2

Problem: Sort a huge randomly-ordered file of small records

Application: Process transaction record for a phone company

Which sorting method to use?

A. bubble sort

B. selection sort

C. a system sort, guaranteed to run in time ~N log N

D. insertion sort

Sorting huge randomly-ordered files

Selection sort?

NO, always takes quadratic time

Bubble sort?

NO, quadratic for randomly-ordered files

Insertion sort?

NO, quadratic for randomly-ordered files

Fast system sort?

YES, it's designed for this problem.

Stay tuned for next lecture

Sorting Challenge 3

Problem: Sort a huge number of tiny files (independent of each other)

Application: Daily customer transaction records

Which sorting method to use?

A. bubble sort

B. selection sort

C. a system sort, guaranteed to run in time ~N log N

D. insertion sort

Sorting huge numbers of tiny files

Fast system sort or custom method?

NO, too much overhead

Insertion sort?

YES, much less overhead than system sort

Selection sort?

YES, same as insertion sort

Bubble sort?

NO, twice as slow as insertion/selection

Ex:

	4NlogN + 35	$2N^2$
4-record files	70	32

Sorting Challenge 4

Problem: Sort a file that is already almost in order

Applications:

Re-sort a huge database after a few changes

Doublecheck that someone else sorted a file

Which sorting method to use?

A. a system sort, guaranteed to run in time ~N log N

B. selection sort

C. bubble sort

D. a custom algorithm for almost-in-order files

E. insertion sort

Sorting files that are almost in order

Selection sort?

NO, always takes quadratic time

Bubble sort?

NO, bad for some definitions of “almost in order”

Ex: B C D E F G H I J K L M N O P Q R S T U V W X Y Z A

Insertion sort?

YES, takes linear time for most definitions of “almost in order”

Fast system sort or custom method?

Probably not: insertion sort simpler and faster

h-sort

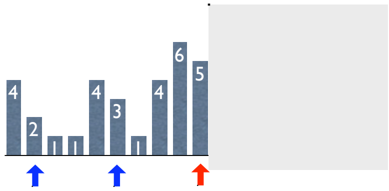
↑ scans from left to right

Invariant:

- keep elements to left of ↑ on the same h-cut in sorted order

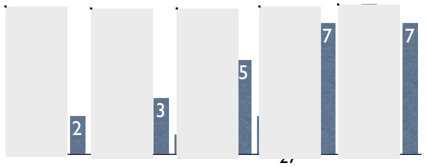
Def: An h-cut is defined by taking every h-th item

- file has h different h-cuts



Implication at end of scan:

file is h-sorted (each h-cut is sorted)



a 4-sorted file

Shellsort

h-sort for a sequence of increments, ending with 1

Ex: 3280-sort, then 1093-sort, then 364-sort, then 121-sort

then 40-sort, then 13-sort, then 4-sort, then 1-sort

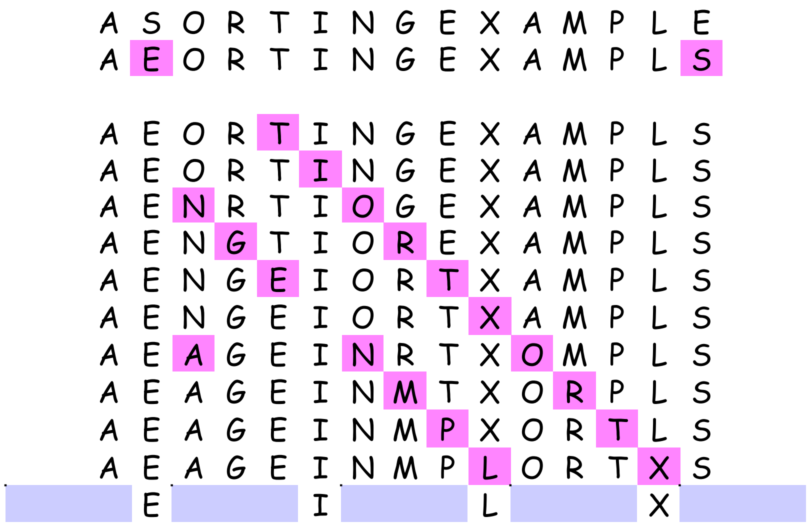
Invariants:

- a file that is h-sorted remains so even when you k-sort it
- 1-sort is insertion sort

Premises:

- big increments: easy to sort large number of small files
- small increments: easy to sort files that are nearly sorted

13-sorting, then 4-sorting, a file

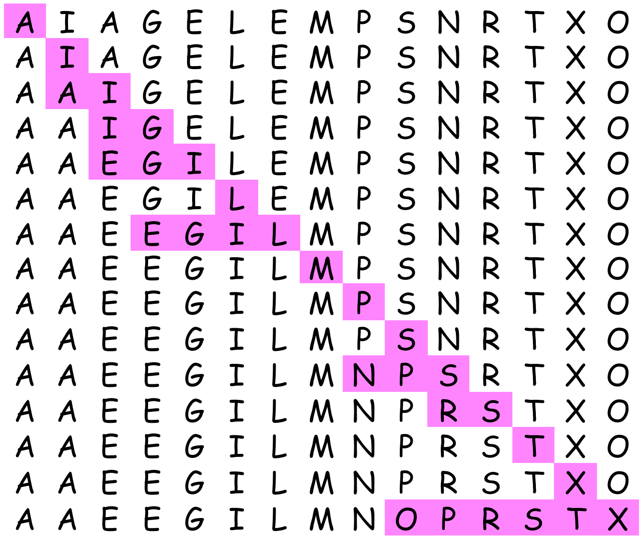


```
A S O R T I N G E X A M P L E
A E O R T I N G E X A M P L S

A E O R T I N G E X A M P L S
A E O R T I N G E X A M P L S
A E N R T I O G E X A M P L S
A E N G T I O R E X A M P L S
A E N G E I O R T X A M P L S
A E N G E I O R T X A M P L S
A E A G E I N R T X O M P L S
A E A G E I N M T X O R P L S
A E A G E I N M P X O R T L S
A E A G E I N M P L O R T X S
  E       I       L       X
```

First premise of Shellsort: easy to insertionsort tiny files

1-sorting a 4-sorted file



```
A I A G E L E M P S N R T X O
A I A G E L E M P S N R T X O
A A I G E L E M P S N R T X O
A A I G E L E M P S N R T X O
A A E G I L E M P S N R T X O
A A E G I L E M P S N R T X O
A A E E G I L M P S N R T X O
A A E E G I L M P S N R T X O
A A E E G I L M P S N R T X O
A A E E G I L M P S N R T X O
A A E E G I L M N P S R T X O
A A E E G I L M N P R S T X O
A A E E G I L M N P R S T X O
A A E E G I L M N P R S T X O
A A E E G I L M N O P R S T X
```

Second premise of Shellsort: easy to sort a file that is partially sorted

Shellsort code

```
void shellsort(Item a[], int l, int r)
  {int incs[16] = { 1391376, 463792, 198768,
      86961, 33936, 13776, 4592, 1968, 861,
      336, 112, 48, 21, 7, 3, 1 };
    for ( k = 0; k < 16; k++)
      { int h = incs[k];
        int i;
        for (i = l+h; i <= r; i++)
          { Item v = a[i];
            int j = i;
            while (j > l && less(v, a[j-h]))
              { a[j] = a[j-h]; j -= h; }
            a[j] = v;
          }
      }
  }
```

To sort a[l]...a[r]

h-sort for magic sequence of increments

insertion sort code when h=1

Shellsort increment sequences

What sequence to use? (Many have been heavily studied)

1 2 4 8 16 32 64...	No, could be quadratic at end
1 3 7 15 31 63 127...	OK, but slow
1 4 13 40 121 ...	Not bad, easy to compute
1 2 3 4 6 9 8 12 18 27...	No, too many increments
1 3 7 21 48 112 ...	about as good as we know

relatively prime? big common divisors? a mixture?

Open problems:

- Is there a set of ~20 numbers that makes Shellsort faster than any other known algorithm for huge random files??
- Average running time: N^{1+c} ?? N log N ??

Shellsort: an elementary algorithm that isn't so elementary

SEND MAIL if you find a sequence better than 1, 3, 7, 21, 48, ...