Sorting Algorithms

Biostatistics 615/815

Lecture 5

815 Projects

- 33% of your overall grade
- Hand-out details choice of 6 projects
  - MCMC evaluation of contingency table p-values
  - Rapid fitting of logistic regression models
  - Classify texts according to word distribution
  - Search for similar phrases in two texts
  - Fit a multivariate normal mixture distribution
  - Align short sequence reads

815 Projects – Next Step

- Rank project options
- E-mail me your choices by Friday
  - My address: goncalo@umich.edu
  - Subject: 815 Project
- Projects should be completed in pairs
  - If you have a partner preference, let me know!

Last Lecture ...

- Recursive Functions
  - Natural expression for many algorithms

- Dynamic Programming
  - Automatic strategy for generating efficient versions of recursive algorithms

Today …

- Properties of Sorting Algorithms

- Elementary Sorting Algorithms
  - Selection Sort
  - Insertion Sort
  - Bubble Sort

Applications of Sorting

- Facilitate searching
  - Building indices
- Identify quantiles of a distribution
- Identify unique values
- Browsing data

Elementary Methods

- Suitable for
  - Small datasets
  - Specialized applications
- Prelude to more complex methods
  - Illustrate ideas
  - Introduce terminology
  - Sometimes useful complement

… but beware!

- Elementary sorts are very inefficient
  - Typically, time requirements are $O(N^2)$
- Probably, most common inefficiency in scientific computing
  - Make programs “break” with large datasets

Aim

- Rearrange a set of keys
  - Using some predefined order
    - Integers
    - Doubles
    - Indices for records in a database
- Keys stored as array in memory
  - More complex sorts when we can only load part of the data

Basic Building Blocks

- An type for each element

```
#define Item int
```

- Compare two elements
- Exchange two elements
- Compare and exchange two elements

Comparing Two Elements

- Define a function to compare two elements

```
bool isLess(Item a, Item b)
    { return a < b; }
```

- Alternative is to use macros, but I don't recommend it

```
#define isLess(a,b)   ((a)<(b))
```

Exchanging Two Elements

- The best way is to use a C++ function

```
void Exchange(Item & a, Item & b)
    { Item temp = a; a = b; b = temp; }
```

- But using a macro is still an alternative

```
#define Exchange(a,b) \
    {                  \
    Item tmp = (a);    \
    (a) = (b);         \
    (b) = tmp;         \
    }
```

Comparing And Exchange

- Using C++ function

```
Item CompExch(Item & a, Item & b)
    {
    if (isLess(b, a))
       Exchange(a, b);
    }
```

- Using a macro

```
#define CompExch(a,b) \
  if (isLess((b),(a))) Exchange((a),(b));
```

A Simple Sort

- Gradually sort the array by:

- Sorting the first 2 elements
- Sorting the first 3 elements
- …
- Sort all N elements

A Simple Sort Routine

```
void sort(Item a[], int start, int stop)
  {
  int i, j;

  for (i = start + 1; i <= stop; i++)
    for (j = i; j > start; j--)
       CompExch(a[j-1], a[j]);
  }
```

Properties of this Simple Sort

- Non-adaptive
  - Comparisons do not depend on data
- Stable
  - Preserves relative order for duplicates
- Requires $O(N^2)$ running time

Sorts We Will Examine Today

- Selection Sort
- Insertion Sort
- Bubble Sort

Recipe: Selection Sort

- Find the smallest element
  - Place it at beginning of array
- Find the next smallest element
  - Place it in the second slot
- …

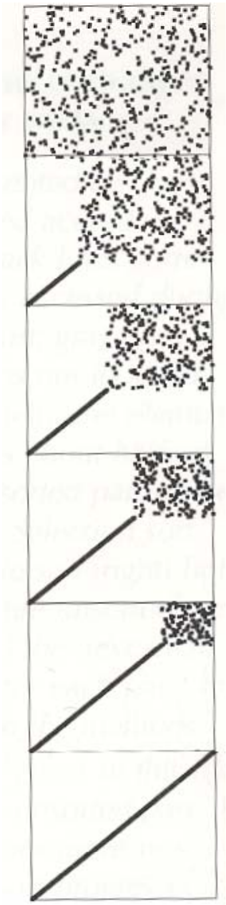
C Code: Selection Sort

```
void sort(Item a[], int start, int stop)
  {
  int i, j;

  for (i = start; i < stop; i++)
    {
    int min = i;
    for (j = i + 1; j < stop; j++)
       if (isLess(a[j], a[min])
          min = j;

    Exchange(a[i], a[min]);
    }
  }
```

Selection Sort



Notice:

Each exchange moves element into final position.

Right portion of array looks random.

Properties of Selection Sort

- Running time does not depend on input
  - Random data
  - Sorted data
  - Reverse ordered data…
- Performs exactly N-1 exchanges
- Most time spent on comparisons

Recipe: Insertion Sort

- The “Simple Sort” we first considered
- Consider one element at a time
  - Place it among previously considered elements
  - Must move several elements to “make room”
- Can be improved, by “adapting to data”

Improvement I

- Decide when further comparisons are futile
- Stop comparisons when we reach a smaller element
- What speed improvement do you expect?

Insertion Sort (I)

```
void sort(Item a[], int start, int stop)
  {
  int i, j;

  for (i = start + 1; i <= stop; i++)
    for (j = i; j > start; j--)
       if (isLess(a[j], a[j-1]))
         Exchange(a[j-1], a[j]);
       else
         break;
  }
```

Improvement II

- Notice that inner loop continues until:
  - First element reached, or
  - Smaller element reached
- If smallest element is at the beginning…
  - Only one condition to check

Insertion Sort (II)

```
void sort(Item a[], int start, int stop)
  {
  int i, j;

  // This ensures that smallest element is at the beginning
  for (i = stop; i > start; i--)
    CompExch(a[i-1], a[i]);

  // Now, we don't need to check that j > start
  for (i = start + 2; i <= stop; i++)
    {
    int j = i;
    while (isLess(a[j], a[j-1]))
       {
       Exchange(a[j], a[j-1]);
       j--;
       }
    }
  }
```

Improvement III

- The basic approach requires many exchanges involving each element
- Instead of carrying out many exchanges …
- Find out position for the new element and shift elements to the right to make room

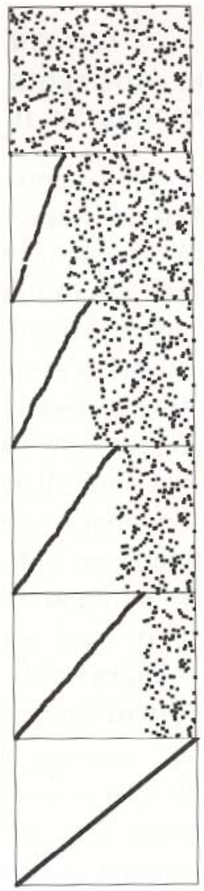
Insertion Sort (III)

```
void sort(Item a[], int start, int stop)
  {
  int i, j;

  for (i = stop; i > start; i--)
    CompExch(a[i-1], a[i]);

  for (i = start + 2; i <= stop; i++)
    {
    int j = i;
    Item val = a[j];             // Store the value of new element
    while (isLess(val, a[j-1])) // Proceed through larger elements
       {
       a[j] = a[j-1];           // Shifting things to the right …
       j--;
       }
    a[j] = val;                 // Finally, insert new element in place
    }
}
```

Insertion Sort



Notice:

Elements in left portion of array can still change position.

Right remains untouched.

Properties of Insertion Sort

- Adaptive version running time depends on input
  - About 2x faster on random data
  - Improvement even greater on sorted data
  - Similar speed on reverse ordered data
- Stable sort

Three Improvements Discussed

- Improvement I: Early termination
- Improvement II: Sentinels
- Improvement III: “Shift” instead of “swap”

- How would you rate their relative utility?
- Do any seem like tinkering with code?

Recipe: Bubble Sort

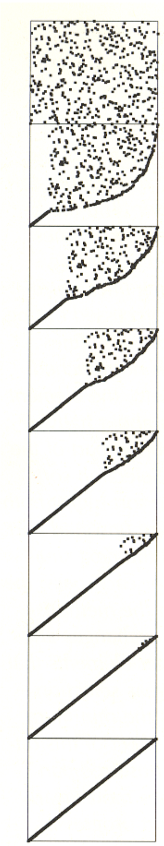
- Pass through the array
  - Exchange elements that are out of order
- Repeat until done…
- Very “popular”
  - Very inefficient too!

C Code: Bubble Sort

```
void sort(Item a[], int start, int stop)
  {
  int i, j;

  for (i = start; i <= stop; i++)
    for (j = stop; j > i; j--)
       CompExch(a[j-1], a[j]);
  }
```

Bubble Sort

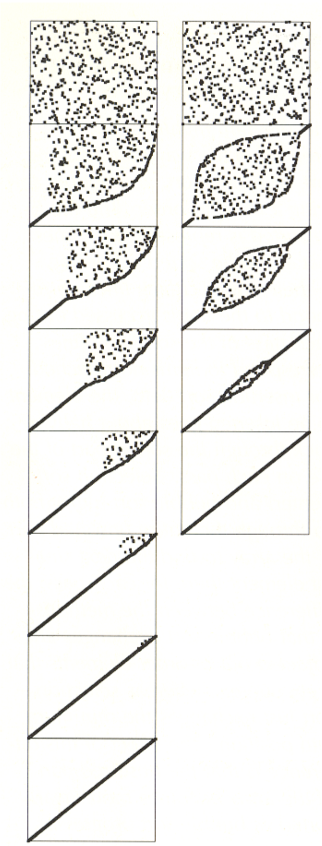


<u>Notice:</u>

Each pass moves one element into position.

Right portion of array is partially sorted

Shaker Sort



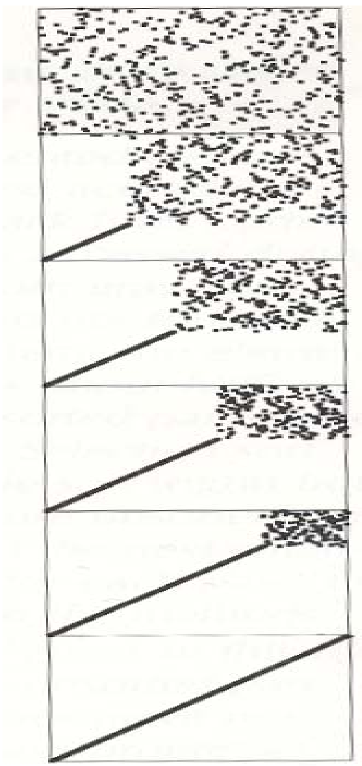
Notice:

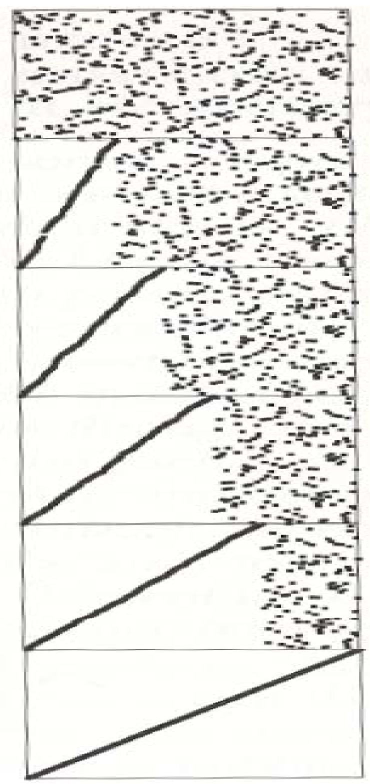
Things improve slightly if bubble sort alternates directions…

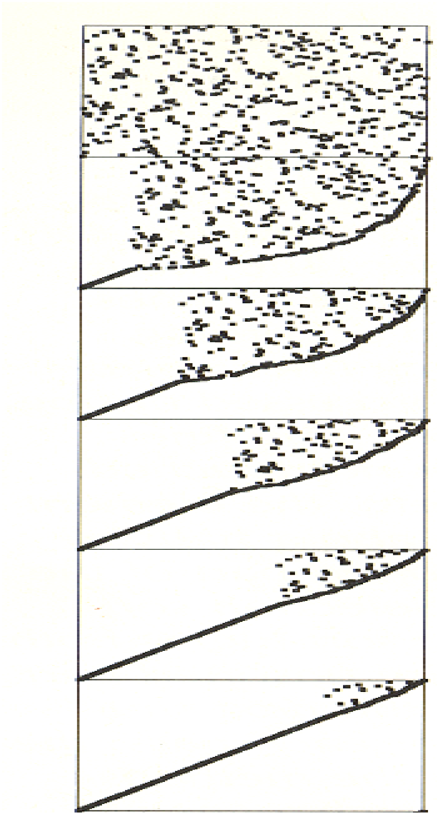
Notes on Bubble Sort

- Similar to non-adaptive Insertion Sort
  - Moves through unsorted portion of array
- Similar to Selection Sort
  - Does more exchanges per element
- Stop when no exchanges performed
  - Adaptive, but not as effective as Insertion Sort

Selection Insertion Bubble







Performance Characteristics

- Selection, Insertion, Bubble Sorts
- All quadratic
  - Running time differs by a constant
- Which sorts do you think are stable?

Selection Sort

- Exchanges
  - N – 1
- Comparisons
  - N * (N – 1) / 2
- Requires about N^2 / 2 operations
- Ignoring updates to min variable

Adaptive Insertion Sort

- Half - Exchanges
  - About N^2 / 4 on average (random data)
  - N * (N – 1) / 2 (worst case)

- Comparisons
  - About N^2 / 4 on average (random data)
  - N * (N – 1) / 2 (worst case)

- Requires about N^2 / 4 operations
- Requires nearly linear time on nearly sorted data

Bubble Sort

- Exchanges
  - N * (N – 1) / 2
- Comparisons
  - N * (N – 1) / 2
- Average case and worst case very similar, even for adaptive method

Empirical Comparison

	Sorting Strategy				
N	**Selection**	**Insertion**	**Insertion (adaptive)**	**Bubble**	**Shaker**
1000	5	7	4	11	8
2000	21	29	15	45	34
4000	85	119	62	182	138

(Running times in seconds)

Reading

- Sedgewick, Chapter 6