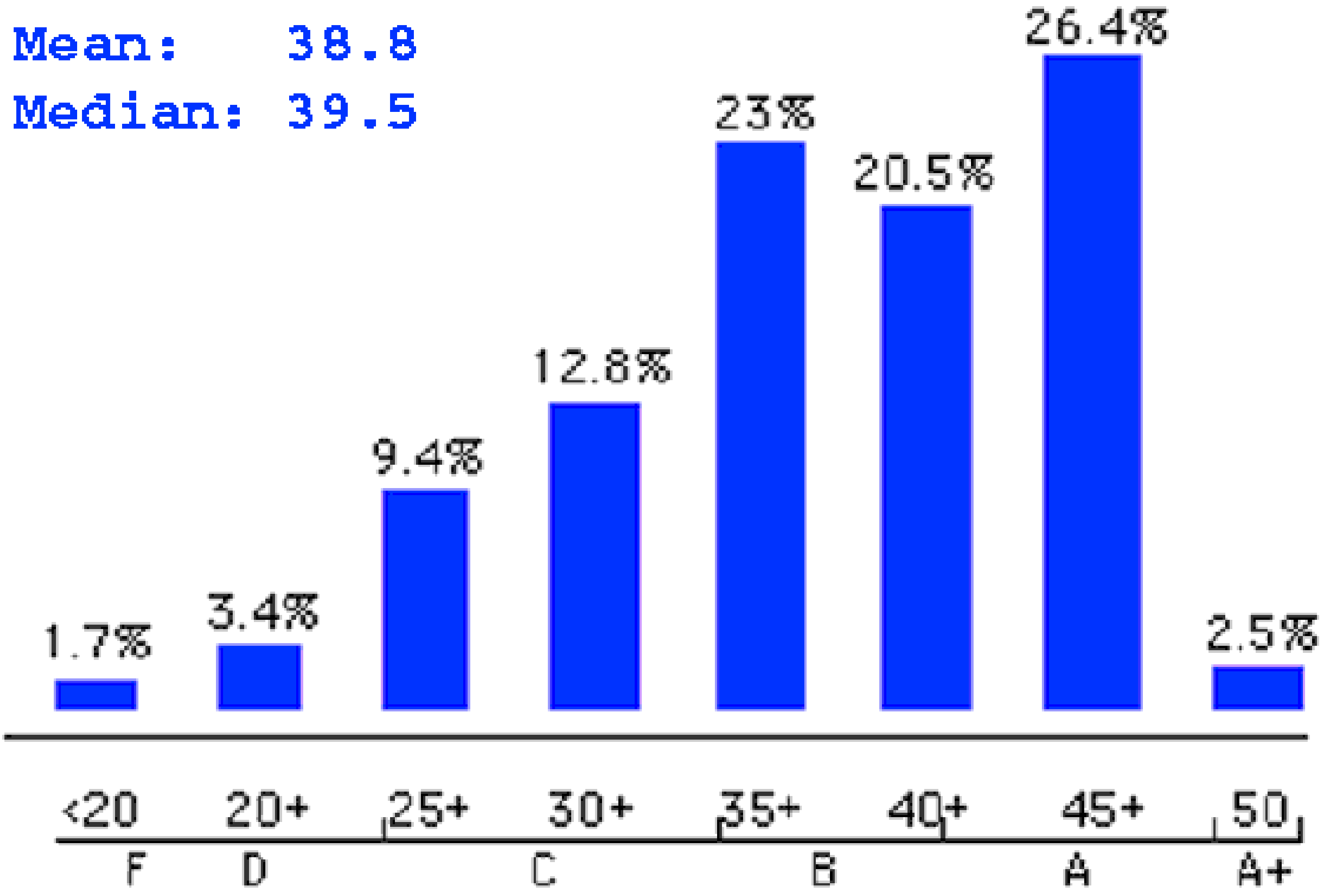
CS 126 Lecture T5: Algorithm Design/Analysis

Second Midterm Stats



Outline

- **Introduction**
- Insertion sort: algorithm
- Insertion sort: performance
- Quick sort: algorithm
- Quick sort: performance
- Conclusions

Where We Are

- T1 - T4:
  - **Computability**: whether a problem is solvable at all
  - Bad news: “most” problems are not solvable!
- T5 - T6:
  - **Complexity**: how long it takes to solve a problem
  - Bad news: many hard problems take so long to solve that they are almost as bad as non-solvable!
- Today:
  - **Examples** of “fast” vs. “slow” algorithms
- Thursday:
  - **Classes** of problems depending on how “hard” they are

Algorithm Design Tradeoffs

- Algorithm: step-by-step instruction of how to solve a problem
- There are usually many different algorithms for solving a single problem
- Goals
  - Correctness
  - Simplicity (elegance, ease of programming and debugging)
  - Time-efficient
  - Space-efficient
  - Other than correctness, the remaining goals are more often than not conflicting ones and can be traded off against each other
- We focus on speed here

How to Solve a Problem “Faster”?

- Wait till next year: bet on Moore’s Law: +60% per year?
  - Can’t wait till next year
  - 1.6 speedup is not enough
- Buy more machines
  - 2X machines result in < 2X speedup
  - Requires cleverness to use more machines efficiently
- Buy a faster machine
  - Supercomputers are a dying breed
  - This option is increasingly converging towards the last option
- Find a more clever algorithm
  - Potentially much greater gain than any of the above
  - Enables qualitative leaps instead of quantitative crawl

Example Problem: Sorting

- Problem: Given an array of integers, rearrange them so that they are in increasing order
- Of great practical importance in databases
- Important “data-intensive” benchmark (more on this later)

Outline

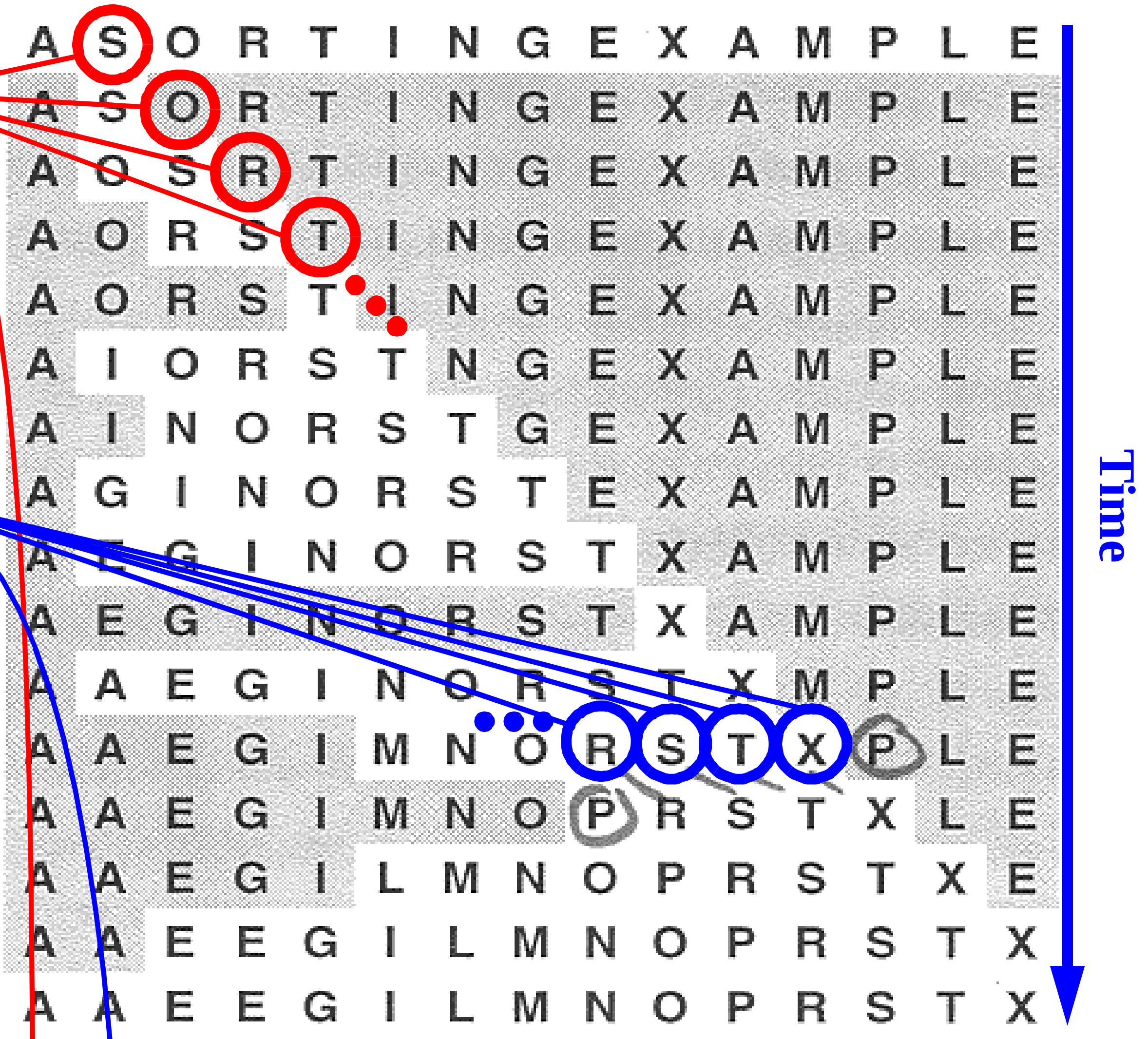
- ~~Introduction~~
- **Insertion sort: algorithm**
- Insertion sort: performance
- Quick sort: algorithm
- Quick sort: performance
- Conclusions

“Cat Sort” Demo

Insertion Sort

Each iteration of the outer loop sorts everything to the left of one array element a[i].

Each iteration of the inner loop compares this element to an element to its left (j).
By repeatedly swapping adjacent pairs from right to left, we put this element in its right spot at the end of the iteration.



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

```
void insertion(Item a[], int l, int r)
  { int i, j;
    for (i = l+1; i <= r; i++)
      for (j = i; j > l; j--)
        compexch(&a[j-1],&a[j]);
```

The Rest of the Code

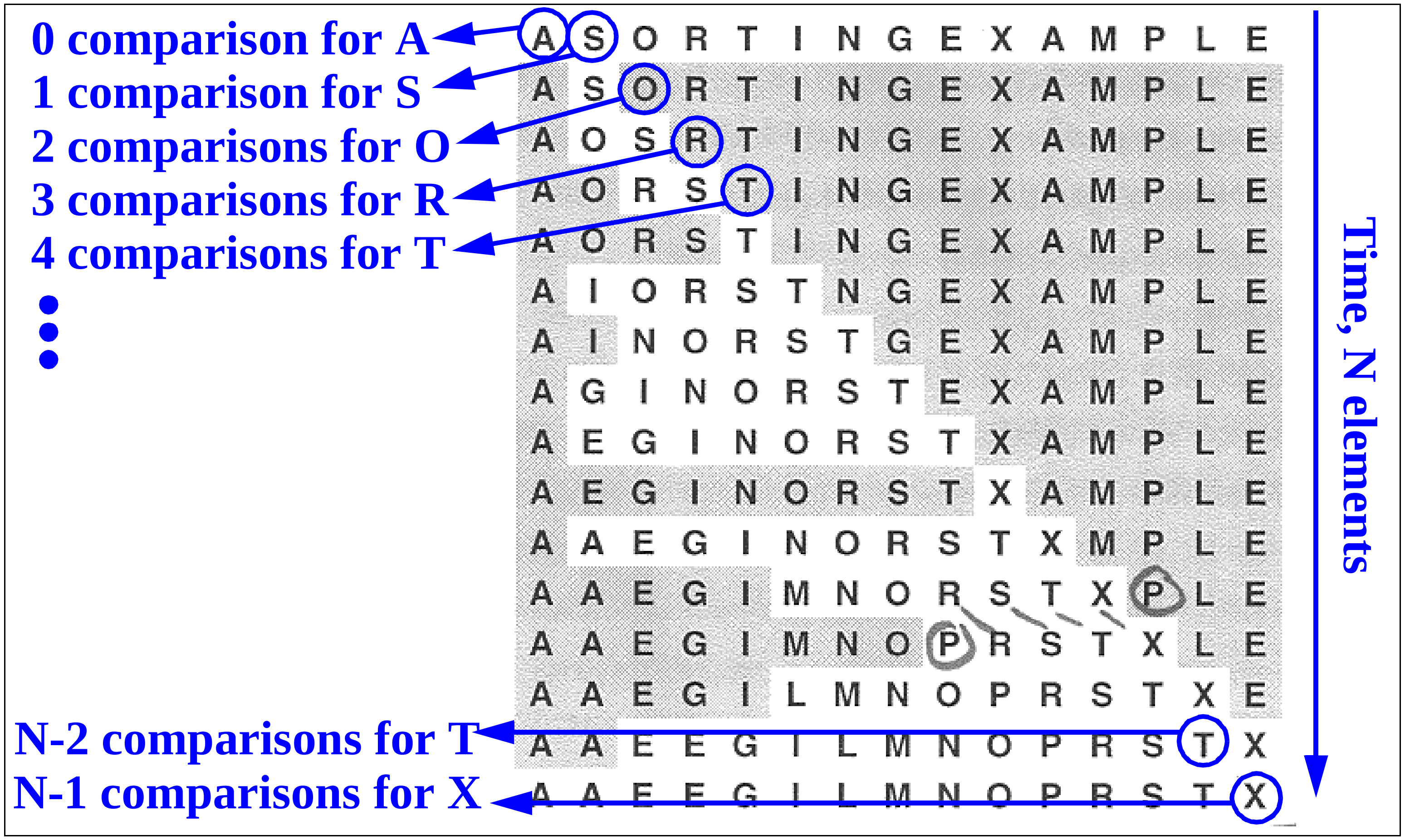
```
void
compexch (int *a, int *b) {
    int t;
    if (*b < *a) {
        t = *a;
        *a = *b;
        *b = t;
    }
}
```

- The course packet uses macros (#define), not wrong, but bad idea--bad style, for many reasons, don’t follow it.

Outline

- ~~Introduction~~
- ~~Insertion sort: algorithm~~
- **Insertion sort: performance**
- Quick sort: algorithm
- Quick sort: performance
- Conclusions

How Many Comparisons?

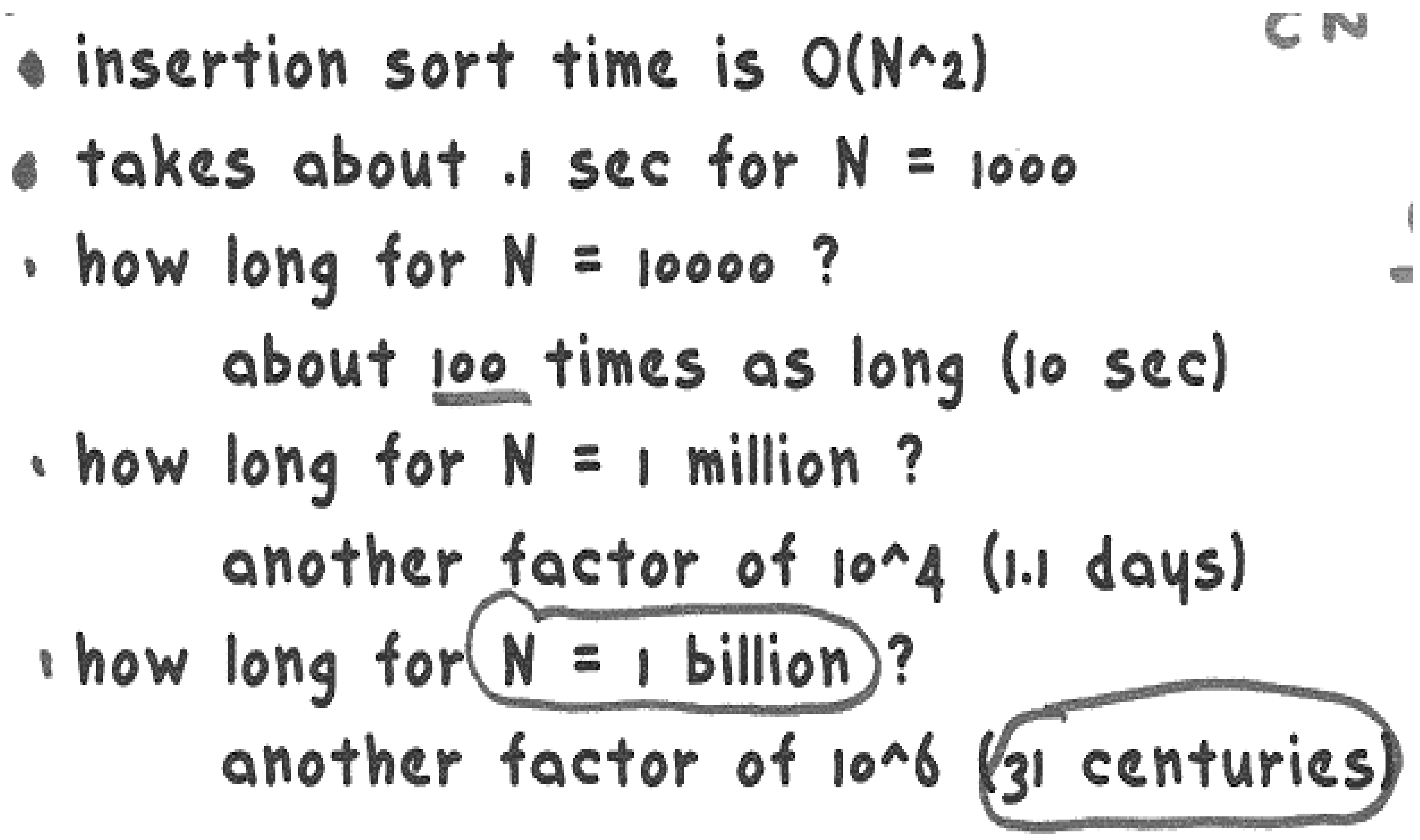

0 comparison for A
1 comparison for S
2 comparisons for O
3 comparisons for R
4 comparisons for T
N-2 comparisons for T
N-1 comparisons for X
Time, N elements
A S O R T I N G E X A M P L E
A S O R T I N G E X A M P L E
A O S R T I N G E X A M P L E
A O R S T I N G E X A M P L E
A O R S T I N G E X A 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
A A E E G I L M N O P R S T X

- Total comparisons: 0+1+2+3+...+(N-1) = (N-1)*N/2

Essential Description of Running Time: Big-O Notation

- Insertion sort takes $\frac{N \cdot (N-1)}{2} = \frac{N^2}{2} - \frac{N}{2}$ comparisons
- N/2 grows much slower than $N^2/2$, so we can toss that
- The constant 1/2 is affected by the details of a machine, which are not essential either.
- We are left only with N^2
- We say the complexity of insertion sort is ***$O(N^2)$***
- What is it good for? for example,
  - If we increase the size of the problem 10X,
  - We increase the running time 100X

More Examples of Growth Rate of $O(N^2)$



- insertion sort time is O(N^2)
- takes about .1 sec for N = 1000
- how long for N = 10000 ?

  about 100 times as long (10 sec)
- how long for N = 1 million ?

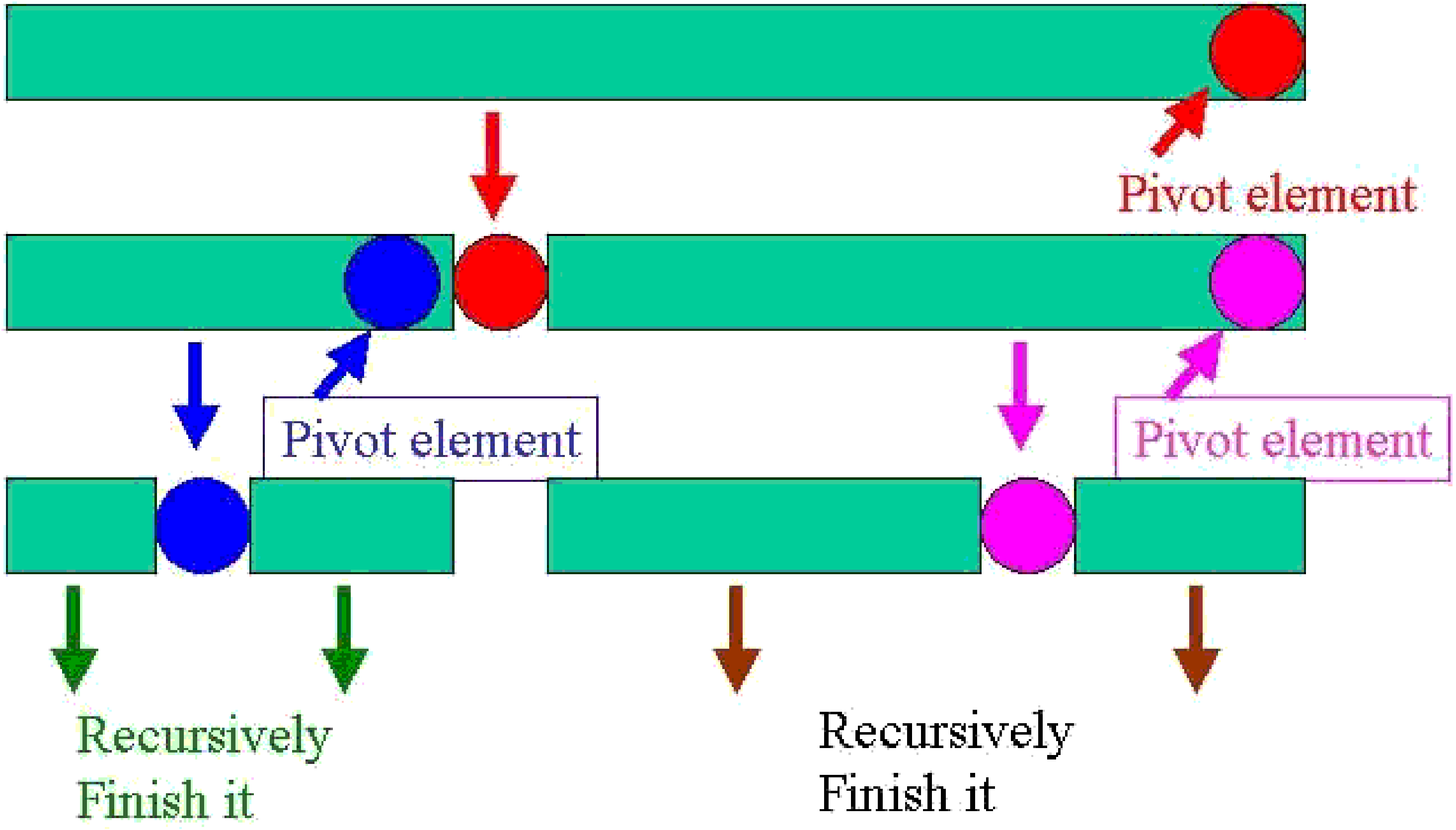
  another factor of 10^4 (1.1 days)
- how long for N = 1 billion ?

  another factor of 10^6 (31 centuries)

Outline

- ~~Introduction~~
- ~~Insertion sort: algorithm~~
- ~~Insertion sort: performance~~
- **Quick sort: algorithm**
- Quick sort: performance
- Conclusions

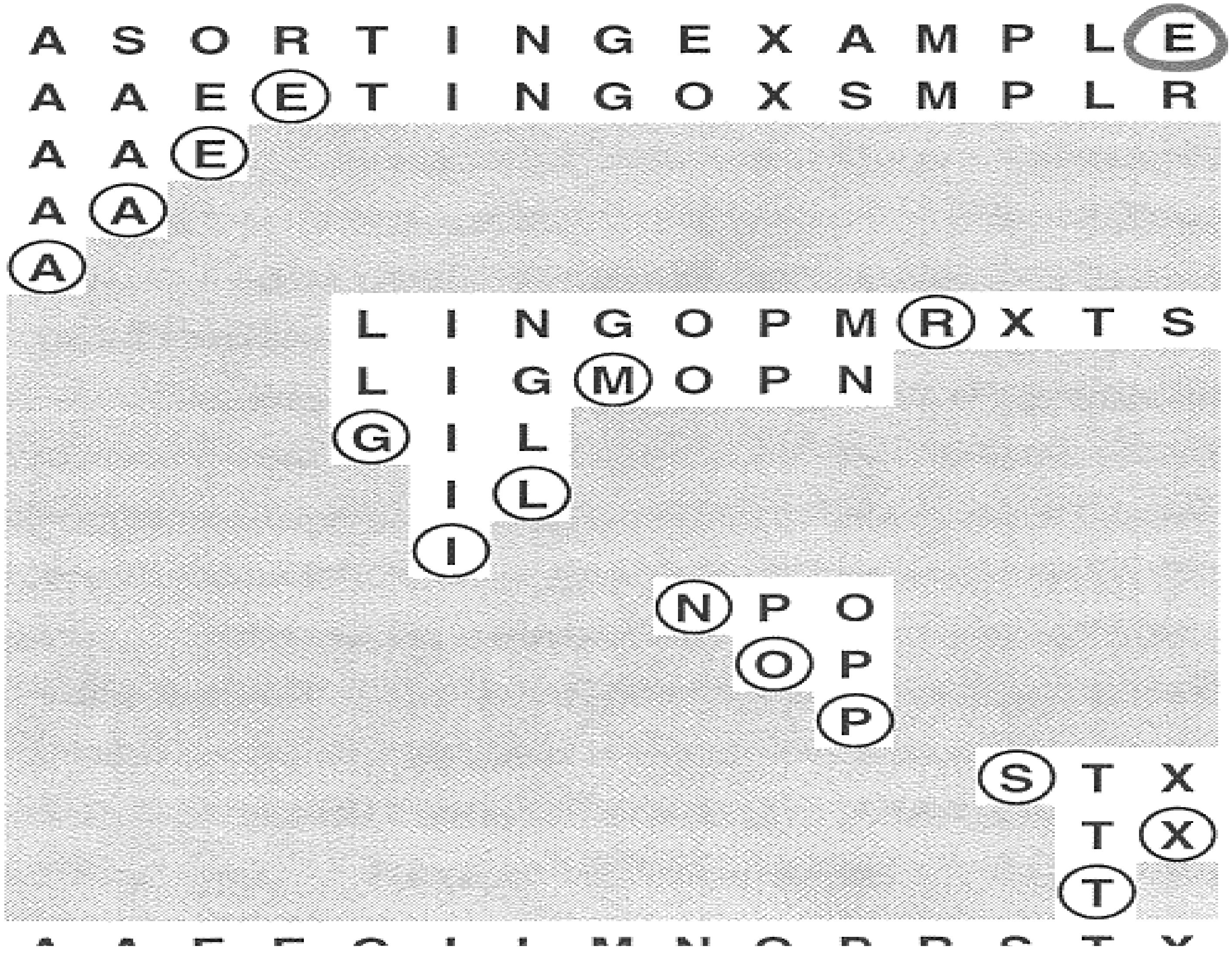
Demo Recursive Quicksort: Divide-and-Conquer



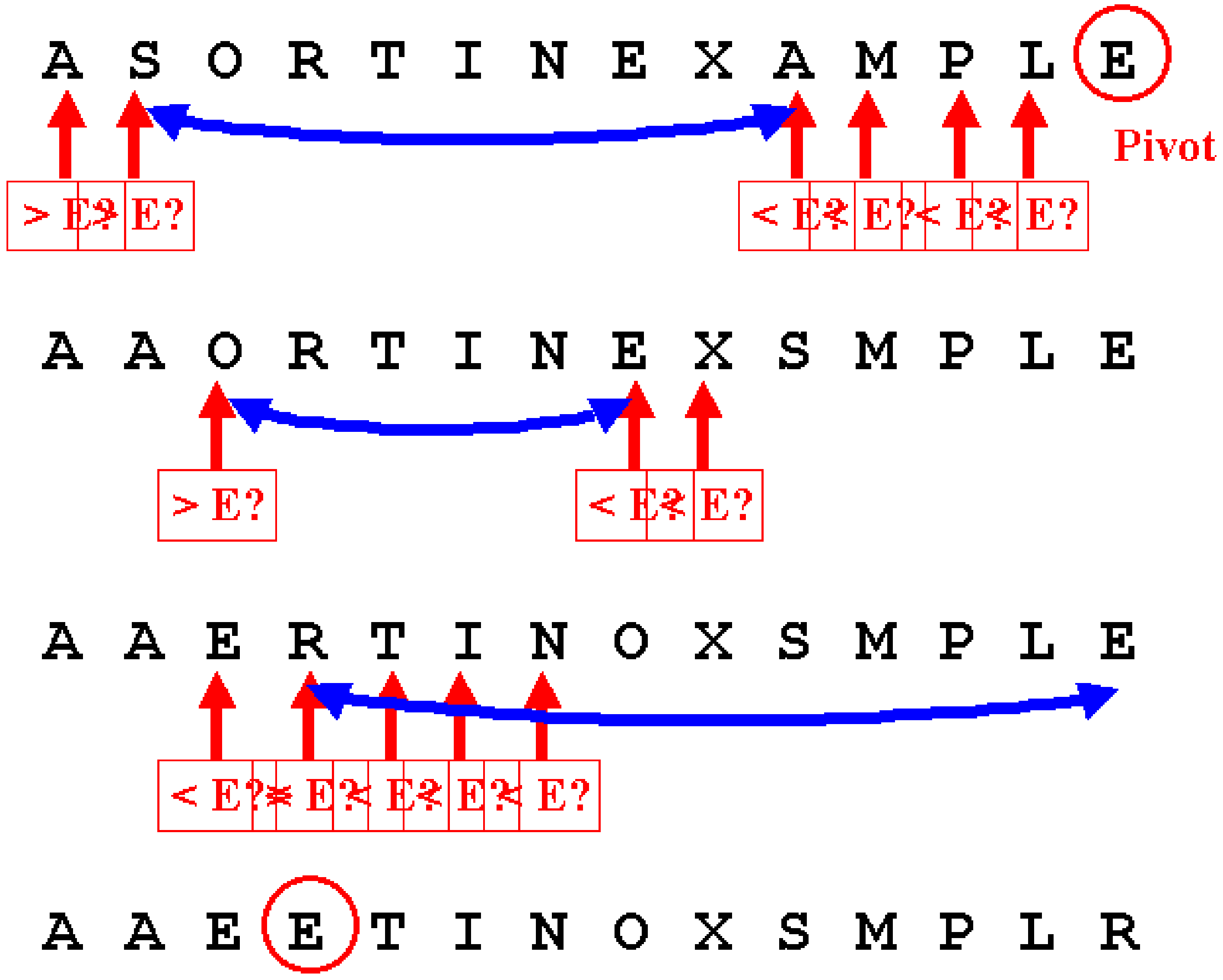
Quicksort Example

To sort an array, first divide it so that
- * some element a[i] is in its final position
- * no larger element left of i
- * no smaller element right of i

Then sort the left and right parts recursively

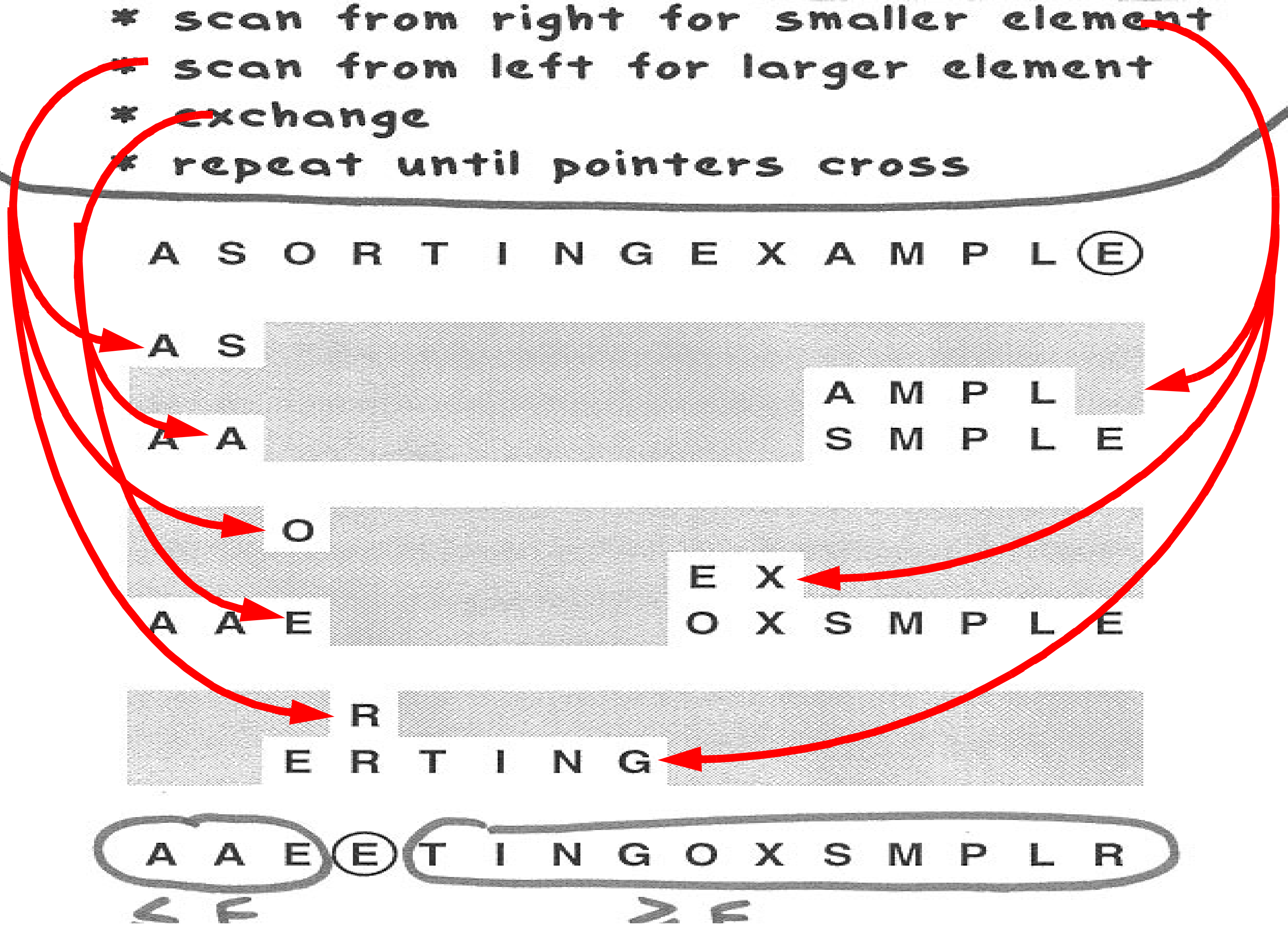


Partition Demo


A S O R T I N E X A M P L E
Pivot
> E? > E? < E? < E? < E? < E?
A A O R T I N E X S M P L E
> E? < E? < E?
A A E R T I N O X S M P L E
< E? > E? > E? > E? > E?
A A E E T I N O X S M P L R

Partitioning

To partition an array, pick a partitioning elemen
* scan from right for smaller element
* scan from left for larger element
* exchange
* repeat until pointers cross



Partitioning implementation

v: partitioning element
i: left-to-right pointer
j: right-to-left pointer

```
int partition(Item a[], int l, int r)
  {
    int i, j; Item v;
    v = a[r]; i = l-1; j = r;
    for (;;)
      {
        while (a[++i] < v) ;
        while (v < a[--j])
             if (j == l) break;
        if (i >= j) break;
        exch(&a[i],&a[j]);
      }
    exch(&a[i],&a[r]);
    return i;
  }
```

Scan from left → `while (a[++i] < v) ;`

Scan from right → `while (v < a[--j])`

Stop scanning if pointers cross → `if (j == l) break;`, `if (i >= j) break;`

Swap → `exch(&a[i],&a[j]);`

Put the pivot in place → `exch(&a[i],&a[r]);`

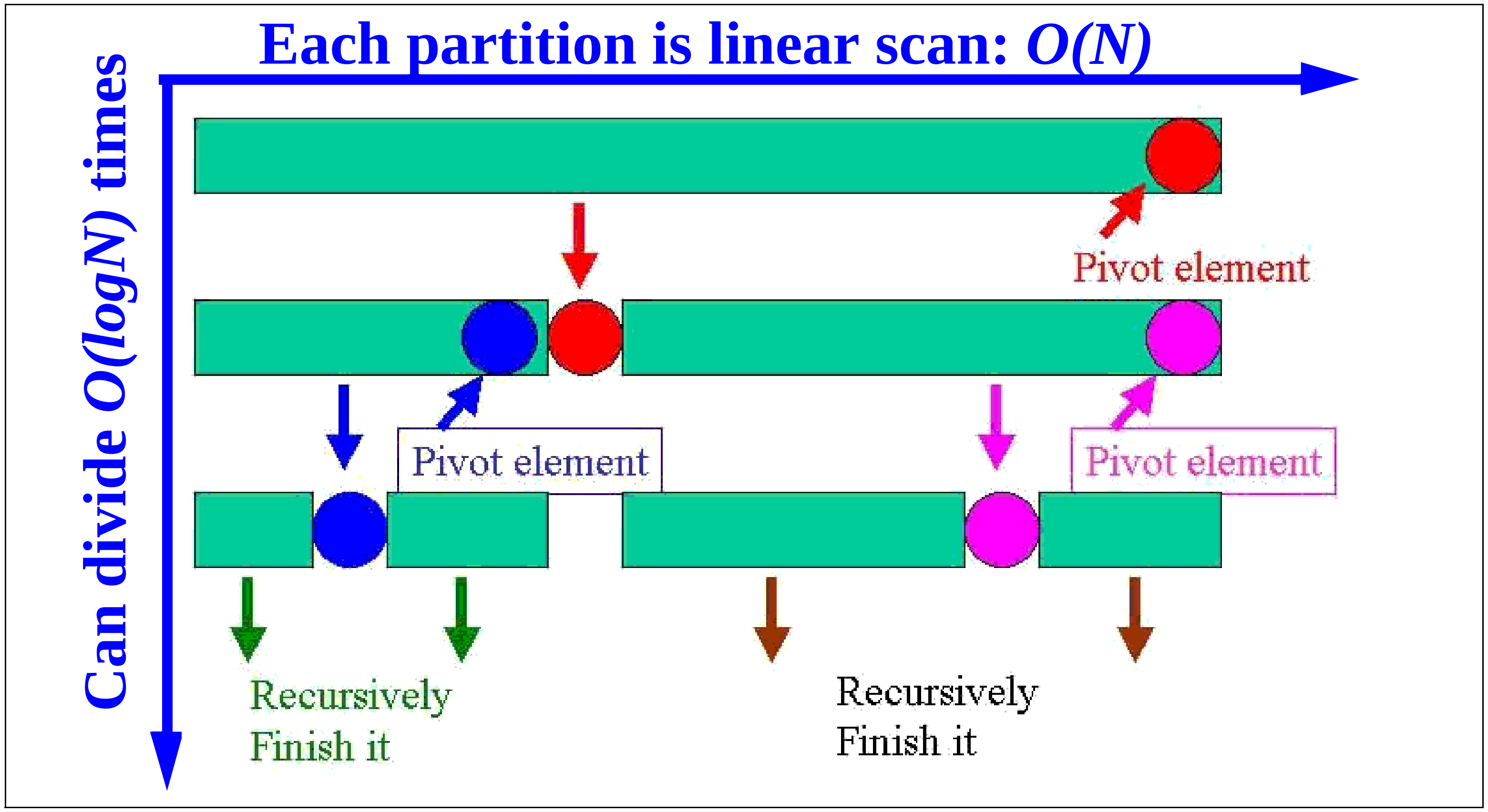
Quicksort implementation

```
quicksort(int a[], int l, int r)
  {
     int i;
     if (r > l)
        {
           i = partition(a, l, r);
           quicksort(a, l, i-1);
           quicksort(a, i+1, r);
        }
  }
```

Outline

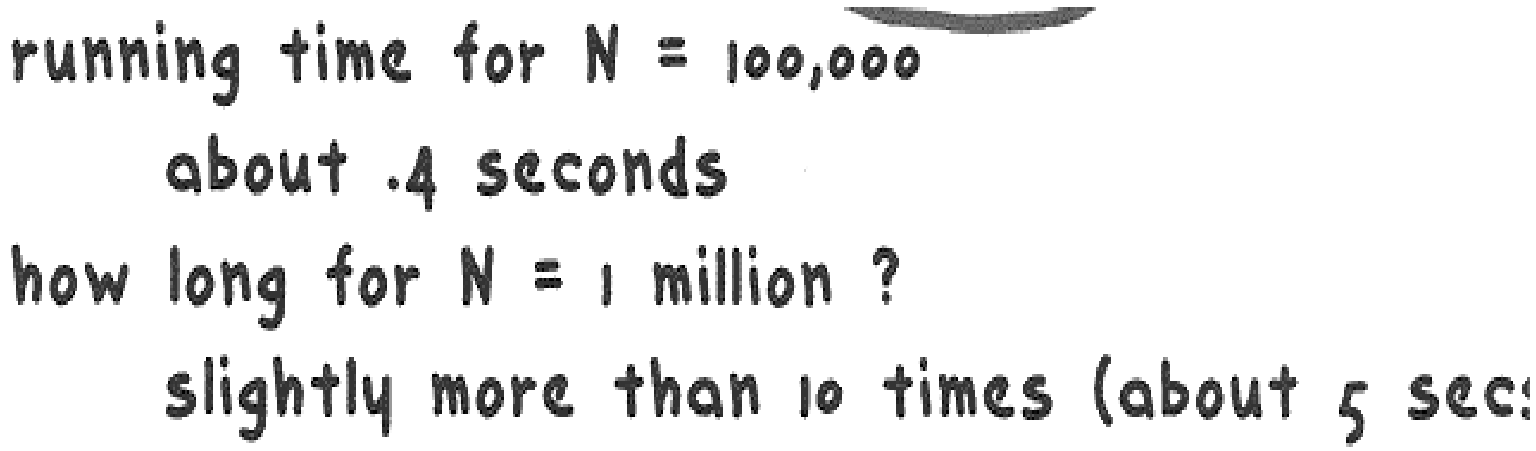
- ~~Introduction~~
- ~~Insertion sort: algorithm~~
- ~~Insertion sort: performance~~
- ~~Quick sort: algorithm~~
- **Quick sort: performance**
- Conclusions

How Many Comparisons?



- Quick sort is ***O(N*LogN)***

So What Does *O(N*LogN)* Mean in Time?



running time for N = 100,000

about .4 seconds

how long for N = 1 million ?

slightly more than 10 times (about 5 sec

Whereas insertion sort would take 100X, or 40 sec

Outline

- ~~Introduction~~
- ~~Insertion sort: algorithm~~
- ~~Insertion sort: performance~~
- ~~Quick sort: algorithm~~
- ~~Quick sort: performance~~
- **Conclusions**

Sorting analysis summary

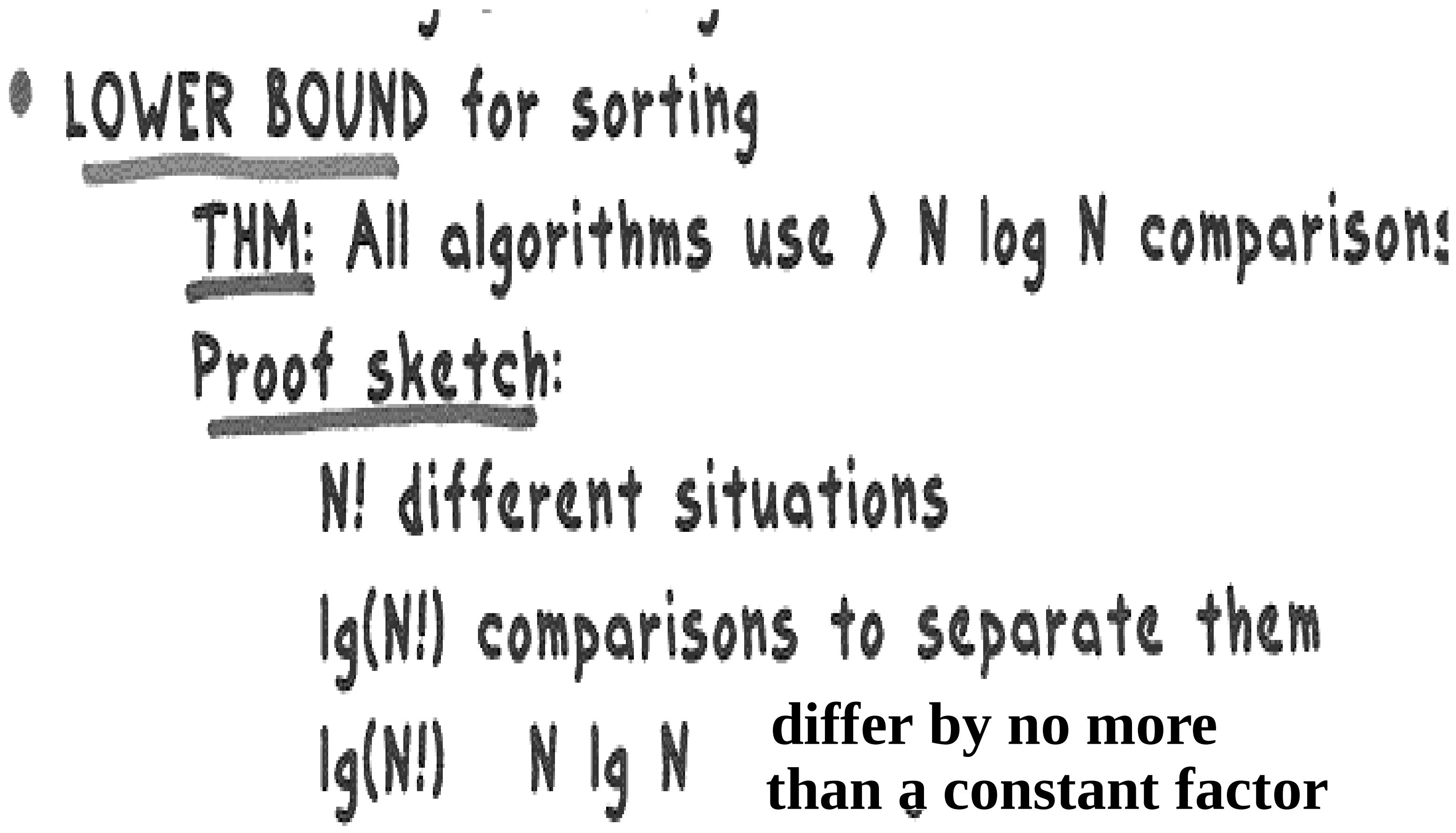
Good algorithms
are *more powerful* than supercomputers

Ex: assume that
home PC executes 10^8 comparisons/second
supercomputer does 10^{12} comparisons/second

Running time estimates

	thousand	million	billion
Insertion sort			
home PC	instant	2 hours	310 years
supercomputer	instant	1 sec	1.6 weeks
Quicksort			
home PC	instant	.28 sec	6 minutes
supercomputer	instant	instant	instant

Can We Do Better Than O(N*Log(N))?



- **LOWER BOUND** for sorting

  THM: All algorithms use > N log N comparisons

  Proof sketch:

  N! different situations

  lg(N!) comparisons to separate them

  lg(N!)   N lg N   **differ by no more than a constant factor**

What’s the Real World Like?

- Highly contested “land speed records”: Daytona vs. Indy
  - Daytona: commercially available systems
  - Indy: experimental systems
- 1999 sort records
  - Daytona Minute Sort: 7.6 GB, SGI 32-CPU Origin
  - Indy Minute Sort: 10.3 GB, 60 NT PCs, UIUC/UCSD
- Observations from previous records held at Berkeley:
  - The real world is a lot uglier!
  - Details hidden in the constant in $O(\mathbf{c}*N*LogN)$
  - Hard to make a giant cluster appear as a seamless whole
  - Difficult challenge for system software to optimize utilization of networks and disks

Obsession with Speed

- The obsession with speed is as old as computers, advances on all fronts
- The sort land speed records are a good illustration
- Theory
  - Better algorithms
  - New computation models: quantum computing?
- Architecture
  - Faster processors
  - Faster everything else: networks, disks, ...
- Systems software
  - Deliver the potential of the pile of silicon to applications

What We Have Learned Today

- Sort
  - How does insertion sort work? What's its complexity? Why is it so?
  - Same questions for quick sort.
- Complexity
  - Given simple/similar code, you should be able to analyze its complexity. Is it $O(LogN)$, $O(N)$, $O(N*LogN)$, $O(N^2)$, $O(N^3)$, ...?
  - Performance prediction by scaling problem size