COS 126 Lecture 23: Algorithm design/analysis

Refs: Algorithms in C
Bentley, "Writing Efficient Programs"

ALGORITHM
- "method" used to solve a problem
- generally independent of
  - machine, programming language

DESIGN: find a way to solve the problem
ANALYSIS: determine its effectiveness

New machine
- costs $$$ or more
- makes "everything" finish sooner
- may not help much with some specific problem
  - (but could)

New algorithm
- costs $ or less
- may make the difference
  - allowing specific problem to be solved
- may not help much with some other problems
  - (but could)

Ex: N-body simulation
- brute-force N^2 method
- N lg N algorithm enables new research

Case study: sorting

PROBLEM: Given an array of integers, rearrange them so that they are in increasing order

Ex: sort program with driver, in C

```
#include <stdio.h>
typedef int Item;
#define less(A, B) (A < B)
#define exch(A, B) { Item t = A; A = B; B = t; }
#define compexch(A, B) if (less(B, A)) exch(A, B)
void sort(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]);
  }
main(int argc, char *argv[])
  { int i, N = atoi(argv[1]), pr = atoi(argv[2]);
    int *a = malloc(N*sizeof(int));
    for (i = 0; i < N; i++)
      a[i] = 10000*(1.0*rand()/RAND_MAX);
    sort(a, 0, N-1);
    if (pr)
      for (i = 0; i < N; i++)
        printf("%4d ", a[i]);
  }
```

[see Program 6.1]

Insertion sort

Brute-force sorting solution

* move left-to-right through array
* exchange next element with larger
  elements to its left, one-by-one

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]);
  }
```

Profiling insertion sort

Use lcc "profiling" capability to get
frequency-of-execution counts

```
lcc -b insertion.c
a.out 1000 0
bprint
```

makes a file "prof.out" that has
the counts for each instruction

```
void insertionsort(Item a[], int l, int r)
  <1>{ int i, j;
  for (<1>i = l+1; <1000>i <= r; <999>i++)
    for (<999>j = i; <500499>j > l; <499500>j--)
    <499500>   compexch(a[j-<240732>1], a[j]);
  <1>}
```

Striking feature: HUGE numbers

To sort N items,
insertion sort does about (N^2)/2 comparisons

running time is QUADRATIC
[proportional to N^2]

Estimating the run time

Total run time =

    sum over all instructions: frequency * cost

frequency:

* determined by algorithm and input
* lcc -b (or analysis) can help estimate

cost:

* determined by compiler and machine
* could estimate by lcc -S (plus manuals)

Easier alternative

1. analyze asymptotic growth
2. for small N, run and measure time
3. for large N, use 1 and 2 to predict time

asymptotic growth rate:

    estimate time as function of input size
    ex: N, N*log N, N^2, N^3, 2^N

Ex:

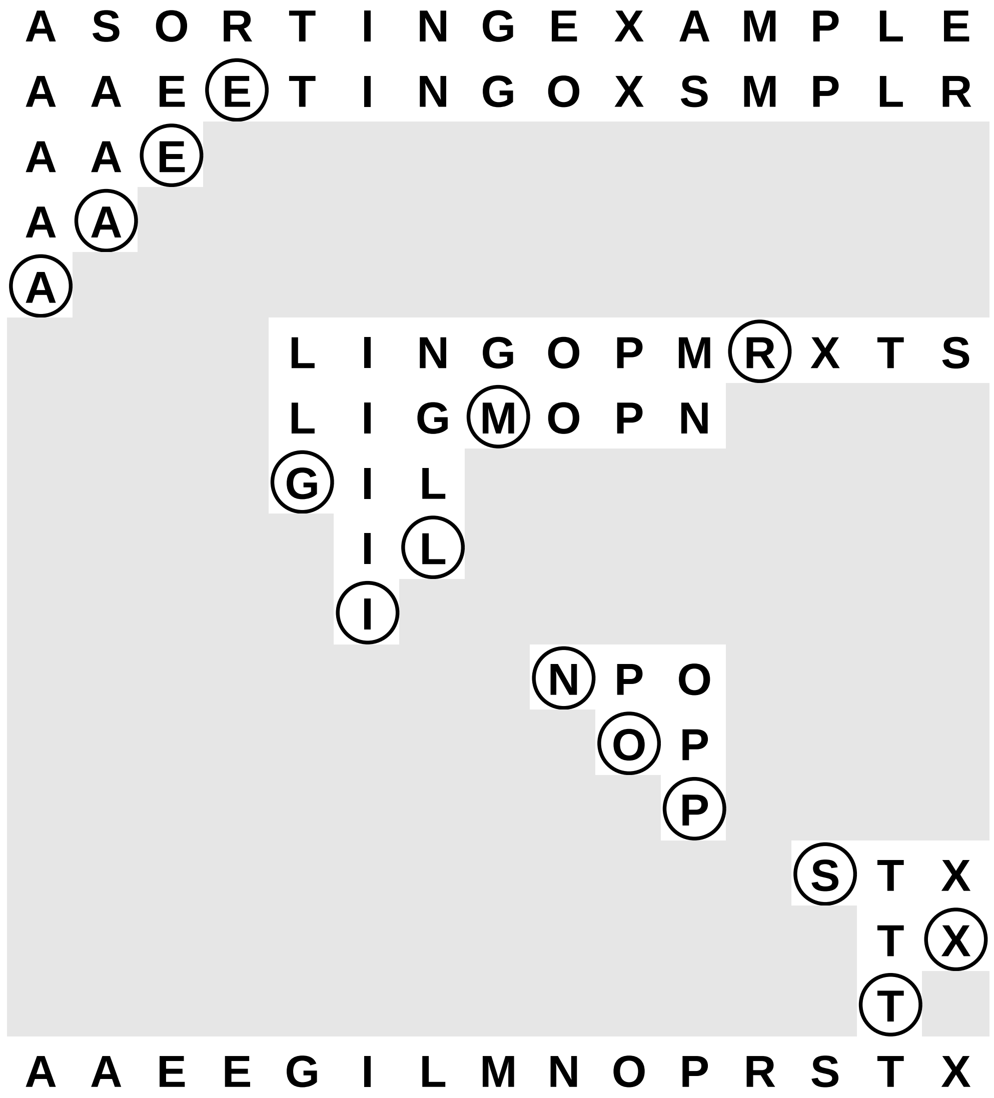
    insertion sort time is O(N^2)
    takes about .1 sec for N = 1000
    how long for N = 10000 ?
        about 100 times as long (10 sec)
    how long for N = 1 million ?
        another factor of 10^4 (1.1 days)
    how long for N = 1 billion ?
        another factor of 10^6 (31 centuries)

Quicksort

Divide-and-conquer sorting method

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



Partitioning

To partition an array, pick a partitioning element

* scan from right for smaller element
* scan from left for larger element
* exchange
* repeat until pointers cross

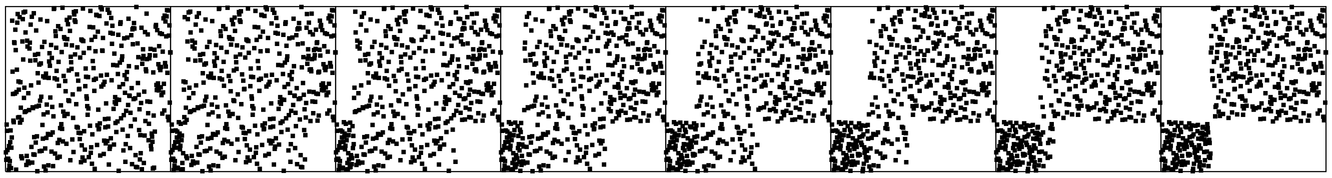
```
A S O R T I N G E X A M P L E

A S
                    A M P L
A A                 S M P L E

    O
                E X
A A E           O X S M P L E

      R
    E R T I N G

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



Partitioning implementation

v: partitioning element

i: left-to-right pointer

j: right-to-left pointer

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

Issues

* stop pointers on keys equal to v?
* sentinels or explicit tests for array bounds?
* details of pointer crossing

Quicksort implementation

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

Running time proportional to N*log N

running time for N = 100,000

about .4 seconds

how long for N = 1 million ?

slightly more than 10 times (about 5 secs

Novices beware: could be slow for some inputs

[stay tuned, COS 226]

Quicksort profile

```
int partition(Item a[], int l, int r)
   <654>{ int i = <654>l-1, j = <654>r;
               Item v = <654>a[r];
      for (<654>;<1704>;<1704>)
         <2358>{
            while (<6362>less(a[++i], v<4004>)) ;
            while (<5852>less(v, a[--j]))
               <3648>if (j == <154>l) break;
            if (<2358>i >= j) <654>break;
<1704>          <1704>exch(a[i<1704>], a[j]);
         <1704>}
<654>    exch(a[<654>i], a<654>[r]);
      return <654>i;
   }
void quicksort(Item a[], int l, int r)
   <1309>{ int i;
      if (<1309>r <= l) return<655>;
      <654>i = partition(a, l, r);
      <654>quicksort(a, l, i-1);
      <654>quicksort(a, i+1, r);
   <654>}
```

Striking feature: NO huge numbers

Sorting analysis summary

Good algorithms

are *more powerful* than supercomputers

Ex: assume that

home PC executes 10^7 comparisons/second

supercomputer does 10^12 comparisons/second

Running time estimates

.	thousand	million	billion
Insertion sort			
home PC	instant	1 day	3100 years
supercomputer	instant	1 sec	1.6 weeks
Quicksort			
home PC	instant	2.8 sec	1 hour
supercomputer	instant	instant	instant

Implementations and analysis

validate each other

Further refinements possible

design-implement-experiment-analyze cycle

Computational complexity

Framework to study efficiency of algorithms
    machine models, average case, worst-case
Upper bound: algorithm to solve the problem
Lower bound: proof that no alg can do better
Optimal algorithm: lower bound = upper bound
UPPER BOUND for sorting
    mergesort algorithm
        divide-and conquer method
        * sort two halves
        * merge them to make sorted whole
    always takes time proportional to N log N
        quicksort is usually faster
        mergesort is guaranteed not to be slow
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 [Stirling's formula]
Caveats
    worst- or average-case may be unrealistic
    costs ignored in analysis may dominate
    machine model may be restrictive
Complexity studies provide
    starting point for practical implementations
    indication of approaches to be avoided