Overview

Lecture T4:
- What is an algorithm?
  - Turing machine.
- Is it possible, in principle, to write a program to solve any problem?
  - No. Halting problem and others are unsolvable.

This Lecture:
- For many problems, there may be several competing algorithms.
  - Which one should I use?
- Computational complexity:
  - Rigorous and useful framework for comparing algorithms and predicting performance.
  - Use sorting as a case study.

Design and Analysis of Algorithms

Algorithm.
- "Step-by-step recipe" used to solve a problem.
- Generally independent of programming language or machine on which it is to be executed.

Design.
- Find a method to solve the problem.

Analysis.
- Evaluate its effectiveness and predict theoretical performance.

Implementation.
- Write actual code and test your theory.

Better Machines vs. Better Algorithms

New machine.
- Costs $$$ or more.
- Makes "everything" finish sooner.
- Incremental quantitative improvements (60% per year).
- May not help much with some problems.

New algorithm.
- Costs $ or less.
- Dramatic qualitative improvements possible! (million times faster)
- May make the difference, allowing specific problem to be solved.
- May not help much with some problems.
Impact of Better Algorithms

Example 1: N-body-simulation.
- Simulate the gravitational interactions among N bodies.
  - See Assignment 9.
  - Physicists want $N = \#$ atoms in universe.
- Brute force method takes $N^2$ steps.
- Appel (1985) algorithm takes $N \log N$ time and enables new research.

Example 2: Discrete Fourier Transform (DFT).
- Multiplying polynomials.
  - Foundation of signal processing
  - CD players, analyzing astronomical data, etc.
- Brute force method takes $N^2$ steps.
- Runge-König (1924), Cooley-Tukey (1965) FFT algorithm takes $N \log N$ time and enables new technology.

Case Study: Sorting

Sorting problem:
- Given an array of N integers, rearrange them so that they are in increasing order.
- Among most fundamental problems.

Insertion sort
- Brute-force sorting solution.
- Move left-to-right through array.
- Exchange next element with larger elements to its left, one-by-one.

Generic Item to Be Sorted

Define generic Item type to be sorted.
- Associated operations:
  - less, show, swap, rand
- Example: integers

```c
typedef int Item;

int ITEMless(Item a, Item b);
void ITEMshow(Item a);
void ITEMswap(Item *pa, Item *pb);
itm ITEMscan(Item *pa);
```

Item Implementation

```c
#include <stdio.h>
#include "ITEM.h"

int ITEMless(Item a, Item b) {
    return (a < b);
}

void ITEMswap(Item *pa, Item *pb) {
    Item t; t = *pa; *pa = *pb; *pb = t;
}

void ITEMshow(Item a) {
    printf("%4d ", a);
}

void ITEMscan(Item *pa) {
    return scanf("%d", pa);
}
```

swap integers – need to use pointers

Generic Sorting Program

```c
#include <stdio.h>
#include <stdlib.h>
#include "Item.h"
#define N 2000000

tmain(void) {
    int i, n = 0;
    Item a[N];
    for(ITEMscan(&a[n]) != EOF) 
        n++;
    sort(a, 0, n-1);  for (i = 0; i < n; i++)
        ITEMprint(a[i]);
    return 0;
}
```

Max number of items to sort.

Read input.

Call generic sort function.

Print results.

Insertion Sort Function

```c
void insertionsort(Item a[], int left, int right) {
    int i, j;
    for (i = left + 1; i <= right; i++)
        for (j = i; j > left; j--)
            if (ITEMless(a[j], a[j-1]))
                ITEMswap(&a[j], &a[j-1]);
            else
                break;
}
```

Insertion sort function.

Profiling Insertion Sort Empirically

Use lcc "profiling" capability.
- Automatically generates a file "prof.out" that has frequency counts for each instruction.
- Striking feature:
  - HUGE numbers!

```c
prof.out
```

```c
void insertionsort(Item a[], int left, int right) <1>{
    int i, j;
    for (<1>i = left + 1; <1000>i <= right; <999>i++)
        for (<999>j = i; <256320>j > left; <255321>j--)
            if (<256313>ITEMless(a[j], a[j-1]))
                ITEMswap(&a[j], &a[j-1]);
            else
                break;
} <1>
```
Profiling Insertion Sort Analytically

How long does insertion sort take?
- Depends on number of elements \( N \) to sort.
- Depends on specific input.
- Depends on how long compare and exchange operation takes.

Worst case.
- Elements in reverse sorted order.
  - \( i^{th} \) iteration requires \( i - 1 \) compare and exchange operations
  - \( \text{total} = 0 + 1 + 2 + \ldots + N-1 = \frac{N(N-1)}{2} \)

Average case.
- Elements are randomly ordered.
  - \( i^{th} \) iteration requires \( \frac{i}{2} \) comparison on average
  - \( \text{total} = 0 + \frac{1}{2} + \frac{2}{2} + \ldots + \frac{(N-1)/2}{2} = \frac{N(N-1)}{4} \)
  - check with profile: 249750 vs. 256313

Estimating the Running Time

Total run time:
- Sum over all instructions: frequency * cost.

Frequency:
- Determined by algorithm and input.
- Can use `lcc -b` (or analysis) to help estimate.

Cost:
- Determined by compiler and machine.
- Could estimate by `lcc -s` (plus manuals).
Estimating the Running Time

Easier alternative.
(i) Analyze asymptotic growth.
(ii) For small $N$, run and measure time.
   For large $N$, use (i) and (ii) to predict time.

Asymptotic growth rates.
- Estimate time as a function of input size.
  - $N$, $N \log N$, $N^2$, $N^3$, $2^N$, $N!$
- Big-Oh notation hides constant factors and lower order terms.
  - $6N^3 + 17N^2 + 56$ is $O(N^3)$

Insertion sort is $O(N^2)$. Takes 0.1 sec for $N = 1,000$.
- How long for $N = 10,000$? 10 sec (100 times as long)
- $N = 1$ million? 1.1 days (another factor of $10^4$)
- $N = 1$ billion? 31 centuries (another factor of $10^6$)

Average Case vs. Worst Case

Worst-case analysis.
- Take running time of worst input of size $N$.
- Advantages:
  - performance guarantee
- Disadvantage:
  - pathological inputs can determine run time

Average case analysis.
- Take average run time over all inputs of some class.
- Advantage:
  - can be more accurate measure of performance
- Disadvantage:
  - hard to quantify what input distributions will look like in practice
  - difficult to analyze for complicated algorithms, distributions
  - no performance guarantee

Sorting Case Study: mergesort

Insertion sort (brute-force)
Mergesort (divide-and-conquer)
- Divide array into two halves.
- Sort each half separately.
- Merge two halves to make sorted whole.

Profiling Mergesort Analytically

How long does mergesort take?
- Bottleneck = merging (and copying).
  - merging two files of size $N/2$ requires $N$ comparisons
- $T(N) = \text{comparisons to mergesort array of } N \text{ elements.}$

Unwind recurrence: (assume $N = 2^k$).

$$T(N) = \begin{cases} 
0 & \text{if } N = 1 \\
2T(N/2) + N & \text{merging} \\
2T(N/4) + N/2 & \text{sorting both halves} \\
2T(N/8) + 2N & \text{sorting both halves} \\
\vdots \\
N T(1) + k N & \text{merging} \\
0 + N \log_2 N & \text{merging}
\end{cases}$$
Profiling Mergesort Analytically

How long does mergesort take?
- Bottleneck = merging (and copying).
  - merging two files of size \( N/2 \) requires \( N \) comparisons
- \( N \log_2 N \) comparisons to sort *any* array of \( N \) elements.
  - even already sorted array!

How much space?

Implementing Mergesort

```c
void mergesort(Item a[], int left, int right) {
    int mid = (right + left) / 2;
    if (right <= left)
        return;
    mergesort(a, left, mid);
    mergesort(a, mid + 1, right);
    merge(a, left, mid, right);
}
```

merge (see Sedgewick Program 8.2)

```c
void merge(Item a[], int left, int mid, int right) {
    int i, j, k;
    for (i = mid+1; i > left; i--)
        aux[i-1] = a[i-1];
    for (j = mid; j < right; j++)
        aux[right+mid-j] = a[j+1];
    for (k = left; k <= right; k++)
        if (ITEMless(aux[i], aux[j]))
            a[k] = aux[i++];
        else
            a[k] = aux[j--];
}
```

Sorting Case Study: quicksort

Insertion sort (brute-force)
Mergesort (divide-and-conquer)
Quicksort (conquer-and-divide)

- Partition array so that:
  - some partitioning element \( a[m] \) is in its final position
    - no larger element to the left of \( m \)
    - no smaller element to the right of \( m \)
- Sort each "half" recursively.
Sorting Case Study: quicksort

Insertion sort (brute-force)
Mergesort (divide-and-conquer)
Quicksort (conquer-and-divide)

- Partition array so that:
  - some partitioning element \( a[m] \) is in its final position
  - no larger element to the left of \( m \)
  - no smaller element to the right of \( m \)
- Sort each "half" recursively.

```c
void quicksort(Item a[], int left, int right) {
    int m; if (right > left) {
        m = partition(a, left, right);
        quicksort(a, left, m - 1);
        quicksort(a, m + 1, right);
    }
}
```

Implementing Partition

```c
int partition(Item a[], int left, int right) {
    int i = left-1; /* left to right pointer */
    int j = right; /* right to left pointer */
    Item p = a[right]; /* partition element */
    for(;;) {
        while (ITEMless(a[++i], p));
        while (ITEMless(p, a[--j])) if (j == left) break;
        if (i >= j) break;
        Itemswap(&a[i], &a[j]);
    }
    Itemswap(&a[i], &a[right]);
    return i;
}
```

Profiling Quicksort Empirically

```c
void quicksort(Item a[], int left, int right) {
    int p;
    if (right <= left)
        return;
    p = partition(a, left, right);
    quicksort(a, left, p-1);
    quicksort(a, p+1, right);
}
```

Striking feature: no HUGE numbers!

```
Profiling Quicksort Empirically
```
int partition(Item a[], int left, int right) { 
    int i = left - 1, j = right; 
    Item swap, p = a[right]; 
    for(; ; ) { 
        while (ITEMless(a[++i], p)) ; 
        while (ITEMless(p, a[--j])) 
            if (j == left) break; 
        if (i >= j) break; 
        ITEMswap(&a[i], &a[j]); 
    } 
    ITEMswap(&a[i], &a[right]); return i; 
}

Profiling Quicksort Empirically

int partition(Item a[], int left, int right) { 
    int i = left - 1, j = right; 
    Item swap, p = a[right]; 
    for(; ; ) { 
        while (ITEMless(a[++i], p)) ; 
        while (ITEMless(p, a[--j])) 
            if (j == left) break; 
        if (i >= j) break; 
        ITEMswap(&a[i], &a[j]); 
    } 
    ITEMswap(&a[i], &a[right]); return i; 
}

Striking feature: no huge numbers!
Profiling Quicksort Analytically

Partition on random element:
- No bad inputs.
- Algorithm can get unlucky and take $N^2$ time.

Partition on median element:
- Guaranteed $N \log N$ performance.
- But need to find median element in $O(N)$ time.
  - see COS 226/423


Sorting Analysis Summary

Running time estimates:
- Home pc executes $10^8$ comparisons/second.
- Supercomputer executes $10^{12}$ comparisons/second.

<table>
<thead>
<tr>
<th>Computer</th>
<th>Insertion Sort ($N^2$)</th>
<th>Quicksort ($N \log N$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Home pc</td>
<td>instant</td>
<td>instant</td>
</tr>
<tr>
<td>Supercomputer</td>
<td>instant, 2 hour</td>
<td>instant, 0.3 sec</td>
</tr>
<tr>
<td></td>
<td>310 years</td>
<td>6 min</td>
</tr>
</tbody>
</table>

- Implementations and analysis validate each other.
- Further refinements possible.
  - design-analysis-implement cycle

Good algorithms are more powerful than supercomputers.

Comparison of Different Sorting Algorithms

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Insertion Sort</th>
<th>Quick Sort</th>
<th>Merge Sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Worst case</td>
<td>$N^2$</td>
<td>$N^2$</td>
<td>$N \log N$</td>
</tr>
<tr>
<td>Best case</td>
<td>$N$</td>
<td>$N \log N$</td>
<td>$N \log N$</td>
</tr>
<tr>
<td>Average case</td>
<td>$N^2$</td>
<td>$N \log N$</td>
<td>$N \log N$</td>
</tr>
<tr>
<td>Already sorted</td>
<td>$N$</td>
<td>$N$</td>
<td>$N \log N$</td>
</tr>
<tr>
<td>Reverse sorted</td>
<td>$N^2$</td>
<td>$N^2$</td>
<td>$N \log N$</td>
</tr>
<tr>
<td>Space</td>
<td>$N$</td>
<td>$N$</td>
<td>2 $N$</td>
</tr>
<tr>
<td>Stable</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

Computational Complexity

Framework to study efficiency of algorithms.
- Depends on machine model, average case, worst case.
- UPPER BOUND = algorithm to solve the problem.
- LOWER BOUND = proof that no algorithm can do better.
- OPTIMAL ALGORITHM: lower bound = upper bound.

Example: sorting.
- Measure costs in terms of comparisons.
- Upper bound = $N \log_3 N$ (mergesort).
  - quicksort usually faster, but mergesort never slow
- Lower bound = $N \log_3 N - N \log_3 e$
  (applies to any comparison-based algorithm).
  - Why?

Sorting algorithms have different performance characteristics.
- Other choices: bubblesort, heapsort, shellsort, selection sort, shaker sort, radix sort, BST sort, solitaire sort, hybrid methods.

Q: Which one should I use?
A: Depends on application.
Comparison Based Sorting

- $a_1 < a_2$
- $a_1 < a_3$
- $a_2 < a_3$
- $a_1 < a_3$
- $a_2 < a_3$

2, 1, 3
2, 3, 1
3, 2, 1
1, 3, 2
3, 1, 2
1, 2, 3

Lower Bound

Lower bound = $N \log_2 N$ (applies to any comparison-based algorithm).

- Worst case dictated by tree height $h$.
- $N!$ different orderings.
- One (or more) leaves corresponding to each ordering.
- Binary tree with $N!$ leaves must have

$$h \geq \log_2 (N!)
\geq \log_2 \left( \frac{N}{e} \right)^N
= N \log_2 N - N \log_2 e
= \Theta \left( N \log_2 N \right)$$

Computational Complexity

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.

Summary

How can I evaluate the performance of a proposed algorithm?
- Computational experiments.
- Complexity theory.

What if it’s not fast enough?
- Use a faster computer.
- performance improves incrementally
- Understand why.
- Develop a better algorithm (if possible).
- performance can improve dramatically