Lecture T4: Analysis of Algorithm

Overview

Lecture T3:

- 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.

Linear Growth

Grade school addition.

- Work is proportional to number of digits N.
- Linear growth: kN for some constant k.

| N = 4 | 1 1 1 0 + 1 1 1 0 = 1 1 0 0 0 1 |
| N = 8 | 1 1 1 1 1 1 0 1 + 0 1 1 1 1 1 0 1 = 1 0 1 0 1 0 0 1 0 |

2N reads
2N + 1 write operations
N odd parity operations
N majority operations

Quadratic Growth

Grade school multiplication.

- Work is proportional to square of number of digits N.
- Quadratic growth: kN^2 for some constant k.

| N = 4 | 1 0 1 1 * 1 1 0 1 = 1 0 1 1 |
| N = 8 | 1 1 0 1 0 1 0 1 * 0 1 1 1 1 1 1 0 1 = 1 1 0 1 0 1 0 1 0 |

2N reads
N^2 + 2N + 1 writes
N-1 adds on N-bit integers
### Why Does It Matter?

<table>
<thead>
<tr>
<th>Run time (nanoseconds)</th>
<th>1.3 N^3</th>
<th>10 N^2</th>
<th>47 N log N</th>
<th>48 N</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>1.3 seconds</td>
<td>10 msec</td>
<td>0.4 msec</td>
<td>0.048 msec</td>
</tr>
<tr>
<td>10,000</td>
<td>22 minutes</td>
<td>1 second</td>
<td>6 msec</td>
<td>0.48 msec</td>
</tr>
<tr>
<td>100,000</td>
<td>15 days</td>
<td>1.7 minutes</td>
<td>78 msec</td>
<td>4.8 msec</td>
</tr>
<tr>
<td>million</td>
<td>41 years</td>
<td>2.8 hours</td>
<td>0.94 seconds</td>
<td>48 msec</td>
</tr>
<tr>
<td>10 million</td>
<td>41 millennia</td>
<td>1.7 weeks</td>
<td>11 seconds</td>
<td>0.48 seconds</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Max size problem solved in one</th>
</tr>
</thead>
<tbody>
<tr>
<td>second</td>
</tr>
<tr>
<td>minute</td>
</tr>
<tr>
<td>hour</td>
</tr>
<tr>
<td>day</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>N multiplied by 10, time multiplied by</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000</td>
</tr>
</tbody>
</table>

### Orders of Magnitude

<table>
<thead>
<tr>
<th>Seconds</th>
<th>Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 second</td>
</tr>
<tr>
<td>10</td>
<td>10 seconds</td>
</tr>
<tr>
<td>10^2</td>
<td>1.7 minutes</td>
</tr>
<tr>
<td>10^3</td>
<td>17 minutes</td>
</tr>
<tr>
<td>10^4</td>
<td>2.8 hours</td>
</tr>
<tr>
<td>10^5</td>
<td>1.1 days</td>
</tr>
<tr>
<td>10^6</td>
<td>1.6 weeks</td>
</tr>
<tr>
<td>10^7</td>
<td>3.8 months</td>
</tr>
<tr>
<td>10^8</td>
<td>3.1 years</td>
</tr>
<tr>
<td>10^9</td>
<td>3.1 decades</td>
</tr>
<tr>
<td>10^10</td>
<td>3.1 centuries</td>
</tr>
</tbody>
</table>

**Meters Per Second**

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>10^{-10}</td>
</tr>
<tr>
<td>10^{-8}</td>
</tr>
<tr>
<td>10^{-6}</td>
</tr>
<tr>
<td>10^{-4}</td>
</tr>
<tr>
<td>10^{-2}</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>10^2</td>
</tr>
<tr>
<td>10^4</td>
</tr>
<tr>
<td>10^6</td>
</tr>
<tr>
<td>10^8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Powers of 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2^{10} thousand</td>
</tr>
<tr>
<td>2^{20} million</td>
</tr>
<tr>
<td>2^{30} billion</td>
</tr>
</tbody>
</table>

### Historical Quest for Speed

**Multiplication:** \( a \times b \).

- **Naive:** add \( a \) to itself \( b \) times. \( N 2^N \) steps
- **Grade school.** \( N^2 \) steps
- **Divide-and-conquer (Karatsuba, 1962).** \( N^{1.58} \) steps
- **Ingenuity (Schönhage and Strassen, 1971).** \( N \log N \log \log N \) steps

**Greatest common divisor:** \( \gcd(a, b) \).

- **Naive:** factor \( a \) and \( b \), then find \( \gcd(a, b) \). \( 2^N \) steps
- **Euclid (20 BCE):** \( \gcd(a, b) = \gcd(b, a \mod b) \). \( N \) steps

### Better Machines vs. Better Algorithms

**New machine.**

- Costs $$$ or more.
- Makes “everything” finish sooner.
- Incremental quantitative improvements (Moore’s Law).
- 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 gravitational interactions among N bodies.
  - physicists want N = # atoms in universe
- Brute force method: N^2 steps.

Example 2: Discrete Fourier Transform (DFT).
- Breaks down waveforms (sound) into periodic components.
  - foundation of signal processing
  - CD players, JPEG, analyzing astronomical data, etc.
- Grade school method: N^2 steps.
  FFT algorithm: N log N steps, enables new technology.

Case Study: Sorting

Sorting problem:
- Given N items, 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.

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

Item.h
Generic Sorting Program

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

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

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;
}
```

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
% lcc -b insertion.c item.c % a.out < sort1000.txt % bprint
```

```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]))
                <255321>ITEMswap(&a[j], &a[j-1]);
            else
                <992>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.
  - 1st iteration requires 1 - 1 compare and exchange operations
  - total = 0 + 1 + 2 + ... + N-1 = N (N-1) / 2

![Diagram](image1)

![Diagram](image2)

Average case.
- Elements are randomly ordered.
  - 1st iteration requires i / 2 comparison on average
  - total = 0 + 1/2 + 2/2 + ... + (N-1)/2 = N (N-1) / 4
  - check with profile: 249,750 vs. 256,313

![Diagram](image3)

Worst case: N (N - 1) / 2.
Best case: N - 1.
Average case: N (N - 1) / 4.
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 use `lcc -s` (plus manuals).

Easier alternative.
(i) Analyze asymptotic growth.
(ii) For medium 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!
- Ignore lower order terms and leading coefficients.
  - Ex. 6N^3 + 17N^2 + 56 is proportional to N^3

Insertion sort is quadratic. On arizona: 1 second for N = 10,000.
- How long for N = 100,000? 100 seconds (100 times as long).
- N = 1 million? 2.78 hours (another factor of 100).
- N = 1 billion? 317 years (another factor of 10^6).

Sorting Case Study: mergesort

Insertion sort (brute-force)
Mergesort (divide-and-conquer)
- Divide array into two halves.
- Sort each half separately. How do we sort half size files?
  - Any sorting algorithm will do.
  - Use mergesort recursively!

```
MERGESORTME
MERGESORTME
divide
EEGMRS EMORT
```

```
MERGESORTME
MERGESORTME
divide
EEGMRS EMORT
sort
EEGM toxic
merge
```
Profiling Mergesort Analytically

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

\[
T(N) = \begin{cases} 
0 & \text{if } N = 1 \\
2T(N/2) + N & \text{otherwise} 
\end{cases}
\]

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?
- Can’t do “in-place” like insertion sort.
- Need auxiliary array of size N.

Implementing Mergesort

```c
Item aux[MAXN];

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);
}
```
Implementing Mergesort

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--];
}

Profiling Mergesort Empirically

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--];
}

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

Mergesort prof.out

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--];
}

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

Profile Mergesort

\# comparisons
Theory ~ N log 2 N = 9,966
Actual = 9,976

Quicksort

Quick sort.
- 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 \(m\)

Quicksort

Quick sort.
- 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);
    }
}
```

partition (see Sedgewick Program 7.2)

```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 */

    while(1) {
        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;
}
```

Implementing Partition

Profiling Quicksort Empirically

```c
void quicksort(Item a[], int left, int right) <1337> {
    int p;if (<1337> right <= left)
        return <669> ;
    <668>p = partition(a, left, right);
    <668>quicksort(a, left, p-1);
    <668>quicksort(a, p+1, right);
    <1337>}
```

Quicksort prof.out

Striking feature: no HUGE numbers!
Profiling Quicksort Empirically

```c
int partition(Item a[], int left, int right) {
    int i = left-1, j = right;
    Item p = a[right];
    while (1) {
        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

Intuition.
- Assume all elements unique.
- Assume we always select median as partition element.
- T(N) = # comparisons.

![Analysis formula](image)

If N is a power of 2.
⇒ \( T(N) = N \log_2 N \)

Can you find median in O(N) time?
- Yes, see COS 226/423.


Profiling Quicksort Analytically

Partition on median element.
- Proportional to N log₂ N in best and worst case.

Partition on rightmost element.
- Proportional to N² in worst case.
- Already sorted file: takes \( N^2/2 + N/2 \) comparisons.

Partition on random element.
- Roughly 2 \( N \log_2 N \) steps.
- Choose random partition element.

Check profile.
- 2 \( N \log_2 N \): 13815 vs. 12372 (5708 + 6664).
- Running time for \( N = 100,000 \) about 1.2 seconds.
- How long for \( N = 1 \) million?
  - Slightly more than 10 times (about 12 seconds)

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 (N²)</th>
<th>Quicksort (N lg N)</th>
</tr>
</thead>
<tbody>
<tr>
<td>home</td>
<td>thousand 2.8 hours</td>
<td>thousand instant</td>
</tr>
<tr>
<td>super</td>
<td>million 1 second</td>
<td>million instant</td>
</tr>
<tr>
<td></td>
<td>billion 317 years</td>
<td>billion 6 min</td>
</tr>
</tbody>
</table>

![Comparison table](image)

Lesson: good algorithms are more powerful than supercomputers.
Design, Analysis, and Implementation 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.

Sorting Analysis Summary

Comparison of Different Sorting Algorithms

<table>
<thead>
<tr>
<th>Attribute</th>
<th>insertion</th>
<th>quicksort</th>
<th>mergesort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Worst case complexity</td>
<td>$N^2$</td>
<td>$N^2$</td>
<td>$N \log_2 N$</td>
</tr>
<tr>
<td>Best case complexity</td>
<td>$N$</td>
<td>$N \log_2 N$</td>
<td>$N \log_2 N$</td>
</tr>
<tr>
<td>Average case complexity</td>
<td>$N^2$</td>
<td>$N \log_2 N$</td>
<td>$N \log_2 N$</td>
</tr>
<tr>
<td>Already sorted</td>
<td>$N$</td>
<td>$N^2$</td>
<td>$N \log_2 N$</td>
</tr>
<tr>
<td>Reverse sorted</td>
<td>$N^2$</td>
<td>$N^2$</td>
<td>$N \log_2 N$</td>
</tr>
<tr>
<td>Space</td>
<td>$N$</td>
<td>$N$</td>
<td>$2N$</td>
</tr>
<tr>
<td>Stable</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

Sorting algorithms have different performance characteristics.
- Other choices: BST sort, bubblesort, heapsort, shellsort, selection sort, shaker sort, radix sort, distribution sort, solitaire sort, hybrid methods.
- Which one should I use?
  - Depends on application.

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 = $N \log_2 N$ (mergesort).
  - quicksort usually faster, but mergesort never slow
- Lower bound = $N \log_2 N - N \log_2 e$
  (applies to any comparison-based algorithm).
  - Why?
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.
  - Discover a better algorithm.
    - performance can improve dramatically
    - not always easy / possible to develop better algorithm