Lecture T4: Analysis of Algorithms

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?
- Analysis of algorithms.
  - framework for comparing algorithms and predicting performance
    - case study: sorting
- Computational complexity.
  - framework for studying intrinsic difficulty of problems

Linear Growth

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

Grade school multiplication.
- Work is proportional to square of number of digits $N$.
- Quadratic growth: $kN^2$ for some constant $k$.
**Why Does It Matter?**

<table>
<thead>
<tr>
<th>Run time in nanoseconds —&gt;</th>
<th>1.3 $N^3$</th>
<th>10 $N^2$</th>
<th>47 $N \log_2 N$</th>
<th>48 $N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time to solve a problem of size</td>
<td>1000</td>
<td>1.3 seconds</td>
<td>10 msec</td>
<td>0.4 msec</td>
</tr>
<tr>
<td></td>
<td>10,000</td>
<td>22 minutes</td>
<td>1 second</td>
<td>6 msec</td>
</tr>
<tr>
<td></td>
<td>100,000</td>
<td>15 days</td>
<td>1.7 minutes</td>
<td>78 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>

| Max size problem solved in one | second | 920 | 1 million | 21 million |
|                               | minute | 3,600 | 77,000 | 49 million | 1.3 billion |
|                               | hour   | 14,000 | 600,000 | 2.4 trillion | 76 trillion |
|                               | day    | 41,000 | 2.9 million | 50 trillion | 1,800 trillion |
| N multiplied by 10, time multiplied by | 1,000 | 100 | 10+ | 10 |

**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>10²</td>
<td>1.7 minutes</td>
</tr>
<tr>
<td>10³</td>
<td>17 minutes</td>
</tr>
<tr>
<td>10⁴</td>
<td>2.8 hours</td>
</tr>
<tr>
<td>10⁵</td>
<td>1.1 days</td>
</tr>
<tr>
<td>10⁶</td>
<td>1.6 weeks</td>
</tr>
<tr>
<td>10⁷</td>
<td>3.8 months</td>
</tr>
<tr>
<td>10⁸</td>
<td>3.1 years</td>
</tr>
<tr>
<td>10⁹</td>
<td>3.1 decades</td>
</tr>
<tr>
<td>10¹⁰</td>
<td>3.1 centuries</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Meters Per Second</th>
<th>Imperial Units</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^{-10}$</td>
<td>1.2 in / decade</td>
<td>Continental drift</td>
</tr>
<tr>
<td>$10^{-8}$</td>
<td>1 ft / year</td>
<td>Hair growing</td>
</tr>
<tr>
<td>$10^{-6}$</td>
<td>3.4 in / day</td>
<td>Glacier</td>
</tr>
<tr>
<td>$10^{-4}$</td>
<td>1.2 ft / hour</td>
<td>Gastro-intestinal tract</td>
</tr>
<tr>
<td>$10^{-2}$</td>
<td>2 ft / minute</td>
<td>Ant</td>
</tr>
<tr>
<td>1</td>
<td>2.2 mi / hour</td>
<td>Human walk</td>
</tr>
<tr>
<td>$10^2$</td>
<td>220 mi / hour</td>
<td>Propeller airplane</td>
</tr>
<tr>
<td>$10^4$</td>
<td>370 mi / min</td>
<td>Space shuttle</td>
</tr>
<tr>
<td>$10^6$</td>
<td>620 mi / sec</td>
<td>Earth in galactic orbit</td>
</tr>
<tr>
<td>$10^8$</td>
<td>62,000 mi / sec</td>
<td>1/3 speed of light</td>
</tr>
</tbody>
</table>

**Historical Quest for Speed**

- **Multiplication:** $a \times b$.
  - **Naïve:** 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:** $\gcd(a, b)$.

- **Naïve:** 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.

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;
}
```

insertionsort.c (see Sedgewick Program 6.1)
Profiling Insertion Sort Empirically

Use lcc "profiling" capability.
- Automatically generates a file prof.out that has frequency counts for each instruction.

```
Unix
$ lcc -b insertion.c item.c
$ a.out < unsorted1000.txt
$ bprint
```

```
prof.out

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;
}
```

Striking feature: HUGE numbers!

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.
  - ith iteration requires i - 1 compare and exchange operations
  - total = 0 + 1 + 2 + ... + N-1 = N (N-1) / 2

```
      E   F   G   H   I   J   D   C   B   A
```

```
unsorted  active  sorted
```

Best case.
- Elements in sorted order already.
  - ith iteration requires only 1 compare operation
  - total = 0 + 1 + 1 + ... + 1 = N - 1

```
      A   B   C   D   E   F   G   H   I   J
```

```
unsorted  active  sorted
```

Average case.
- Elements are randomly ordered.
  - ith 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

```
      A   D   E   F   H   J   C   B   I   G
```

```
unsorted  active  sorted
```
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: \( N (N - 1)/2 \).

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).

<table>
<thead>
<tr>
<th></th>
<th>computer</th>
<th>thousand</th>
<th>million</th>
<th>billion</th>
</tr>
</thead>
<tbody>
<tr>
<td>home</td>
<td>instant</td>
<td>2.8 hours</td>
<td>317 years</td>
<td></td>
</tr>
<tr>
<td>super</td>
<td>instant</td>
<td>1 second</td>
<td>1.6 weeks</td>
<td></td>
</tr>
</tbody>
</table>

Insertion Sort (\( N^2 \))

Estimating the Running Time

Easier alternative.
(i) Analyze asymptotic growth.
(ii) For medium N, run and measure time.
(iii) 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 asymptotically 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)

Jon von Neumann (1945)
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!

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.

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

- N log₂ N comparisons to sort ANY array of N elements.
  - even already sorted array!
### Profiling Mergesort Analytically

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

- \( T(N/2) \)
- \( T(N/2) \)
- \( T(N/4) \)
- \( T(N/4) \)
- \( T(N/4) \)
- \( T(N/4) \)
- \( T(2) \)
- \( T(2) \)
- \( T(2) \)
- \( T(2) \)
- \( T(2) \)
- \( T(2) \)

### Implementing Mergesort

**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--];
}
```

**mergesort (see Sedgewick Program 8.3)**

```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);
}
```

### Implementing Mergesort Empirically

**Mergesort prof.out**

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

**mergesort (see Sedgewick Program 8.3)**

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

**Striking feature:**

All numbers SMALL!

# comparisons

Theory \( N \log_2 N = 9,966 \)

Actual = 9,976
Sorting Analysis Summary

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

<table>
<thead>
<tr>
<th></th>
<th>computer</th>
<th>thousand</th>
<th>million</th>
<th>billion</th>
</tr>
</thead>
<tbody>
<tr>
<td>home</td>
<td>instant</td>
<td>2.8 hours</td>
<td>317 years</td>
<td></td>
</tr>
<tr>
<td>super</td>
<td>instant</td>
<td>1 second</td>
<td>1.6 weeks</td>
<td></td>
</tr>
</tbody>
</table>

Lesson 1: good algorithms are better than supercomputers.

How does quicksort fit into the picture?

Quicksort

Quicksort.
- 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 $m$

Profiling Quicksort Empirically

Profiling Quicksort Analytically

Precondition: file is randomly shuffled beforehand.
- or could partition on RANDOM element.

Average case running time.
- Roughly $2N \log_e N$ comparisons. (see COS 226 for analysis)
- Faster than mergesort.
  - lower cost of high-frequency instructions (see Knuth slide)

Check profile.
- $2N \log_e N$: 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)

Note: can take time proportional to $N^2$ in worst case.
- More likely that machine struck by lightning.
Sorting Analysis Summary

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

<table>
<thead>
<tr>
<th></th>
<th>computer</th>
<th>thousand</th>
<th>million</th>
<th>billion</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Insertion Sort ($N^2$)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>home</td>
<td>instant</td>
<td>2.8 hours</td>
<td>317 years</td>
<td></td>
</tr>
<tr>
<td>super</td>
<td>instant</td>
<td>1 second</td>
<td>1.6 weeks</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Mergesort (N log N)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>thousand</td>
<td>instant</td>
<td>1 sec</td>
<td>18 min</td>
<td></td>
</tr>
<tr>
<td></td>
<td>instant</td>
<td>instant</td>
<td>instant</td>
<td></td>
</tr>
</tbody>
</table>

|                | thousand | million | billion |
| **Quicksort (N log N)** |           |          |         |
| instant         | 0.3 sec  | 6 min    |         |
| instant         | instant  | instant  | instant |

Lesson 1: good algorithms are better than supercomputers.
Lesson 2: great algorithms are better than good ones.

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.

Computational Complexity

Framework to study efficiency of algorithms.
- UPPERBOUND = algorithm to solve the problem (worst-case).
- LOWERBOUND = proof that no algorithm can do better.
- OPTIMAL ALGORITHM: lower bound ~ upper bound.

Example 1: 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
- proof: see COS 226
- Optimal algorithm = mergesort.

Example 2: TSP.
- Upper bound = $N!$ (2$^N$ also possible)
- Lower bound = $N$
- Optimal algorithm = ask again in 50 years

Essence of computational complexity.
- Closing the gap.
Summary

How can I evaluate the performance of a proposed algorithm?
- Computational experiments.
- Analysis of algorithms.

What if it's not fast enough?
- Understand why.
  - complexity theory
- Use a faster computer.
  - performance improves incrementally
- Discover a better algorithm.
  - performance can improve dramatically
  - not always easy / possible to develop better algorithm

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);
```

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

int ITEMless(Item a, Item b) { return (a < b); }
void ITEMshow(Item a) { printf("%d\n", a); }
void ITEMswap(Item *pa, Item *pb) {
  Item t;
  t = *pa; *pa = *pb; *pb = t;
}
int ITEMscan(Item *pa) {
  return scanf("%d", pa);
}
```

Item Implementation

swap integers – need to use pointers
### Generic Sorting Program

```
#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;
}
```

### Profiling Quicksort Analytically

**Average case.**
- Assume partition element chosen at random and all elements are unique.
- Denote jth largest element by j.
- Probability that i and j (where j > i) are compared = \( \frac{2}{j-i+1} \)

Expected # of comparisons = \( \sum_{i<j} \frac{2}{j-i+1} = 2 \sum_{i=1}^{N} \sum_{j=i+1}^{N} \frac{1}{j} \leq 2N \sum_{j=1}^{N} \frac{1}{j} = 2N \ln N \)

### Comparison Based Sorting 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

\[
h \geq \log_2 (N!)
\geq \log_2 (N/e)^N
= N \log_2 N - N \log_2 e
= \Theta(N \log_2 N)
\]