Lecture T5: Analysis of Algorithm

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.

Historical Quest for Speed

Multiplication: $a \times b$.
- Naïve: add a to itself b times. $N^2$ steps
- Grade school. $N^2$ steps
- Divide-and-conquer (1962). $N^{1.58}$ steps
- Ingenuity (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

Complex multiplication: $(a + bi)(c + di) = x + yi$.
- Naïve: $x = ac - bd$, $y = bc + ad$. 4 multiplications
- Gauss (1800): 3 multiplications
  - $x_1 = (a + b)(c + d)$, $x_2 = ac, x_3 = bd$
  - $x = x_2 - x_3, y = x_1 - x_2 - x_3$

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 the gravitational interactions among N bodies.
  - Physicists want \( N = \# \) atoms in universe.
- Brute force method takes \( N^2 \) steps.
- Appel (1981) algorithm takes \( N \log N \) time and enables new research.

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 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);
int ITEMscan(Item *pa);
```

image
**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("%d\n", a);
}

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

---

**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!

**Unix**

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

**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]))
                <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.
  - \( i \)th iteration requires \( i - 1 \) compare and exchange operations
  - total = \( 0 + 1 + 2 + \ldots + N-1 = N(N-1)/2 \)

Best case.
- Elements in sorted order already.
  - \( i \)th iteration requires only 1 compare operation
  - total = \( 0 + 1 + 1 + \ldots + 1 = N - 1 \)

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

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 estimate by `lcc -S` (plus manuals).

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 $10^6$)

Sorting Case Study: mergesort

Insertion sort (brute-force)
Mergesort (divide-and-conquer)
- Divide array into two halves.

M E R G E S O R T M E
M E R G E S O R T M E
divide

M E R G E S O R T M E
M E R G E S O R T M E
divide

E E G M R S E M O R T
sort
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
  - \(T(N)\) = comparisons to mergesort array of \(N\) elements.

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

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

\[
T(N) = 2T(N/2) + N = 2(2T(N/4) + N/2) + N \\
= 4T(N/4) + 2N = 4(2T(N/8) + N/4) + 2N \\
= 8T(N/8) + 3N = 16T(N/16) + 4N \\
\ldots \\
= N T(1) + kN \\
= 0 + N \log_2 N
\]

How much space?

Can't do "in-place" like insertion sort.

Need extra 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

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.

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) <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>}
```

Striking feature: no HUGE numbers!
Profiling Quicksort Empirically

Profiling Quicksort Analytically

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

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

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

Can you find median in \( O(N) \) time?


Profiling Quicksort Analytically

Partition on median element.

Partition on rightmost element.

Partition on random element.

Check profile.
- \( 2N \log_2 N \): 13815 vs. 12372 (5708 + 6664).
- Running time for \( N = 100,000 \) about 1.2 seconds.
- How long for \( N = 1 \text{ million} \)?
  - 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.

Insertion Sort \( (N^2) \)

<table>
<thead>
<tr>
<th>Computer</th>
<th>thousand</th>
<th>million</th>
<th>billion</th>
</tr>
</thead>
<tbody>
<tr>
<td>home pc</td>
<td>instant</td>
<td>2 hour</td>
<td>310 years</td>
</tr>
<tr>
<td>super</td>
<td>instant</td>
<td>1 sec</td>
<td>1.6 weeks</td>
</tr>
</tbody>
</table>

Quicksort \( (N \log N) \)

<table>
<thead>
<tr>
<th>Computer</th>
<th>thousand</th>
<th>million</th>
<th>billion</th>
</tr>
</thead>
<tbody>
<tr>
<td>home pc</td>
<td>instant</td>
<td>0.3 sec</td>
<td>6 min</td>
</tr>
<tr>
<td>super</td>
<td>instant</td>
<td>instant</td>
<td>instant</td>
</tr>
</tbody>
</table>

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

Good algorithms are more powerful than supercomputers.
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.

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>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Best case complexity</td>
<td>$O(n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Average case complexity</td>
<td>$O(n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Already sorted</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Reverse sorted</td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Space</td>
<td>$O(n)$</td>
<td>$O(n)$</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: bubblesort, heapsort, shellsort, selection sort, shaker sort, radix sort, BST sort, solitaire sort, hybrid methods.
- Which one should I use?

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 = $O(n \log n)$ (mergesort).
  - quicksort usually faster, but mergesort never slow
- Lower bound = $O(n \log n - n \log n e)$
  (applies to any comparison-based algorithm).
  - Why?

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

Lecture T5: Extra Slides

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

Profiling Quicksort Analytically

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

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

\[
\log_2(N!) \geq \log_2(\frac{N}{e})^N = N \log_2 N - N \log_2 e = \Theta(N \log_2 N)
\]

Stirling’s formula