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.

Linear Growth

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

<table>
<thead>
<tr>
<th>N = 4</th>
<th>1 1 1 0</th>
<th>1 1 0 0 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 0 1 1</td>
<td>1 0 1 0 1</td>
</tr>
<tr>
<td>+</td>
<td>1 1 1 0</td>
<td>0 1 1 1 1</td>
</tr>
<tr>
<td></td>
<td>1 1 1 0</td>
<td>1 0 1 0 0</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>N = 8</th>
<th>1 1 1 1 1 1 0 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 1 0 1 0 1 0 1</td>
</tr>
<tr>
<td>+</td>
<td>0 1 1 1 1 1 0 1</td>
</tr>
<tr>
<td></td>
<td>1 0 1 0 1 0 0 1</td>
</tr>
</tbody>
</table>

Quadratic Growth

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

<table>
<thead>
<tr>
<th>N = 4</th>
<th>1 0 1 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 1 0 1</td>
</tr>
<tr>
<td></td>
<td>1 0 1 1</td>
</tr>
<tr>
<td></td>
<td>1 0 0 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>N = 8</th>
<th>1 1 0 1 0 1 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 1 0 1 1 1 0</td>
</tr>
<tr>
<td></td>
<td>1 1 0 1 1 1 1</td>
</tr>
<tr>
<td></td>
<td>0 0 0 1 1 1 1</td>
</tr>
</tbody>
</table>

2N reads
N^2 + 2N + 1 writes
N-1 adds on N-bit integers

N-1 odd parity operations
N-1 majority operations
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>Time to solve a problem of size</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<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>

Max size problem solved in one:
- second: 920
- minute: 3,600
- hour: 14,000
- day: 41,000

These are equivalent to:
- 1 million
- 49 million
- 2.4 trillion
- 50 trillion

N multiplied by 10, time multiplied by 10:
- $1,000$
- $100$
- $10^+$
- $10$

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>$10^1$</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>

... forever

$10^{21}$ age of universe

Meters Per Second | Imperial Units | Example |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></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>$10^{10}$</td>
<td>62,000 mi / sec</td>
<td>1/3 speed of light</td>
</tr>
</tbody>
</table>

Powers of 2

| $2^0$ | thousand |
| $2^{20}$ | million |
| $2^{30}$ | billion |

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.

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.

```c
typedef int Item;

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

```
#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);
}
```

---

## 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;
}
```

---

## sort.c (see Sedgewick 6.1)

Max number of items to sort.

- Read input.
- Call generic sort function.
- Print results.

---

# Insertion Sort Function

```
#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);
}
```

---

## insertionsort.c (see Sedgewick Program 6.1)

```
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!

## Insertion Sort prof.out

```
void insertionsort(Item a[], int left, int right) <1>{
    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;
}
```
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

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

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

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?

```mergesort```

divide

```mergesort```

sort

```mergesort```

merge
Profiling Mergesort Analytically

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

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

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

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

= \( N T(1) + kN \)
= \( 0 + N \log_2 N \)

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

mergesort (see Sedgewick Program 8.3)

```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);
}
```

merge (see Sedgewick Program 8.2)

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

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);
    }
}

Sorting Case Study: quicksort

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

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);
    }
}
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.
  - How do we partition efficiently?
    - \( N-1 \) comparisons
    - easy with auxiliary array
    - better solution: use no extra space!

Implementing Partition

```
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;
}
```

Profiling Quicksort Empirically

```
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);
}
```

Quicksort prof.out

```
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);
}
```

Quicksort prof.out (cont)

```
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))
            if (<right <= 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.

$$T(N) = \begin{cases} 0 & \text{if } N = 1 \\ 2T\left(\frac{N}{2}\right) + \frac{N}{2} & \text{otherwise} \\ \text{sorting both halves} \hspace{1cm} \text{partitioning} \end{cases}$$

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

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


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 $N = 1$ 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 ($N^2$)</th>
<th>Quicksort ($N \log N$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>home</td>
<td>thousand</td>
<td>million</td>
</tr>
<tr>
<td>super</td>
<td>instant</td>
<td>2.8 hours</td>
</tr>
<tr>
<td></td>
<td>instant</td>
<td>1 second</td>
</tr>
</tbody>
</table>

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**

<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>

**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?

**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.
- Discover a better algorithm.
  - performance can improve dramatically
  - not always easy / possible to develop better algorithm
### 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 $i^{th}$ largest element by $i$.
- Probability that $i$ and $j$ (where $j > i$) are compared = $\frac{2}{j - i + 1}$

**Expected # of comparisons**

$$
\sum_{i<j} \frac{2}{j - i + 1} = 2 \sum_{i=1}^{N} \sum_{j=i+1}^{N} \frac{1}{j} \\
\leq 2N \sum_{j=1}^{N} \frac{1}{j} \\
= 2N \ln N
$$

---

#### Comparison Based Sorting Lower Bound

Decision Tree of Program
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

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

Stirling’s formula