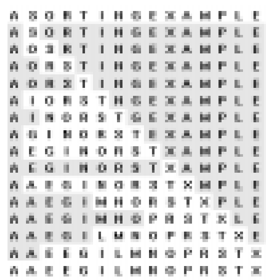
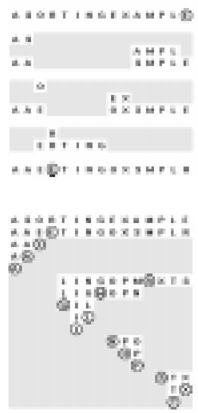
Lecture T5: Analysis of Algorithm





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.

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.

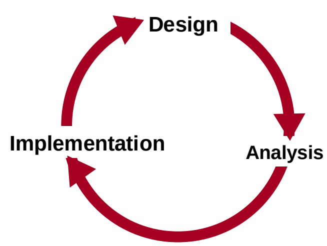
- Find a method to solve the problem.

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.

name
Hauser
Hong
Hsu
Hayes
Haskell
Hanley
Hornet
Hill

→

name
Hanley
Haskell
Hauser
Hayes
Hill
Hong
Hornet
Hsu

Case Study: Sorting

Sorting problem:

- Given an array of N integers, rearrange them so that they are in increasing order.

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.

ITEM.h

```
typedef int Item;

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

return 1 if a < b (→ ITEMless)

swap 2 Items (→ ITEMswap)

Item Implementation

item.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

sort.c (see Sedgewick 6.1)

```
#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++)
     ITEMprint(a[i]);
  return 0;
}
```

Max number of items to sort.

Read input.

Call generic sort function.

Print results.

Insertion Sort Function

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!

Unix

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

prof.out

```
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 + . . . + N-1 = N (N-1) / 2

E	F	G	H	I	J	D	C	B	A

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.

Best case.

- Elements in sorted order already.
  - i^{th} 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

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.

Average case.

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

B	E	F	R	T	U	O	R	C	E

unsorted   active   sorted

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

Donald Knuth

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.

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**

E E E G M M O R R S T   **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 array of N elements.

$$T(N) = \begin{cases} 0 & \text{if } N = 1 \\ \underbrace{2T(N/2)}_{\text{sorting both halves}} + \underbrace{N}_{\text{merging}} & \text{otherwise} \end{cases}$$

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

$$\begin{aligned} T(N) &= 2\,T(N/2) + N = 2\,(2\,T(N/4) + N/2) + N \\ &= 4\,T(N/4) + 2N = 4\,(2\,T(N/8) + N/4) + 2N \\ &= 8\,T(N/8) + 3N \\ &= 16\,T(N/16) + 4N \\ &\;\;\vdots \\ &= N\,T(1) + k\,N \\ &= 0 + N \log_2 N \end{aligned}$$

Profiling Mergesort Analytically

How long does mergesort take?

- Bottleneck = merging (and copying).
  - merging two files of size N/2 requires N comparisons
- N log2 N comparisons to sort *any* array of N elements.
  - even already sorted array!

How much space?

Implementing Mergesort

mergesort (see Sedgewick Program 8.3)

```
Item aux[MAXN];          ← assume scratch array

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)

```
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--];
}
```

← copy to temporary array

← merge two sorted sequences

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.

partitioning element ↓

Q	U	I	C	K	S	O	R	T	I	S	C	O	O	**L**
I	C	K	I	C	**L**	Q	U	S	O	R	T	S	O	O
C	C	I	I	K		O	O	O	Q	R	S	S	T	U

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.

quicksort.c (see Sedgewick Program 7.1)

```
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
  - no extra space!

Implementing Partition

partition (see Sedgewick Program 7.2)

```
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;
}
```

- find element on left to swap
- look for element on right to swap, but don't run off end
- pointers cross
- swap partition element

Profiling Quicksort Empirically

prof.out

```
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

prof.out (cont)

```
int partition(Item a[], int left, int right) <668>{
  int i = <668>left-1, j = <668>right;
  Item swap, p = <668>a[right];

  for(<668>;<1678>;<1678>) {
    while (<5708>ITEMless(a[++i], p))
       <3362>;
    while (<6664>ITEMless(p, a[--j]))
       if (<4495>j == left)
         <177>break;
    if (<2346>i >= j)
       <668>break;
    <1678>ITEMswap(&a[i], &a[j]);
  }
  <668>ITEMswap(&a[i], &a[right]);
  return <668>i;
<668>}
```

Striking feature: no HUGE numbers!

Profiling Quicksort Analytically

Intuition.

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

$$T(N) = \begin{cases} 0 & \text{if } N = 1 \\ \underbrace{2T(N/2)}_{\text{sorting both halves}} + \underbrace{N}_{\text{partitioning}} & \text{otherwise} \end{cases}$$

If N is a power of 2.

$\Rightarrow$ **T(N) = N $\log_2$ N**

How can we find median in O(N) time?

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 =

$$\begin{aligned} \sum_{i<j} \frac{2}{j-i+1} &= 2 \sum_{i=1}^{N} \sum_{j=2}^{i} \frac{1}{j} \\ &\le 2N \sum_{j=1}^{N} \frac{1}{j} \\ &\approx 2N \int_1^N \frac{1}{j} \\ &= 2N \ln N \end{aligned}$$

Profiling Quicksort Analytically

Average case. (roughly 2 N ln N)

- Check with profile: 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)
  - on arizona, 13.7

Best case. (N $\log_2$N)

- Always partition on median.

Worst case. (N^2 / 2)

- Novices beware: could be slow for some inputs.
- Already sorted file: takes $N^2/2$ + N/2 comparisons.
  - all partitions are degenerate

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

Bob Tarjan, et al (1973)

Sorting Analysis Summary

Running time estimates:

- Home pc executes 10^8 comparisons/second.
- Supercomputer executes 10^{12} comparisons/second.

Insertion Sort (N^2)

computer	thousand	million	billion
home pc	instant	2 hour	310 years
super	instant	1 sec	1.6 weeks

Quicksort (N lg N)

thousand	million	billion
instant	0.3 sec	6 min
instant	instant	instant

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

Good algorithms are more powerful than supercomputers.

Sorting Analysis Summary

Comparison of Different Sorting Algorithms

Attribute	insertion	quicksort	mergesort
Worst case complexity	N^2	N^2	$N \log_2 N$
Best case complexity	N	$N \log_2 N$	$N \log_2 N$
Average case complexity	N^2	$N \log_2 N$	$N \log_2 N$
Already sorted	N	N^2	$N \log_2 N$
Reverse sorted	N^2	N^2	$N \log_2 N$
Space	N	N	2 N
Stable	yes	no	yes

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.

Computational Complexity

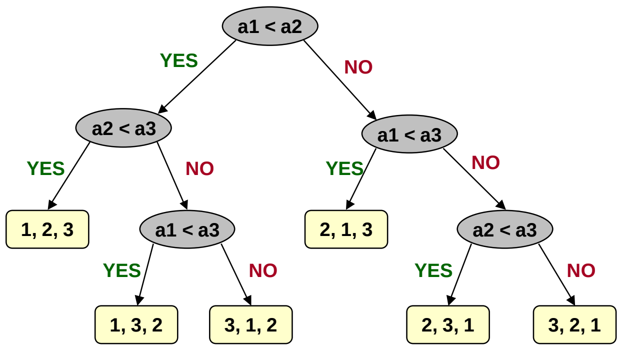
Framework to study efficiency of algorithms.

- 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?

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

$$\begin{aligned} h &\geq \log_2 (N!) \\ &\geq \log_2 (N/e)^N \quad \text{(Stirling's formula)} \\ &= N \log_2 N - N \log_2 e \\ &= \Theta(N \log_2 N) \end{aligned}$$

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

