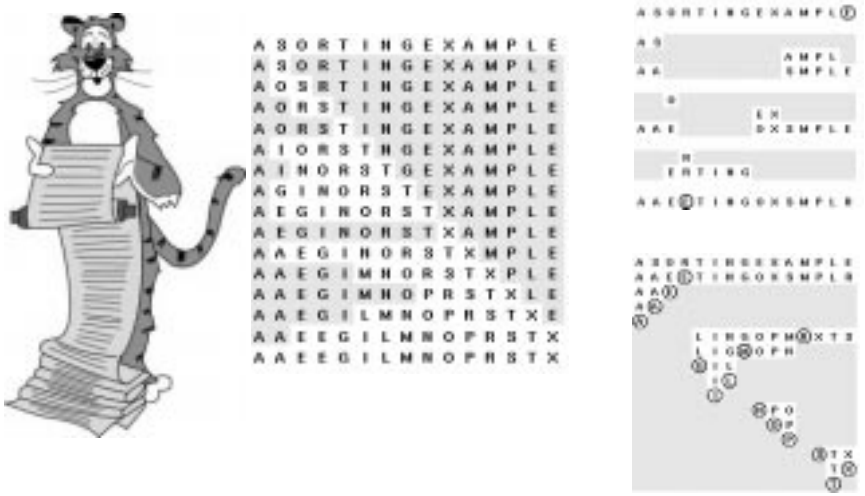
Lecture T4: Analysis of Algorithms



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: k N for some constant k.

```
  1 1 1 0          1 1 1 1 1 1 0 1
    1 0 1 1          1 1 0 1 0 1 0 1
+   1 1 1 0      +   0 1 1 1 1 1 0 1
-----------      -----------------
  1 1 0 0 1        1 0 1 0 1 0 0 1 0
```

N = 4 N = 8

2N	read operations
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: k N^2 for some constant k.

N = 4

```
    1 0 1 1
  * 1 1 0 1
  ---------
    1 0 1 1
  0 0 0 0
+ 1 0 1 1
+ 1 0 1 1
---------------
1 0 0 0 1 1 1 1
```

N = 8

```
      1 1 0 1 0 1 0 1
    * 0 1 1 1 1 1 0 1
    -----------------
      1 1 0 1 0 1 0 1 0
    0 0 0 0 0 0 0 0
  + 1 1 0 1 0 1 0 1 0
  + 1 1 0 1 0 1 0 1 0
  + 1 1 0 1 0 1 0 1 0
  + 1 1 0 1 0 1 0 1 0
  + 0 0 0 0 0 0 0 0
-------------------------
0 1 1 0 1 0 0 0 0 0 0 0 1 0
```

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

Why Does It Matter?

Run time in nanoseconds -->		1.3 N^3	10 N^2	47 N $\log_2 N$	48 N
Time to solve a problem of size	1000	1.3 seconds	10 msec	0.4 msec	0.048 msec
	10,000	22 minutes	1 second	6 msec	0.48 msec
	100,000	15 days	1.7 minutes	78 msec	4.8 msec
	million	41 years	2.8 hours	0.94 seconds	48 msec
	10 million	41 millennia	1.7 weeks	11 seconds	0.48 seconds
Max size problem solved in one	second	920	10,000	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

Seconds	Equivalent
1	1 second
10	10 seconds
10^2	1.7 minutes
10^3	17 minutes
10^4	2.8 hours
10^5	1.1 days
10^6	1.6 weeks
10^7	3.8 months
10^8	3.1 years
10^9	3.1 decades
10^{10}	3.1 centuries
. . .	forever
10^{21}	age of universe

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

Powers of 2		
	2^{10}	thousand
	2^{20}	million
	2^{30}	billion

Historical Quest for Speed

Multiplication: a × 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

N = # bits in binary representation of a, b

step = integer division

Greatest common divisor: gcd(a, b).

- Naïve: factor a and b, then find gcd(a, b). $2^{\sqrt{N}}$ 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.
- Appel (1981). N log N steps, 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: N^2 steps.
- Runge-König (1924), Cooley-Tukey (1965).
  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.

name
Hauser
Hong
Hsu
Hayes
Haskell
Hanley
Hornet
Hill

name
Hanley
Haskell
Hauser
Hayes
Hill
Hong
Hornet
Hsu

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

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.

Unix

```
% lcc -b insertion.c item.c
% a.out < unsorted1000.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>}
```

Striking feature: HUGE numbers!

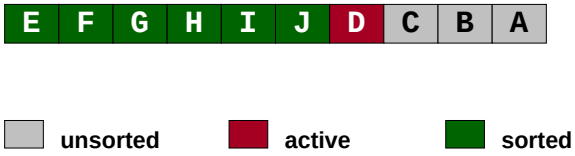
Profiling Insertion Sort Analytically

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.
  - i^{th} iteration requires i - 1 compare and exchange operations
  - total = 0 + 1 + 2 + . . . + N-1 = N (N-1) / 2



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

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: 249,750 vs. 256,313

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).

Donald Knuth

Insertion Sort (N^2)

computer	thousand	million	billion
home	instant	2.8 hours	317 years
super	instant	1 second	1.6 weeks

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)

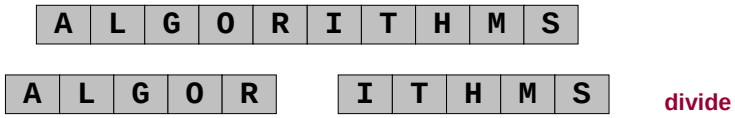
Jon von Neumann (1945)

Sorting Case Study: mergesort

Insertion sort (brute-force)

Mergesort (divide-and-conquer)

- Divide array into two halves.

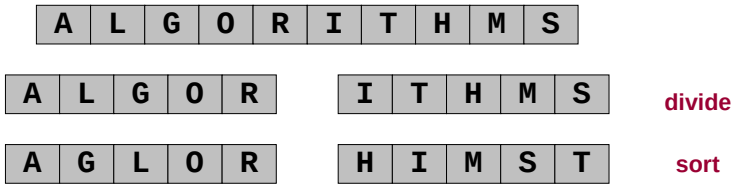


Sorting Case Study: mergesort

Insertion sort (brute-force)

Mergesort (divide-and-conquer)

- Divide array into two halves.
- Sort each half separately. How do we sort half size files?

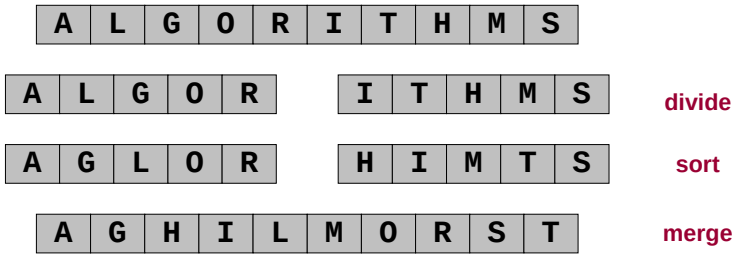


Sorting Case Study: mergesort

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?

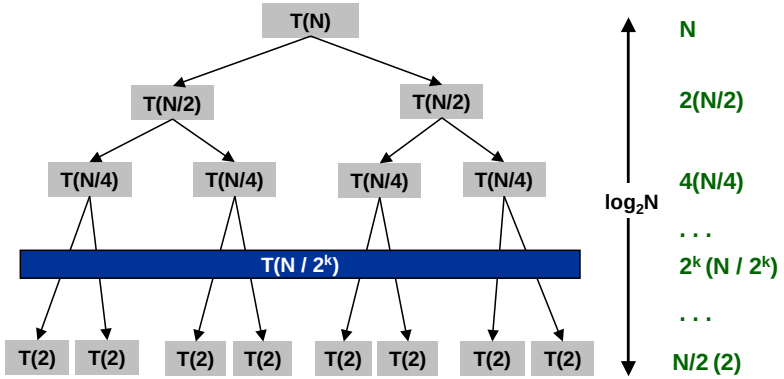
- Bottleneck = merging (and copying).
  - merging two files of size N/2 requires N comparisons
- T(N) = comparisons to mergesort N elements.

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

- N $\log_2$ N comparisons to sort ANY array of N elements.
  - even already sorted array!

Profiling Mergesort Analytically

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



N

2(N/2)

4(N/4)

. . .

$2^k (N / 2^k)$

. . .

N/2 (2)

$N \log_2 N$

Implementing Mergesort

mergesort (see Sedgewick Program 8.3)

```
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);
}
```

uses scratch array

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

Profiling Mergesort Empirically

Mergesort prof.out

```
void merge(Item a[], int left, int mid, int right) <999>{
  int i, j, k;
  for (<999>i = mid+1; <6043>i > left; <5044>i--)
     <5044>aux[i-1] = a[i-1];
  for (<999>j = mid; <5931>j < right; <4932>j++)
     <4932>aux[right+mid-j] = a[j+1];
  for (<999>k = left; <10975>k <= right; <9976>k++)
     if (<9976>ITEMless(aux[i], aux[j]))
        <4543>a[k] = aux[i++];
     else
        <5433>a[k] = aux[j--];
<999>}

void mergesort(Item a[], int left, int right) <
  int mid = <1999>(right + left) / 2;
  if (<1999>right <= left)
     return<1000>;
  <999>mergesort(a, aux, left, mid);
  <999>mergesort(a, aux, mid+1, right);
  <999>merge(a, aux, left, mid, right);
<1999>}
```

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.

Insertion Sort (N^2)

computer	thousand	million	billion
home	instant	2.8 hours	317 years
super	instant	1 second	1.6 weeks

Mergesort (N log N)

thousand	million	billion
instant	1 sec	18 min
instant	instant	instant

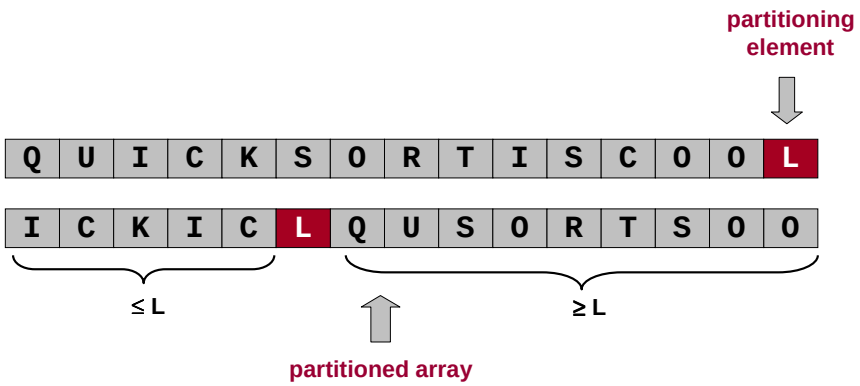
Lesson 1: good algorithms are better than supercomputers.

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

Quicksort 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];

  <668>while<1678>(<1678>1) {
     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: All numbers SMALL!

Profiling Quicksort Analytically

Precondition: file is randomly shuffled beforehand.

- or could partition on RANDOM element.

Average case running time.

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

Check profile.

- 2 N $\log_e$ 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.

Insertion Sort (N^2)

computer	thousand	million	billion
home	instant	2.8 hours	317 years
super	instant	1 second	1.6 weeks

Mergesort (N log N)

thousand	million	billion
instant	1 sec	18 min
instant	instant	instant

Quicksort (N log N)

thousand	million	billion
instant	0.3 sec	6 min
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.

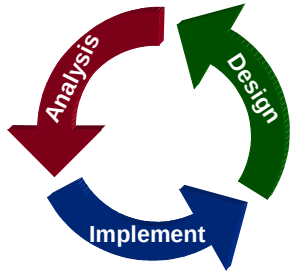
- Find a method to solve the problem.

Analysis.

- Evaluate its effectiveness and predict theoretical performance.

Implementation.

- Write actual code and test your theory.



Computational Complexity

Framework to study efficiency of algorithms.

- UPPER BOUND = algorithm to solve the problem (worst-case).
- LOWER BOUND = 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.

Computational Complexity

Framework to study efficiency of algorithms.

- UPPER BOUND = algorithm to solve the problem (worst-case).
- LOWER BOUND = proof that no algorithm can do better.
- OPTIMAL ALGORITHM: lower bound ~ upper bound.

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

Lecture T4: Supplemental Notes

Generic Item to Be Sorted

Define generic Item type to be sorted.

- Associated operations:
  - less, show, swap, rand
- Example: integers.

return 1 if a < b

swap 2 Items

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);
```


Item Implementation

swap integers – need to use pointers

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

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

Generic Sorting Program

Max number of items to sort.

Read input.

Call generic sort function.

Print results.

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


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 =

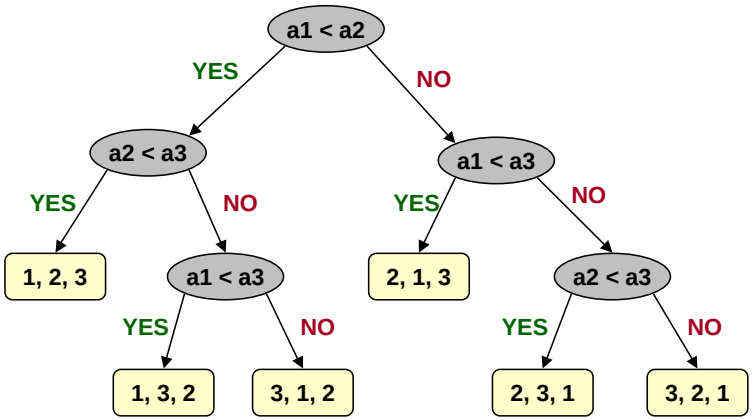
$$\sum_{i<j} \frac{2}{j-i+1} = 2 \sum_{i=1}^{N} \sum_{j=2}^{i} \frac{1}{j}$$

$$\leq 2N \sum_{j=1}^{N} \frac{1}{j}$$

$$\approx 2N \int_{1}^{N} \frac{1}{j}$$

$$= 2N \ln N$$

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

$$h \geq \log_2(N!)$$

$$\geq \log_2(N/e)^N$$

Stirling's formula

$$= N \log_2 N - N \log_2 e$$

$$= \Theta(N \log_2 N)$$