Sorting Concluded; Dynamic Programming

April 4–5, 2012
CS 60: Principles of Computer Science

Assignment 9 (Big-O) due Monday, April 9.

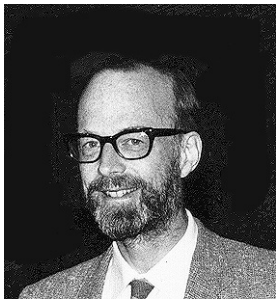
Sorting Algorithms so far

Given an array of n values:

Mergesort	$O(n \log n)$	worst-case
Insertion Sort	$O(n^2)$	worst-case
	$O(n)$	best-case
Minsort/Selection Sort	$O(n^2)$	worst-case
Stoogesort	$O(n^{2.7095})$	worst-case
cs5sort	$O(n^3)$	worst-case
Prologsort	$O(n!)$	worst-case
Bogosort	$O(n!)$	expected

Quicksort

Invented by Tony Hoare. (He also invented `null`, and has apologized for it).



1. Pick a "pivot" element $\mathbf{p}$.
2. *Partition* input into two parts ($< \mathbf{p}$ and $\geqslant \mathbf{p}$)
3. Recursively quicksort the two parts.

3 4 6 2 0 1 3 7

Analysis

1. What is the recurrence for Quicksort in the *best* case? Solution?

$$\begin{aligned} T(1) &= 1 \\ T(n) &= n + 2T(n/2) \qquad (n > 1) \end{aligned}$$

$$T(n) = O(n \log n)$$

2. What is the recurrence in the *worst* case? Solution?

$$\begin{aligned} T(1) &= 1 \\ T(n) &= n + T(1) + T(n-1) \qquad (n > 1) \end{aligned}$$

$$T(n) = O(n^2)$$

Common Sorting Algorithms

Given an array of n values:

Mergesort	$O(n \log n)$	worst-case
Quicksort	$O(n \log n)$	best-case
	$O(n^2)$	worst-case
	$O(n \log n)$	expected case (randomly-chosen pivot)
Insertion Sort	$O(n^2)$	worst-case
	$O(n)$	best-case

CS 70 discusses *Heapsort*, which is $O(n \log n)$ worst-case.

Which $O(n \log n)$ algorithm is best?

It depends. Here are some measurements from 2004:

Measurements

Now given all this information its a bit hard to see which of these should really be fastest in real life. That leaves us with one final possibility -- actually try it out in practice. Below are the results from just such a test:

	Athlon XP 1.620Ghz				Power4 1Ghz
	Intel C/C++ /O2 /G6 /Qaxi /Qxi /Qip	**WATCOM C/C++** /otexan /6r	**GCC** -O3 -march=athlon-xp	**MSVC** /O2 /Ot /Og /G6	**CC** -O3
Heapsort	2.09	4.06	4.16	4.12	16.91
Quicksort	2.58	3.24	3.42	2.80	14.99
Mergesort	3.51	4.28	4.83	4.01	16.90

Data is time in seconds taken to sort 10000 lists of varying size of about 3000 integers each. Download test here

http://www.azillionmonkeys.com/qed/sort.html

Can we do better?

Theorem

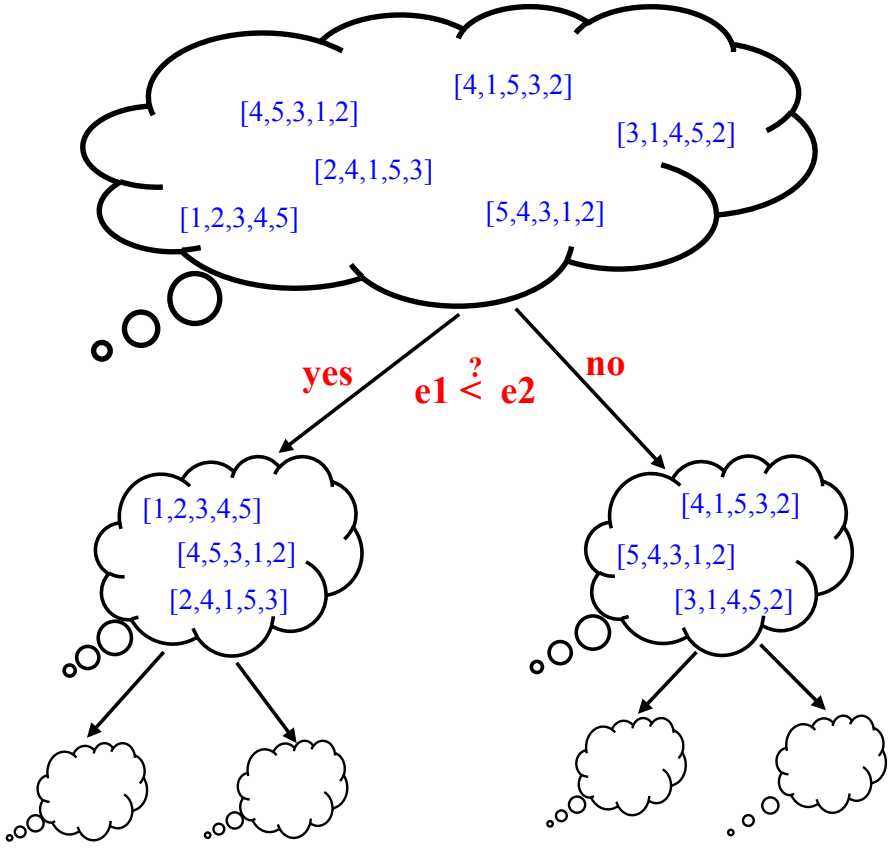
No algorithm based on comparisons can sort n *items in better than* $O(n \log n)$ *worst-case.*

Key insights:

- ✓ Given n distinct values, there are $n!$ possible permutations that might be our input.
- ✓ Sorting is equivalent to determining *which* permutation we started with.

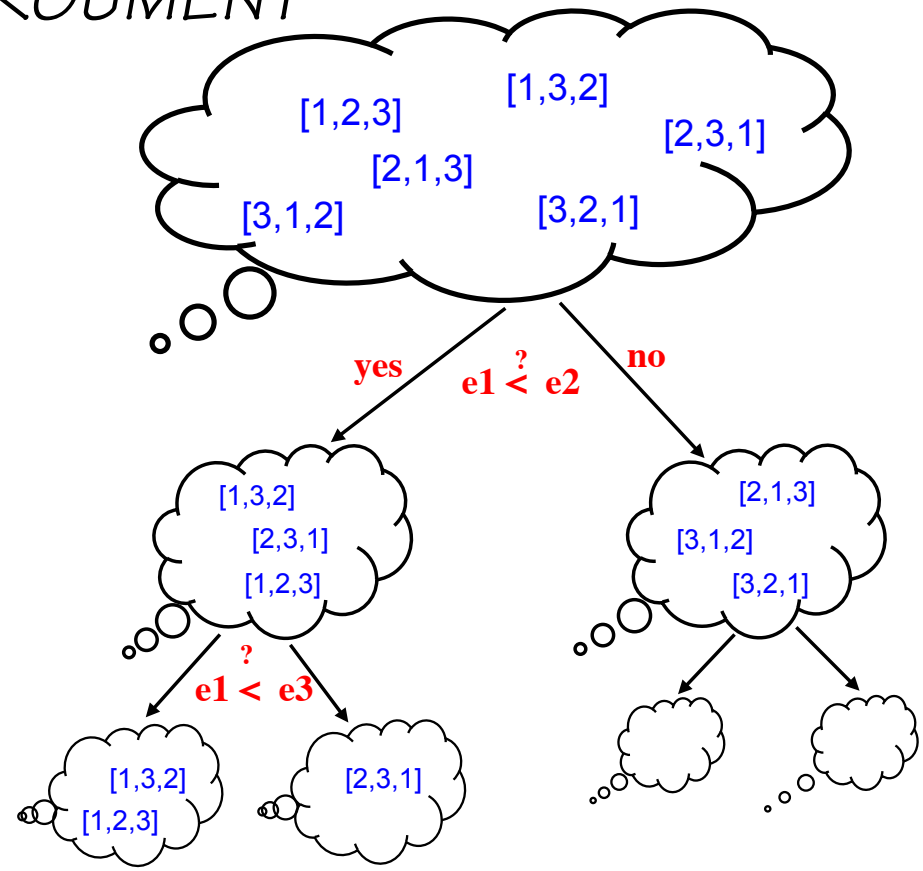
Determining Permutations

Each comparison lets us keep some possible permutations and rule out others.



An Adversarial Argument

The Adversary



If we're unlucky (or the input is chosen evilly), each comparison will rule out no more than *half* of the possibilities.

How many comparisons, to eliminate all but one possibility?

Stirling's Formula

$$\ln(n!) = n \ln n - n + O(\log n)$$

Hence the number of comparisons required is $O(n \log n)$.

Can we do better?

Yes…if you make extra assumptions about the data.

Bucket Sort: Suppose we know the inputs are all in the range $1..k$.

1. Allocate k buckets;
2. Put each item in the appropriate bucket;
3. Scan through the buckets in order.

Running time: $O(n + k)$ or ($O(n)$ if k is constant)

Radix Sort

Suppose the inputs are n *sequences* with k items each:

- ✓ Bucket-sort based the last item
- ✓ Bucket-sort on the next-to-last item
- ✓ ⋮
- ✓ Bucket-sort on the first item

FOR	FOR	BAR	BAR
BOX	BAR	FAR	BOX
FOX	FAR	FOR	FAR
BAR	BOX	BOX	FOR
FAR	FOX	FOX	FOX

worst-case: $O(nk)$.

Fibonacci Numbers

The Fibonacci numbers are easy to compute!

```
def fib(n):
    if n == 0:
       return 0
    elif n == 1:
       return 1
    else:
       return fib(n-1) + fib(n-2)
```

Unfortunately, this code has running time of about $O(1.6^n)$.
Why so bad? We repeatedly call the function with the same inputs

Idea 1: "Memoizing"

Keep track of all questions and answers.
If we see a repeat, immediately return the answer.

```
memopad = {0 : 0, 1 : 1}

def fib(n):
    if memopad.has_key(n):
        return memopad[n]
    else:
        answer = fib(n-1) + fib(n-2)
        memopad[n] = answer
        return answer
```

Idea 2: "Dynamic Programming"

Figure out ahead of time which calls will be needed.
Do them in a clever order to avoid checking

```
def fib(n):
  table = {0 : 0, 1 : 1}

  for i in range(2, n+1):
     table[i] = table[i-1] + table[i-2]

  return table[n]
```

The Knapsack Problem

Suppose we have n kinds of items.

- ✓ They have value (or utility) $v_1, \ldots, v_n$
- ✓ Each has weight (or size or cost) $w_1, \ldots, w_n$.

What should we choose, if the maximum total weight is W?

Knapsack Example

Suppose "a friend" can consume up to 13 units of candy.

- ✓ What's the maximum possible "value"?
- ✓ Does "greed" work?
- ✓ Does "use it or lose it" work?

$v_1 = 100$
$w_1 = 2$

$v_2 = 120$
$w_2 = 3$

$v_3 = 230$
$w_3 = 5$

$v_4 = 560$
$w_4 = 7$

$v_5 = 675$
$w_5 = 9$

Dynamic Programming Solution

Create a table of maximum value, indexed by total weight

Fill it in "bottom up"

The last entry is your answer!

$v_1 = 100$
$w_1 = 2$

$v_2 = 120$
$w_2 = 3$

$v_3 = 230$
$w_3 = 5$

$v_4 = 560$
$w_4 = 7$

$v_5 = 675$
$w_5 = 9$