lecture 21

- lower bound for comparison based sorting
- linear time sorting (not comparison-based)
  - counting sort
  - bucket sort
- more probability (independent events)

Comparison based Sorting

The sorting algorithms you have seen are all based on pairwise comparisons between elements

$O(n^2)$	$O(n \log n)$
• insertion sort	• heapsort
• selection sort	• mergesort
• bubble sort	• quick sort

Q: Is it possible to have a comparison based sorting algorithm that takes $O(n)$ time?

A: No. We will show that all comparison based sorting algorithms take at least $\sim n \log n$ time on some input.

Recall from COMP 250

$O(\ )$ asymptotic upper bound

$\Omega(\ )$ asymptotic lower bound

$\Theta(\ )$ asymptotic upper & lower bound

http://www.cim.mcgill.ca/~langer/250/16-notBigO.pdf

COMP 250/251 are mostly concerned with $O(\ )$

But Ω and Θ are very commonly used too.

To motivate the argument, lets think about a problem where you need to ask a certain number of questions to solve the problem - that is, you face an "information theoretic" lower bound.

The game "20 questions"

http://en.wikipedia.org/wiki/Twenty_Questions

- Player A thinks of an object
- Player B asks yes/no questions about the object
- Good strategy for player B is to ask questions that partition the set of possible solutions into two equal size sets

How do computer scientists play 20 questions

A: "I am thinking of a number between 0 and $2^{20} - 1$."

http://www.cim.mcgill.ca/~langer/250/2-binary.pdf

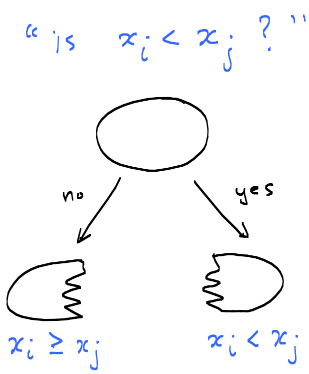
B's strategy:

Binary search! Essentially B would ask about each bit in the binary representation of the number.

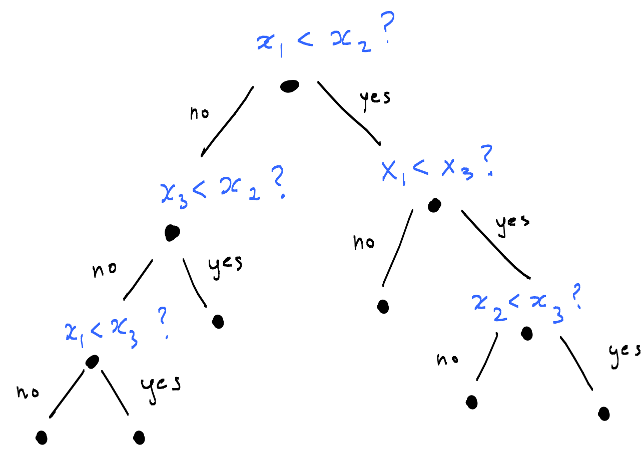
How is "m questions" relevant to comparison-based sorting?

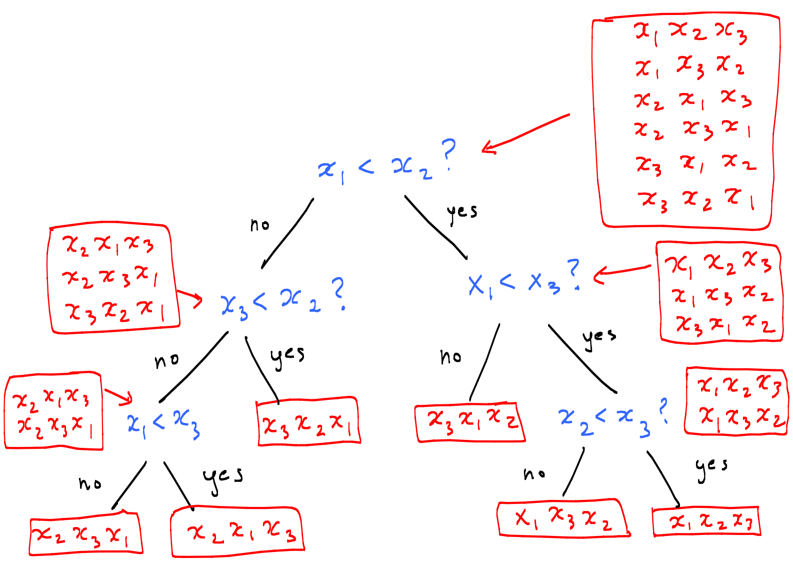
Each comparison ("is $x_i < x_j$?") partitions the set of possible solutions into two sets: those in which $x_i < x_j$ and those in which $x_i \geq x_j$, and the answer eliminates one of the two sets.

Suppose the input is of size n. There are n! possible orderings. "Sorting" means figuring out which one.



Example: Sort $(x_1\ x_2\ x_3)$

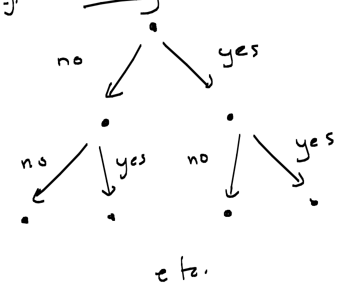




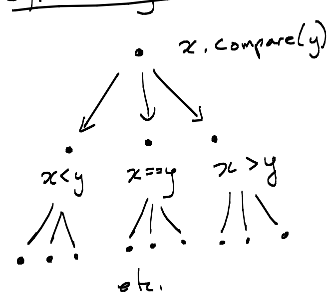
"Decision Tree"

Each (non-leaf) vertex is a question. The children of each vertex are the answers.

e.g. binary tree

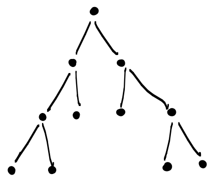


e.g. ternary tree



For the problem of sorting n numbers $\{x_1, x_2 \ldots x_n\}$, *any* comparison based sorting algorithm defines a binary decision tree with $n!$ leaves.

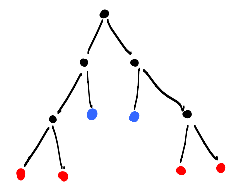
example: $n = 3$



Note:

- tree depends on n and on algorithm
- one leaf per problem instance
- path length = number of comparisons

For any algorithm and hence, for any decision tree, the best / worst / average case in terms of number of comparisons is the shortest / longest / average path length in the decision tree.

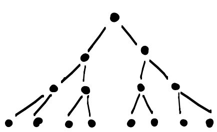


The worst case (max number of questions) is the length of the longest path in the decision tree, i.e. the height h of the tree.

Claim (easy to see)

Suppose a binary tree of height h has ℓ leaves.

Then, $\ell \leq 2^h$. Equivalently, $\log_2 \ell \leq h$



$h = 3$
$\ell = 8$

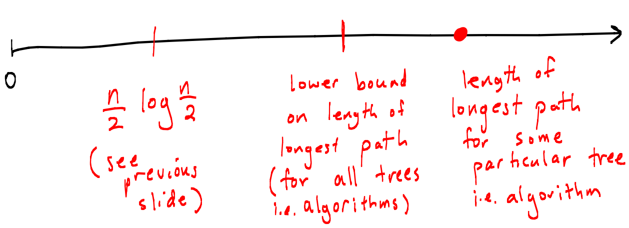
For any comparison-based sorting algorithm, the decision tree has $\ell = n!$ leaves. Thus, the length of the longest path (height of tree)

$$\geq \log(n!)$$
$$= \log(n \cdot (n-1)(n-2) \cdots 1)$$
$$> \log\Big(\underbrace{\tfrac{n}{2} \cdot \tfrac{n}{2} \cdots \tfrac{n}{2}}_{\frac{n}{2} \text{ times}} \cdot (\tfrac{n}{2}-1)(\tfrac{n}{2}-2) \cdots 2 \cdot 1\Big)$$
$$> \frac{n}{2} \log \frac{n}{2}$$

(lower bound on length of longest path)

For comparison based sorting of n elements, the "worst case" requires at least $\frac{n}{2} \log \frac{n}{2}$ comparisons.

i.e. the worst case is $\Omega(n \log n)$.



lecture 21

- lower bound for comparison-based sorting algorithms

- linear time sorting (not comparison-based)
  - counting sort
  - bucket sort
- more probability (independent events)

Linear time sorting algorithms impose more constraints on the input

- range of values (e.g. "counting sort")
- probability of elements (e.g. "bucket sort")

Counting Sort

Suppose inputs $x_1\ x_2 \ldots x_n$ are integers in range $\{1, 2, \ldots r\}$ and we allow for repeating elements.

Claim:

We can sort the x's in time and space $O(n+r)$.

e.g. if $r = n$, then we can sort in time $O(n)$.

Example : $r = 7$, $n = 11$

$x[\,] = [5, 1, 3, 3, 6, 5, 1, 6, 2, 5, 5]$ unsorted input

count

1	2
2	1
3	2
4	0
5	4
6	2
7	0

$\rightarrow$ [1, 1, 2, 3, 3, 5, 5, 5, 5, 6, 6] sorted output

```
for i = 1 to r
    count[i] = 0

for j = 1 to n
    count[ x[j] ] ++

for i = 1 to r {
        for k = 1 to count[i]
            print i
}
```

What is the disadvantage of Countsort?

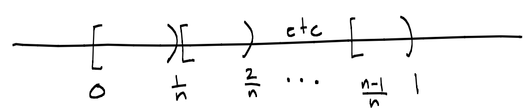
For many problems:

- r is very large
  e.g. sorting students by ID number
- the $x[\,]$ might not be integers.
  e.g. double, string

Bucket Sort

Suppose we have n doubles x_j in $[0,1)$ where $j = 1$ to n.

We consider intervals $\left[\frac{i-1}{n}, \frac{i}{n}\right)$ where $i = 1, \ldots n$



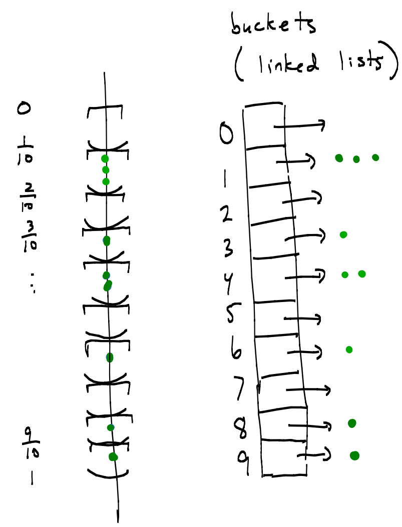
Put each x_j into its interval / bucket.

Then sort the elements in each bucket.

The worst case is that all n of the x_j will go into the same bucket. If we use an $O(n^2)$ algorithm for sorting, then bucketsort is $O(n^2)$.

The power of bucketsort (similar to hashing) comes from the assumption about the input, namely:
Assume each x_j is equally likely to fall in any of the buckets.

Example: $n = 10$



This is similar to a HashSet data structure (array of linked lists) except here there is no hash function.

```
//    0 ≤ x_i < 1   for all i

for  i = 1 to n
     bucket[i] = empty list
for  j = 1 to n {
     add x[j] to bucket[ floor( x[j] * n) ]
}
for  i = 1 to n {
     sort bucket[ i ]    // insertion sort
     print out sorted bucket[i]
}
```

Example: $n = 10$

$x[\,] = [.3, .13, .47, .92, .19, .7, .82, .16, .44, .6]$
unsorted input

0			
1	.13, .19	sort →	.13, .19
2			
3	.3	sort →	.3
4	.47, .44, .42	sort →	.42, .44, .47
5			
6	.6	sort →	.6
7			
8	.82	sort →	.82
9	.92	sort →	.92

$x[\,] = [.13, .19, .3, .42, .44, .47, .6, .82, .92]$

Claim: if each x_j is equally likely to go into any bucket, then the *expected* running time of bucket sort is $O(n)$.

Note: here we are considering random inputs, not a randomized algorithm.

Proof:

Consider any particular bucket i.
Define random variables:

$$B_{ij} = \begin{cases} 1, & \text{if } x_j \text{ falls in bucket } i. \\ 0, & \text{otherwise} \end{cases}$$

$$N_i = \sum_{j=1}^{n} B_{ij}$$ the number of x_j in bucket i.

$$\mathcal{E} N_i = \mathcal{E} \sum_{j=1}^{n} B_{ij}$$

$$= \sum_{j=1}^{n} \mathcal{E} B_{ij} \quad \text{by linearity of expectation}$$

$$= \sum_{j=1}^{n} \left(1 \cdot \frac{1}{n} + 0 \cdot \left(1 - \frac{1}{n}\right)\right)$$

$$= \sum_{j=1}^{n} \frac{1}{n}, \quad \text{by assumption that } x_j \text{ is equally likely to fall in all buckets}$$

$$= n \cdot \frac{1}{n}$$

$$= 1, \quad \text{so expected number of elements in each bucket is 1.}$$

Suppose we use N_i^2 operations to sort the elements in bucket j. (e.g. insertion sort)

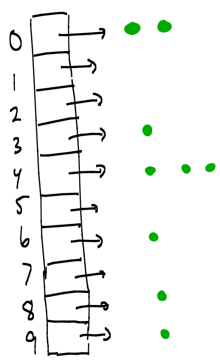
Then bucketsort uses $O\left(n + \sum_{i=1}^{N} N_i^2\right)$

n: putting elements into buckets

$\sum_{i=1}^{N} N_i^2$: sorting the buckets

What is $\mathcal{E} \sum_{i=1}^{n} N_i^2$?

Example



N_i	N_i^2
2	4
0	0
0	0
1	1
3	9
0	0
1	1
0	0
1	1
1	1

average is 1 (always)

average is $\frac{17}{10}$

$\mathcal{E} N_i^2$ for any $i = 1$ to N

$$= \sum_{k=0}^{n} k^2 \Pr(N_i = k)$$

one can show (using tools from MATH 323, namely N_i has a "binomial distribution")

$$< 2$$

Thus, $\mathcal{E} \sum_{i=1}^{n} N_i^2 = \sum_{i=1}^{n} \mathcal{E} N_i^2 < 2n$

Thus, bucketsort has expected running time $O(n)$.

Next class ... data compression.

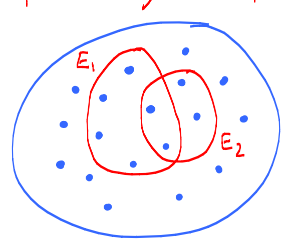
Today one final thing ...

a bit more probability

Independent Events

Definition: Given a sample space S and a probability distribution p, two events E_1 and E_2 are independent if

$$p(E_1 \wedge E_2) = p(E_1) \cdot p(E_2).$$



ASIDE: in a proper course on probability, (eg. MATH 323) the definition of "independent events" is based on the definition of something called "conditional probability".

I believe that going into the details here would be a distraction.

Instead, I ask you to rely on your intuition (dangerous!) in judging that a set of events is "independent" and to to apply the definition on the previous slide

Example

Suppose we flip a fair coin twice.

Let E_1 be the event that the first toss was a fail/tail $\{00, 01\}$ i.e. $\{TT, TH\}$.

Let E_2 be the event that the second toss was a fail/tail: $\{00, 10\}$.

Then, $E_1 \wedge E_2 = \{00\}$

Intuition: E_1 and E_2 are "independent."

means $\underbrace{p(E_1 \wedge E_2)}_{\frac{1}{4}} = \underbrace{p(E_1)}_{\frac{1}{2}} \cdot \underbrace{p(E_2)}_{\frac{1}{2}}$

Example (two slight variations)

Suppose we flip an unfair coin twice.

Let E_1 be the event that the first toss was a fail/tail $\{00, 01\}$.

Let E_2 be the event that the second toss was a head/success: $\{01, 11\}$.

Now, $E_1 \wedge E_2 = \{01\}$

"E_1 and E_2 are independent" means

$p(E_1 \wedge E_2) = \underbrace{p(E_1)}_{p_0} \cdot \underbrace{p(E_2)}_{1-p_0} = p_0(1-p_0)$

Recall from lecture 19:

Let X be the number of coin tosses until we get a success.

The coin tosses are independent:

$$
\begin{aligned}
&Pr(X = i) \\
&= p(\text{fail on tosses } 1 \text{ to } i-1) \cdot p(\text{success on toss } i) \\
&= p(\text{fail on toss } 1) \cdot p(\text{fail on toss } 2) \cdot \ldots\ p(\text{fail on toss } i-1) \cdot p(\text{success on toss } i) \\
&= p_0^{\,i-1}(1-p_0)
\end{aligned}
$$

What is the expected number of coin tosses of an unfair coin until we get a success?

$$
\begin{aligned}
\mathcal{E}(X) &= \sum_{i=1}^{\infty} i \ Pr(X = i) \\
&= \sum_{i=1}^{\infty} i \ p_0^{\,i-1}(1-p_0) \\
&= \frac{1}{1-p_0}
\end{aligned}
$$

Exercise

using derivation from lecture 19 page 6. (recall Calculus I trick)

Exercise: think what happens when $p_0 \to 0, \frac{1}{2}, 1$.

We will return to this scenario next class when we discuss "run length coding".