Linear Search

- Given a deck of 52 playing cards, how many do you have to look at to find the Ace of Spades?

Linear Search

- Given a deck of 52 playing cards, how many do you have to look at to find the Ace of Spades?

- Best case?
- Worst case?
- Average case?

Linear Search

- Given a deck of 52 playing cards, how many do you have to look at to find the Ace of Spades?

- Best case: 1
- Worst case: 52
- Average case: ?

Average Case

- To find the average case:
  1. Enumerate all possible cases
  2. Add the number of cards looked at for all of them
  3. Divide by the number of cases

Average Case

1. Enumerate all possible cases

   1, 2, 3, 4 … 50, 51, 52

2. Add the number of cards looked at for all of them

   1 + 2 + … + 52 = (52 + 1) * (52 / 2) = 1378

3. Divide by the number of cases

   1378 / 52 = 26.5

Average Case

1. Enumerate all possible cases

   1, 2, 3, 4 … n-2, n-1, n

2. Add the number of cards looked at for all of them

   $1 + 2 + \ldots + n = (n + 1) * (n / 2) = \frac{1}{2}n(n + 1)$

3. Divide by the number of cases

   $\frac{1}{2}n(n + 1) / n = \frac{1}{2}(n + 1)$

Linear Search

- Given a deck of 52 playing cards, how many do you have to look at to find the Ace of Spades?

- Best case: 1
- Worst case: 52
- Average case: 26.5

- The number of comparisons is a better metric than the time it takes, because people (computers) may be quicker or slower to pick up each card and look at it

Sorted Data

- Now assume the deck of cards is brand new and sorted by suit and number
  - E.g. 2-A♣, 2-A♦, 2-A♥, 2-A♠
  - Best case?
  - Worst case?
  - Average case?

Sorted Data

- Now assume the deck of cards is brand new and sorted by suit and number
  - E.g. 2-A♣, 2-A♦, 2-A♥, 2-A♠
  - Best case: 1
  - Worst case: 1
  - Average case: 1
- Organized data allows for more predictable, more efficient algorithms

Better Search Algorithms

- Given 128 index cards with random numbers on them, to find a particular number (if it is there) with linear search:
  - Best case: 1
  - Worst case: 128
  - Average case: 64.5

- If the numbers are sorted least to greatest, how would you search?

Binary Search

- To search among sorted elements:
  1. Start with the element in the middle
  2. If it matches, done
  3. Else, if it is too large, eliminate all higher elements
  4. Else, if it is too small, eliminate all lower elements
  5. Repeat from step 1

- Same principle as guessing a number

Binary Search

- With every iteration of binary search, you cut the size of the search in half
  - e.g. 128, 64, 32, 16, 8, 4, 2, 1
- 128 can be divided in half 7 times
  - In other terms, $128 = 2^7$
  - So, the worst case number of checks for binary search is:
    - $\log_2 n + 1$
  - With 2 comparisons per check (equal and less than)
    - $2(\log_2 128 + 1) = 16$ comparisons

Why Do We Care?

- Computational complexity
  - Studying types of problems and how hard they are to solve
  - This is a key part of computer science beyond programming
- Classifying problems helps us:
  - Identify general strategies for solving problem types
  - Avoid wasting time developing programs that can't work

Complexity

- Linear search *scales* linearly with the number of items you search over (n)
  - Worst case number of comparisons = n
  - Average case comparisons = ½(n + 1)

Items	Worst case	Average case
10	10	5.5
100	100	50.5
1000	1000	500.5
10000	10000	5000.5

Complexity

- Binary search *scales* logarithmically with the number of items you search over (n)
  - Worst case number of comparisons = $2(\log_2 n + 1)$

Items	Worst case linear	Worst case binary
10	10	9
100	100	15
1000	1000	22
10000	10000	29

- Binary search scales much better than linear search

Complexity and Big-O Notation

- Consider the worst-case complexity of binary search:
    - $2(\log_2 n + 1)$
  - As n gets bigger and bigger, adding 1 and multiplying by 2 become less and less significant
  - We say that binary search has a worst case complexity that is $O(\log_2 n)$
    - Pronounced "Big-O of $\log_2 n$"
  - This allows us to concentrate on a small number of complexity classes

Common Complexity Functions

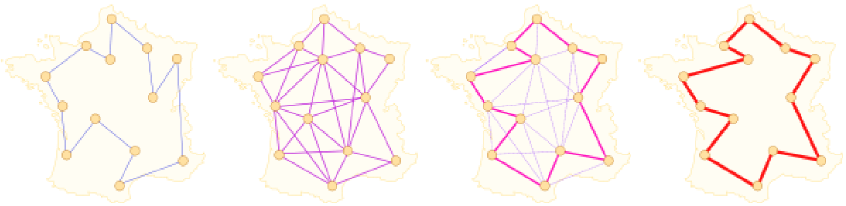
n	$\log_2 n$	$n\log_2 n$	n^2	2^n
2	1	2	4	4
5	2	12	25	32
10	3	33	100	1024
25	5	116	625	33554432
50	6	282	2500	1.13E+15
100	7	664	10000	1.27E+30

On a computer that executes 1 billion instructions per second:

n	n	$\log_2 n$	$n\log_2 n$	n^2	2^n
10	0.01μs	0.003μs	0.033μs	0.1μs	1μs
100	0.1μs	0.007μs	0.664μs	10μs	4×10^{13} years

The Traveling Salesman

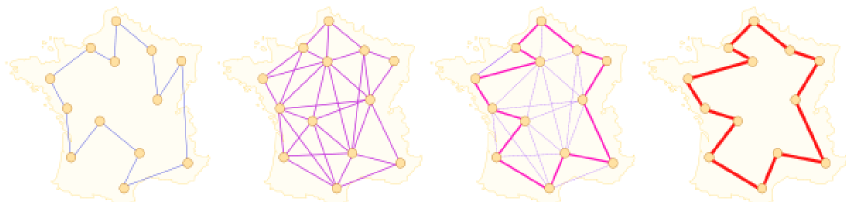
- Consider a salesman who wants to visit a bunch of small towns
  - He has a list of distances between pairs of towns
  - How should he figure out what order to visit them in?
    - How long do you think it would take to figure out the best path for 25 towns?



1 2 3 4

The Traveling Salesman

- Consider a salesman who wants to visit a bunch of small towns
  - He has a list of distances between pairs of towns
  - How should he figure out what order to visit them in?



1 2 3 4

- The brute-force solution to this problem is *O(n!)*, which is even worse than exponential
  - 25 towns = around a billion years on a 2GHz computer
  - Some more clever solutions are $O(n^2)$
- Reasonable solutions can only find an approximately best path