Shuffling and Searching

Dov Kruger

# Topics

SHUFFLING

Linear Search

Linear Search of sorted list

Binary Search

Golden Mean Search

Root Finding

Bisection

Newton

The opposite of sorting is shuffling.

As we discussed, sorting is harder. Why?

O(n2)

O(n log n)

Ω(n) this is the minimum to detect t whether the list is sorted.

1 2 3 4 5 5! = 120

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|  |  |  | 0 |  |  |  | 0 | 0 |  |
| 0 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 8 | 4 | 9 |  |  |  |  |  |  |  |

O(n2)

y← naiveshuffle(x)

for i ← 1 to n //O(n)

do

pick ← rand(0, n-1) //O(1)

while x[pick] < 0 // O(n)

add to y

x[pick] ← -1

end

FIRST TIME: 1/1 (1) +

(n-1)/n (1) + 1/n \*((n-1)/n \* 2 + 1/n\* (...)

((n-1)/n 2 + 1/n ( … ) + (n-2)/n(1) + 2/n(2 …) + 1/n\*1 + (n-1)/n (2 … )

LAST TIME: E = 1/n(1) + (n-1)/n\* (1/n\*2 + (n-1)/n\* ( …. )))) O(n)

1/n ( x) = 1 x = n

first time: O(1)

last time: O(n) = p{1/n}\*n

O(n)

y← FischerYates(x)

n ← x.length

for i ← n-1 to 0

pick ← rand(0, i)

add to y

x[pick] ← x[i]

end

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 10 | 8 | 9 |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 7 |  |  |  |  |  |  |  |  |  |

FischerYatesInPlace(x)

n = x.length

for i ← n-1 to 0

pick ← rand(0, i)

swap (x[pick], x[i])

end

BadShuffle(x)

n = x.length

for i ← n-1 to 0

pick ← rand(0, n-1)

swap (x[pick], x[i])

end

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 10 | 8 | 9 | 7 |

permutations of n object = n!

Summary: Sorting O(n log n) Shuffling: O(n)

Sorting requires us to find one of these solutions.

Shuffling requires us to select any one of the solutions with equal likelihood.

# Searching

Linear Search

Binary Search

Bisection (same as binary search, but in continuous space)

Golden Mean Search

Golden Mean/Max search on a function

Linear Search

O(n) Ω(1) average (if it is present) case n/2 = O(n)

if not there, the average is closer to n

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 9 | 2 | 1 | 3 | 13 | 42 | 11 | 51 | 98 | 22 | -6 | 3 | 14 | 15 | 17 | 19 | -2 | 1 | 6 | 55 |

Linear Search of sorted list O(n)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 9 |  |  |  | 13 | 42 | 11 | 51 | 98 | 22 |  |  | 14 | 15 | 17 | 19 |  |  |  | 55 |
| -6 | -2 | 1 | 1 | 2 | 3 | 3 | 6 | 9 | 11 | 13 | 14 | 15 | 17 | 19 | 22 | 42 | 51 | 55 | 98 |

No better

Binary Search

O(log2 n) Ω(1)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| -6 | -2 | 1 | 1 | 2 | 3 | 3 | 6 | 9 | 11 | 13 | 14 | 15 | 17 | 19 | 22 | 42 | 51 | 55 | 98 |
| i |  |  |  |  |  |  |  |  | x |  |  |  |  |  |  |  |  |  | j |
|  |  |  |  |  |  |  |  |  |  | i |  |  |  |  | x |  |  |  | J |
|  |  |  |  |  |  |  |  |  |  | I |  | X |  | J |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  | I |  | X | J |  |  |  |  |  |  |

binarySearch(x, target)

i← 0

j ← n-1

while i < j

guess ← (i+j)/2

if x[guess] > target

j← guess-1

else if x[guess] < target

i← guess+1

else

return guess

end

return -1

end

// this one is O(1) for targets outside the range of the list. If you are expecting those, this is good

binarySearch(x, target)

if target > x[n-1] or target < x[0]

return -1

i← 0

j ← n-1

while i < j

guess ← (i+j)/2

if x[guess] > target

j← guess-1

else if x[guess] < target

i← guess+1

else

return guess

end

return -1

end

# Bisection

f(x) = x2- 9 roots = +3, -3

bisection(f, 0, 8, .0001)

a

b

## 

## 

## Golden Mean Search

Φ = 1.618

<http://jwilson.coe.uga.edu/EMAT6680Fa06/Hobgood/Kate_files/Golden%20Ratio/GR%20Arch.html>

For any function that is “quadratic” (one-hump)

x

y

f(x) = 8 - x2

a = -3 b = +3

x1=-.708

x2=.708

s

s

a=-3

b=3

s = (b - a) / Φ = 6 / 1.618 = 3.708

x1 = b - s = -.708

x2 = a + s = .708

b = .708

x2=x1 = -.708

s = .708 - - 3 = 3.708

x1 = b - s/Φ = .708 - 3.708 / 1.618 = -1.58

Example 2

Φ = 1.618

f(x) = 8 - x2

a = -4 b = +2

s = (b - a) / Φ = 6/Φ = 3.708

x1 = b - s = -1.708

x2 = a + s = -.292

x1=-1.708

x2=.708

s

s

a=-4

b=2

f(x1) < f(x2)

a = x1 = -1.708

s = (b - a) / Φ = (2 - -1.708)/1.618 = 2.291

x1 = b - s =-.291

x2 = a + s = +.583

O(logɸ  n) = O(log n)

<https://www.cs.cmu.edu/~quake-papers/painless-conjugate-gradient.pdf>

This is optimization for a single hill. To find the best everywhere (global optimization)

heuristics

genetic algorithms

particle algorithms

All have in common: sampling the entire space, and searching locally using something like golden mean

L=0, R=19

phi = 1.618

s = R - L = 19

x2=L + s/phi = 11

s = R - L = 11

x1= R - s/phi = 11/phi = 6

s = R-L = 8

x1=R-s/phi = 8 - 8/phi = 4

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | 5 | 6 | 11 | 92 | 93 | 94 | 50 | 49 | 38 | 36 | 12 | 1 | 0 | 0 | -1 | -5 | -6 | -6 |
| L |  |  |  |  |  |  |  | x1 |  |  | x2 |  |  |  |  |  |  |  | R |
|  |  |  |  |  | x1 |  |  | x2 |  |  | R |  |  |  |  |  |  |  |  |
|  |  |  |  | x1 | x2 |  |  | R |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  | L | x1 |  | x2 | R |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

a = 0 , b = 19

s = (b-a)/1.618 = 11.7 = 12

x1 = b - s

x2 = a + s

f(x) = x2+2 → f’(x) = 2x x = 0

if table[x1] > table[x2]

b = x2

else

a = x1

s = (b-a) / 1.618 = 7.4 = 7

# Root Finding

Bisection: Analogous to binary search in continuous space

linear in the number of bits! (Gets twice as good each iteration, what does this mean?)

sign bit

exponent

f(x) = 0

a

b

bisection(f, a, b, eps)

y1 ← f(a)

y2 ← f(b)

if (y1 \* y2 < 0) //Yay, opposite signs

else

die

do {

guess ← (a + b) / 2

y = f(guess)

if y > 0

b ← guess

else if y < 0

a ← guess

} while abs(b - a) > eps

Newton-Raphson

f(x) = x2-7

f’(x) = 2x

x0 = 3

x1=3 - 2/6 = 2.666666

x0 = 7

x1 = 7 - 47/14 … OMG it works…

x0 = 70

x1 = 70 - 4900/140

Newton does not always converge (if outside the radius of convergence)

Quadratic: double the number of bits each iteration

Numerical Recipes: Ridder’s Algorithm

LFT: Linear Finite Transform