Design and Analysis of Algorithms I

Linear-Time Selection

Randomized Selection (Algorithm)

Prerequisites

Watch this after:

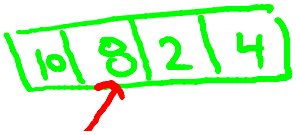
- QuickSort - Partitioning around a pivot
- QuickSort – Choosing a good pivot
- Probability Review, Part I

The Problem

For simplicity

<u>Input</u> : array A with n distinct numbers and a number

<u>Output</u> : i^{th} order statistic (i.e., i^{th} smallest element of input array)



3rd order statistic

<u>Example</u> : median.

($i = (n+1)/2$ for n odd,

$i = n/2$ for n even)

Reduction to Sorting

O(nlog(n)) algorithm

1) Apply MergeSort 2) return i^{th} element of sorted array

Fact : can't sort any faster [see optional video]

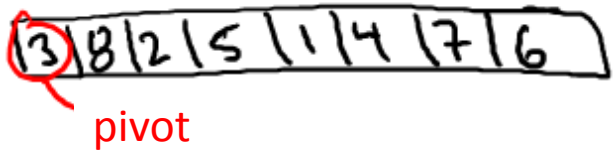
Next : O(n) time (randomized) by modifying Quick Sort.

Optional Video : O(n) time deterministic algorithm.
-- pivot = "median of medians" (warning : not practical)

Partitioning Around a Pivot

<u>Key Idea</u> : partition array around a pivot element.

-Pick element of array



-Rearrange array so that

-Left of pivot => less than pivot

-Right of pivot => greater than pivot

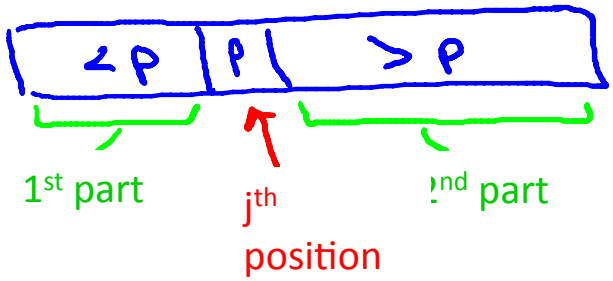
<u>Note</u> : puts pivot in its "rightful position".

Suppose we are looking for the 5th order statistic in an input array of length 10. We partition the array, and the pivot winds up in the third position of the partitioned array. On which side of the pivot do we recurse, and what order statistic should we look for?

- The 3rd order statistic on the left side of the pivot.
- The 2nd order statistic on the right side of the pivot.
- The 5th order statistic on the right side of the pivot.
- Not enough information to answer question – we might need to recurse on the left or the right side of the pivot.

Randomized Selection

Rselect (array A, length n, order statistic i)



0) if n = 1 return A[1]
1) Choose pivot p from A uniformly at random
2) Partition A around p
   let j = new index of p
3) If j = i, return p
4) If j > i, return Rselect(1st part of A, j-1, i)
5) [if j<i] return Rselect (2nd part of A, n-j, i-j)

Properties of RSelect

<u>Claim</u> : Rselect is correct (guaranteed to output ith order statistic)

<u>Proof</u> : by induction. [like in optional QuickSort video]

<u>Running Time ?</u> : depends on "quality" of the chosen pivots.

Ter
ver

What is the running time of the RSelect algorithm if pivots are always chosen in the worst possible way?

- $\theta(n)$
- $\theta(n \log n)$
- $\theta(n^2)$
- $\theta(2^n)$

<u>Example</u> :

-- suppose i = n/2
-- suppose choose pivot = minimum every time
=> $\Omega(n)$ time in each of $\Omega(n)$ recursive calls

Running Time of RSelect?

Running Time ? : depends on which pivots get chosen.
(could be as bad as $\theta(n^2)$)

Key : find pivot giving "balanced" split.

Best pivot: the median ! (but this is circular)

$\Rightarrow$Would get recurrence $T(n) \leq T(n/2) + O(n)$
$\Rightarrow$ T(n) = O(n) [case 2 of Master Method]
Hope : random pivot is "pretty good" "often enough"

Running Time of RSelect

Rselect Theorem : for every input array of length n, the average running time of Rselect is O(n)

-- holds for every input [no assumptions on data]

-- "average" is over random pivot choices made by the algorithm

Design and Analysis of Algorithms I

Linear-Time Selection

Randomized Selection (Analysis)

Running Time of RSelect

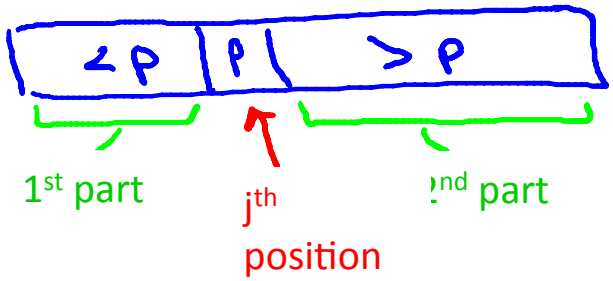
Rselect Theorem : for every input array of length n, the average running time of Rselect is O(n)

-- holds for every input [no assumptions on data]

-- “average” is over random pivot choices made by the algorithm

Randomized Selection

Rselect (array A, length n, order statistic i)



0) if n = 1 return A[1]
1) Choose pivot p from A uniformly at random
2) Partition A around p
   let j = new index of p
3) If j = i, return p
4) If j > i, return Rselect(1st part of A, j-1, i)
5) [if j<i] return Rselect (2nd part of A, n-j, i-j)

Proof I: Tracking Progress via Phases

Note : Rselect uses <= cn operations outside of recursive call [for some constant c > 0] [from partitioning]

Notation : Rselect is in phase j if current array size between $(\frac{3}{4})^{j+1} \cdot n$ and $(\frac{3}{4})^{j} \cdot n$

-X_j = number of recursive calls during phase j

# of phase j subproblems

Note : running time of RSelect $\leq \sum_{phases\ j} X_j \cdot c \cdot (\frac{3}{4})^j \cdot n$

<= array size during phase j

Work per phase j subproblem

Tim Roughgarden

Proof II: Reduction to Coin Flipping

X_j = # of recursive calls during phase j → Size between $(\frac{3}{4})^{j+1} \cdot n$ and $(\frac{3}{4})^{j} \cdot n$

Note : if Rselect chooses a pivot giving a 25 – 75 split (or better) then current phase ends !
(new subarray length at most 75 % of old length)

Recall : probability of 25-75 split or better is 50%

So : $E[X_j]$ <= expected number of times you need to flip a fair coin to get one "heads"
(heads ~ good pivot, tails ~ bad pivot)

Proof III: Coin Flipping Analysis

Let N = number of coin flips until you get heads.

(a "geometric random variable")

Note : $E[N] = 1 + (1/2)*E[N]$

1st coin flip

Probability of tails

# of further coin flips needed in this case

Solution : $E[N] = 2$ (Recall $E[X_j] <= E[N]$)

Putting It All Together

Expected running time of RSelect

$$\leq E[cn \sum_{phase\ j} (\frac{3}{4})^j X_j] \qquad (*)$$

$$= cn \sum_{phase\ j} (\frac{3}{4})^j \boxed{E[X_j]} \qquad \text{[LIN EXP]}$$

= E[# of coin flips N] = 2

$$\leq 2cn \sum_{phase\ j} \boxed{(\frac{3}{4})^j}$$

geometric sum,
<= 1/(1-3/4) = 4

$$\leq 8cn = O(n)$$

Q.E.D.

Tim Roughgarden

Design and Analysis of Algorithms I

Linear-Time Selection

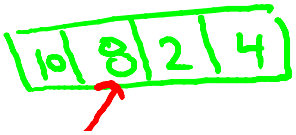
Deterministic Selection (Algorithm)

The Problem

For simplicity

<u>Input</u> : array A with n distinct numbers and a number

<u>Output</u> : i^{th} order statistic (i.e., i^{th} smallest element of input array)



3rd order statistic

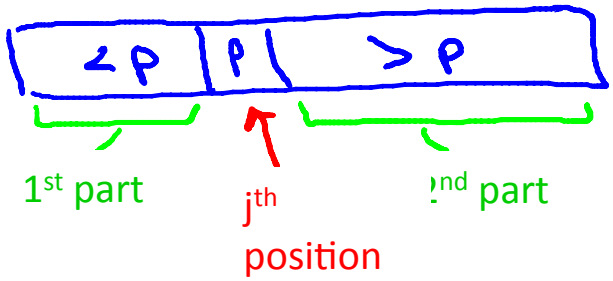
<u>Example</u> : median.

($i = (n+1)/2$ for n odd,

$i = n/2$ for n even)

Randomized Selection

Rselect (array A, length n, order statistic i)



0) if n = 1 return A[1]
1) Choose pivot p from A uniformly at random
2) Partition A around p
   let j = new index of p
3) If j = i, return p
4) If j > i, return Rselect(1st part of A, j-1, i)
5) [if j<i] return Rselect (2nd part of A, n-j, i-j)

Guaranteeing a Good Pivot

Recall : "best" pivot = the median ! (seems circular!)

Goal : find pivot guaranteed to be pretty good.

Key Idea : use "median of medians"!

A Deterministic ChoosePivot

ChoosePivot(A,n)

-- logically break A into n/5 groups of size 5 each
-- sort each group (e.g., using Merge Sort)
-- copy n/5 medians (i.e., middle element of each sorted group)
   into new array C
-- recursively compute median of C (!)
-- return this as pivot

The DSelect Algorithm

DSelect(array A, length n, order statistic i)

1. Break A into groups of 5, sort each group
2. C = the n/5 "middle elements"
3. p = DSelect(C,n/5,n/10) [recursively computes median of C]
4. Partition A around p
5. If j = i return p
6. If j < i return DSelect(1st part of A, j-1, i)
7. [else if j > i] return DSelect(2nd part of A, n-j, i-j)

ChoosePivot

Same as before

Ter
ver

How many recursive calls does DSelect make?

- 0
- 1
- 2
- 3

Running Time of DSelect

Dselect Theorem : for every input array of length n, Dselect runs in O(n) time.

Warning : not as good as Rselect in practice

1) Worse constraints 2) not-in-place

History : from 1973

Blum – Floyd – Pratt – Rivest – Tarjan

('95) ('78) ('02) ('86)

Design and Analysis of Algorithms I

Linear-Time Selection

Deterministic Selection (Analysis II)

Rough Recurrence (Revisited)

Let T(n) = maximum running time of Dselect on an input array of length n.

There is a constant c >= 1 such that :

1. T(1) = 1
2. T(n) <= c*n + T(n/5) + T(?)

<= 7n/10 by Key Lemma

sorting the groups partition

recursive call in line 3

recursive call in line 6 or 7

Rough Recurrence (Revisited)

$T(1) = 1$, $T(n) <= cn + T(n/5) + T(7n/10)$

Constant $c>=1$

Note : different-sized subproblems => can't use Master Method!

Strategy : "hope and check"

Hope : there is some constant a [independent of n]
Such that $T(n) <= an$ for all $n >=1$
[if true, then $T(n) = O(n)$ and algorithm is linear time]

Analysis of Rough Recurrence

T(1) = 1 ; T(n) <= cn + T(n/5) + T(7n/10)

Constant c>=1

Claim : Let a = 10c

Then $\boxed{\text{T(n) <= an for all n >= 1}}$

=> Dselect runs in O(n) time

Proof : by induction on n

Base case : T(1) = 1 <= a*1 (since a >= 1)

Inductive Step : [n > 1]

Inductive Hypothesis : $T(k) \leq ak \ \forall \ k < n$

We have $T(n) \underset{\text{GIVEN}}{\leq} cn + T(n/5) + T(7n/10)$

$$\underset{\text{IND HYP}}{\leq} cn + a(n/5) + a(7n/10)$$

$$= n(c + 9a/10) \underset{\text{Choice of a}}{=} an$$

Q.E.D.

Tim Roughgarden

Design and Analysis
of Algorithms I

Linear-Time Selection

An $\Omega(n \log n)$ Sorting Lower Bound

A Sorting Lower Bound

Theorem : every "comparison-based" sorting algorithm has worst-case running time $\Omega(n \log n)$

[assume deterministic, but lower bound extends to randomized]

Comparison-Based Sort : accesses input array elements only via comparisons ~ "general purpose sorting method"

Examples : Merge Sort, Quick Sort, Heap Sort

Good for data from distributions

good for small integers

Non Examples : Bucket Sort, Counting Sort, Radix Sort

good for medium-size integers

Tim Roughgarden

Proof Idea

Fix a comparison-based sorting method and an array length n

⇒Consider input arrays containing {1,2,3,…,n} in some order.

⇒ n! such inputs

Suppose algorithm always makes <= k comparisons to correctly sort these n! inputs.

=> Across all n! possible inputs, algorithm exhibits <= 2^k distinct executions

i.e., resolution of the comparisons

Proof Idea (con'd)

By the Pigeonhole Principle : if $2^k < n!$, execute identically on two distinct inputs => must get one of them incorrect.

So : Since method is correct,

$$2^k \geq n!$$
$$\geq (\frac{n}{2})^{\frac{n}{2}}$$
$$=> k \geq \frac{n}{2} \cdot \log_2 \frac{n}{2} = \Omega(n \log n)$$